RAMBO II: Rapidly Reconfigurable Atomic Memory for Dynamic Networks [*]

Seth Gilbert[1]
sethg@theory.lcs.mit.edu

Nancy Lynch[1]
lynch@theory.lcs.mit.edu

Alex Shvartsman[1,2]
alex@theory.lcs.mit.edu

MIT Laboratory for Computer Science[1]
200 Technology Square
Cambridge, MA 02139, USA

Dept. of Computer Science and Engineering[2]
191 Auditorium Road, Unit 3155
University of Connecticut, Storrs, CT 06269

Abstract

This paper presents a new algorithm implementing reconfigurable atomic read/write memory for highly dynamic environments. The original RAMBO *algorithm, recently developed by Lynch and Shvartsman [15, 16], guarantees atomicity for arbitrary patterns of asynchrony, message loss, and node crashes.* RAMBO II *implements a different approach to establishing new configurations: instead of operating sequentially, the new algorithm reconfigures aggressively, transferring information from old configurations to new configurations in parallel. This improvement substantially reduces the time to establish a new configuration and to remove obsolete configurations. This, in turn, substantially increases fault tolerance and reduces the latency of read/write operations when the network is unstable or reconfiguration is bursty. This paper presents* RAMBO II, *a correctness proof, and a conditional analysis of its performance. Preliminary empirical studies illustrate the advantages of the new algorithm.*

1. Introduction

In the future, large scale distributed applications will rely on a multitude of communicating, computing devices. These devices will operate in complex, dynamic environments, for example, in major civilian rescue efforts and extensive military maneuvers. These devices will facilitate the coordination of personnel and equipment, as they gather data, store it in survivable repositories, and provide timely and coherent information.

These applications often require data objects with atomic read/write semantics, and the data is often replicated to guarantee fault tolerance and availability. Replication, however, introduces two problems: (i) maintaining *consistency* among the replicas, and (ii) *dynamic participation* – managing the replicas as the participants change.

Consistency. Beginning with the work of Gifford [8] and Thomas [21], many algorithms use collections of intersecting sets to solve the *consistency* problem for replicated data. Such intersecting sets are called *quorums*. Upfal and Wigderson [22] use majority sets of readers and writers to emulate shared memory in a distributed setting. Vitányi and Awerbuch [23] implement multi-writer/multi-reader registers using matrices of single-writer/single-reader registers where the rows and the columns are written and respectively read by specific processors. Attiya, Bar-Noy and Dolev [2] use majorities of processors to implement single-writer/multi-reader objects in message passing systems. Such algorithms assume a static set of processors and rely on a static definition of the quorum systems.

Dynamic Participation. We call a set of quorums used to manage data replicas a *configuration*. In long-lived systems, where processors frequently join and leave, it is necessary to occasionally reconfigure, choosing a new set of quorums. This operation relocates the replicas to non-failed nodes, adapting to a new set of participants. Prior approaches [6, 11, 5, 10, 20] require the new quorums to include processors from the old quorums, restricting the choice of a new configuration. This restriction is a static constraint that needs to be satisfied by the quorum system even before the reconfiguration. In our work on reconfigurable atomic memory [18, 7, 15, 16], we replace the *space-domain* requirement on successive configurations with a *time-domain* requirement: some quorums from both the old and the new system are involved in the reconfiguration algorithm. Such systems are more dynamic because they place fewer restrictions on the choice of a new configuration: there is no requirement that successive configura-

[*]This work was supported in part by NSF ITR Grant CCR-0121277, NSF Grants 9804665, 9988304, 64961-CS, NSF CAREER Award 9984778, AFOSR Contract #F49620-00-1-0097, DARPA Contract #F33615-01-C-1896, and NTT Grant MIT9904-12.

tions intersect.

Some earlier algorithms [18, 7] designate a single distinguished process to initiate all reconfigurations. The advantage of the single-reconfigurer approach is its relative simplicity and efficiency. The disadvantage is that the reconfigurer is a single point of failure – no further reconfiguration is possible if the designated node fails.

Virtually synchronous services [3], and group communication services in general [1], can also be used to implement an atomic data service, e.g., by implementing a global totally ordered broadcast. In most GCS implementations, forming a new view takes a substantial amount of time, and client-level operations are delayed during the view-formation period. In our algorithm, reads and writes can make progress during reconfiguration.

RAMBO. To solve this problem, Lynch and Shvartsman introduced an algorithm called RAMBO: Reconfigurable Atomic Memory for Basic Objects [15, 16]. The RAMBO algorithm guarantees atomicity in any asynchronous execution, in the presence of arbitrary process crashes and network failures. Moreover, the algorithm allows any member of the most recently established configuration to begin reconfiguration to a new quorum system. Conditional performance analysis shows that under certain conditions when the underlying network is stable and reconfiguration is not too frequent, read and write operations complete rapidly.

However, allowing multiple reconfigurers introduces the problem of maintaining multiple configurations and removing old configurations. RAMBO implements a sequential "garbage collection" algorithm where nodes remove obsolete configurations one at a time, in order, until only the most recently established configuration remains. However, when communication is unreliable, or when reconfiguration is frequent, RAMBO may perform poorly. The number of active configurations may grow without bound, leading to high message complexity and limited fault tolerance.

The New Algorithm. The primary contribution of this paper is a new algorithm for reconfigurable atomic memory, called RAMBO II. This algorithm includes a new configuration management protocol that operates aggressively in parallel: it can establish a new configuration, removing an arbitrary number of old configurations, in constant time, under certain reasonable assumptions.

Our conditional performance analysis shows that if a process knows about a sequence of h non-failed configurations, then RAMBO II removes all but one of these configurations in time $\Theta(1)$, when the message delay is bounded. By comparison, in the original RAMBO, the process needs to invoke h garbage collection operations sequentially, taking $\Theta(h)$ time. The new algorithm also reduces the number of messages necessary to process these configurations.

We formally specify RAMBO II using I/O automata [17]. We show correctness (atomicity) of the algorithm for arbitrary asynchrony and failures. We analyze performance *conditionally*, based on certain failure and timing assumptions. Assuming that gossip and configuration upgrades occur periodically, and that quorums of active configurations do not fail, we show that read and write operations complete within time $8d$, where d is the maximum message latency.

Implementations of RAMBO and RAMBO II on a LAN are currently being completed [19]. Preliminary empirical studies performed using this implementation illustrate the advantages of RAMBO II in settings where reconfiguration is bursty and messages may be lost.

Document Structure. In Section 2 we review the original RAMBO algorithm [15], and in Section 3 present RAMBO II. In Section 4 we prove that the new algorithm is correct. In Section 5 we present a conditional latency analysis. In Section 6 we overview an implementation and present preliminary empirical results. Finally, in Section 7 we summarize our results, and describe areas for future research. A complete technical report [9] contains the full results.

2. The Original RAMBO Algorithm

We now present the original RAMBO algorithm. RAMBO replicates data at several network locations in order to achieve fault tolerance and availability. The algorithm uses *configurations* to maintain consistency in the presence of small and transient changes. Each configuration consists of a set of *members* plus sets of *read-quorums* and *write-quorums*. The quorum intersection property requires that every read-quorum intersect every write-quorum. RAMBO supports *reconfiguration*, which modifies the set of members and the sets of quorums, thereby accommodating larger and more permanent changes without violating atomicity. Any quorum configuration may be installed at any time – no intersection requirement is imposed on the sets of members or on the quorums of distinct configurations.

The RAMBO algorithm consists of three kinds of automata: (i) *Joiner* automata, which handle join requests, (ii) *Recon* automata, which handle reconfiguration requests, and generate a totally ordered sequence of configurations, and (iii) *Reader-Writer* automata, which handle read and write requests, manage garbage collection, and send and receive gossip messages.

In this paper, we discuss only the *Reader-Writer* automaton. The *Joiner* automaton is quite simple; it sends a join message when node i joins, and sends a join-ack message in response to join messages. The *Recon* automaton depends on a consensus service, implemented using Paxos [12], to agree on a total ordering of configurations. However, we assume that this total ordering exists, and therefore need not discuss this automaton any further. For more details of these two automata, see the original RAMBO paper [15, 16].

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:

$\mathsf{join\text{-}ack}(\mathsf{rambo})_i, i \in I$

$\mathsf{read\text{-}ack}(v)_i, v \in V, i \in I$

$\mathsf{write\text{-}ack}_i, i \in I$

$\mathsf{recon\text{-}ack}(b)_i, b \in \{ok, nok\}, i \in I$

$\mathsf{report}(c)_i, c \in C, i \in I$

Figure 1. RAMBO**: External signature**

The external signature for RAMBO appears in Figure 1. The algorithm is specified for a single memory location, and extended to implement a complete shared memory. A client uses the join_i action to join the system. After receiving a $\mathsf{join\text{-}ack}_i$, the client can issue read_i and write_i requests, which results in $\mathsf{read\text{-}ack}_i$ and $\mathsf{write\text{-}ack}_i$ responses. The client can issue a recon_i request to propose a new configuration. Finally, the fail_i action is used to model node i failing. For the detailed code, see the full presentation in [16].

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

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

Garbage collection operations remove old configurations from the system. A garbage collection operation involves two configurations: the old configuration being removed and the new configuration being established. A garbage collection operation requires two phases, a query phase and a propagation phase. The first phase contacts a read-quorum and a write-quorum from the old configuration, and the second phase contacts a write-quorum from the new configuration. All three operations, *read*, *write*, and *garbage collection*, are implemented using gossip messages.

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

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

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

3. Rapidly Reconfigurable Atomic Memory

In this section we present RAMBO II. As before, the algorithm is specified for a single memory location. In RAMBO, configurations are upgraded sequentially. Configuration $c(k)$ can be upgraded only if configuration $c(k-1)$ has been previously upgraded. RAMBO II, however, implements a new reconfiguration protocol that can upgrade any configuration, even if configurations with smaller indices have not been upgraded.

After RAMBO II upgrades a configuration, all configurations with smaller indices can be removed. Thus a single configuration upgrade operation in RAMBO II poten-

Signature:

As in RAMBO, with the following modifications:

Internal:

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

Configuration Management Transitions:

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

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

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

Configuration Management State:

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

upg, a record with fields:

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

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

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

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

Figure 2. $Reader\text{-}Writer_i$**: Configuration-Management transitions**

tially has the effect of many garbage collection operations in RAMBO. We have changed the name to emphasize the operation's active role in configuration management: configuration upgrade is an inherent part of preparing a configuration to assume responsibility for the data. The code for the new configuration management mechanism appears in Figure 2. All labeled lines in this section refer to the code therein.

We now describe in more detail the configuration upgrade operation, which is at the heart of RAMBO II. A configuration upgrade operation is initiated at node i with a $\mathsf{cfg\text{-}upgrade}(k)$ event. When this happens, $cmap(k)$ must be defined, that is, must be a valid configuration $\in C$ (line A). Additionally, for every configuration $\ell < k$, $cmap(\ell)$ must be either $\in C$ or removed, that is, $\pm$ (line B).

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

As a result of the $\mathsf{cfg\text{-}upgrade}$ event, node i initializes its upg state (line C), and begins the query phase of the upgrade operation. In particular, node i stores its current $cmap$ in $upg.cmap$, which records the configurations that were active when the operation began. Only these configurations matter during the operation; new configurations are ignored.

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

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

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

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

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

The new algorithm introduces several difficulties not present in RAMBO. Consider, for example, a nice property guaranteed by the sequential garbage collection algorithm in RAMBO: every configuration is upgraded before it is removed. In RAMBO II, on the other hand, some configurations never receive up to date information.

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

For the rest of this paper, we restrict our attention to "good" executions of the algorithm. We assume, for example, that requests are well formed: after a client issues a join request, it waits until receiving a join-ack before issuing further requests; each client issues only one read or write operation at a time, etc. The formal details of these assumptions are specified in [9].

4. Atomic Consistency

This section discusses the proof of atomic consistency. For the full, formal version, including proofs omitted here, see [9]. We focus here on proving one key theorem, which implies the main result. Assume α is an arbitrary, good execution of the algorithm. We show that the tags associated with the values induce a partial-order, $\prec$, on the read and write operations in α with the following properties: (i) $\prec$ totally orders all write operations in α, (ii) $\prec$ orders every read operation in α with respect to every write operation in α, (iii) for each read operation, if there is no preceding write operation in $\prec$, then the read operation returns the initial value; otherwise, the read operation returns the value of the unique write operation immediately preceding it in $\prec$, and (iv) if some operation, θ, completes before another operation, ϕ, begins in α, then ϕ does not precede θ in $\prec$. This leads almost immediately to the conclusion that the algorithm guarantees atomic consistency, using Lemma 13.16 in [14]. We discuss in detail the proof of Property (iv), as the other three properties are self-evident.

In the theorem and proof sketch below, the following terms are used. For every read or write operation π, the query-fix(π) event occurs immediately after the query phase of π completes, and the prop-fix(π) event occurs immediately after the propagation phase of π completes. We define cfg-upg-query-fix and cfg-upg-prop-fix analogously with respect to configuration upgrade operations.

The $query\text{-}cmap$ of a read or write operation, π, initiated at node i is a map from integer indices to $C \cup \{\bot, \pm\}$. It is initially undefined, and set to $op.cmap_i$ when the $query\text{-}fix(\pi)$ event occurs. Similarly, the $prop\text{-}cmap$ of operation π initiated at node i is set to $op.cmap_i$ when the $prop\text{-}fix(\pi)$ event occurs.

The query-phase-start(π) event is defined to be the event that initiates the collection of query results, that is, the event that sets $op.acc$ to $\emptyset$ for the last time prior to the query-fix(π) event. The query-phase-start(π) event, then, is either the read or write event that initiates operation π, or the last rcv event that causes π to restart the query phase.[1] Intuitively, one can think of the query-phase-start(π) as the event that marks the beginning of the query phase of π.

For every read or write operation, π, at node i, we define $tag(\pi)$ to be the value of $op.tag_i$ when the query-fix(π) event occurs. Intuitively, if π is a read operation then $tag(\pi)$ is the largest tag discovered by node i during the query phase. If π is a write operation then $tag(\pi)$ is the new tag chosen by node i that is larger than any tag discovered by i during the preceding query phase. Similarly, for a configuration upgrade operation γ at node i, we define $tag(\gamma)$ to be the tag at node i when the $\mathit{cfg\text{-}upg\text{-}query\text{-}fix}(\gamma)$ event occurs, that is, the largest tag discovered at node i during the query phase of γ.

In the following discussion, we assume that π_1 and π_2 are two read or write operations that occur at i_1 and i_2 respectively, and that π_1 completes before π_2 begins in α. The goal of this section is to show that $tag(\pi_1) \leq tag(\pi_2)$, and that the inequality is strict if π_2 is a write operation.

The first case to consider is when $\mathit{prop\text{-}cmap}(\pi_1) \cap \mathit{query\text{-}cmap}(\pi_2) \neq \emptyset$. Then the following lemma indicates that the result holds.

Lemma 4.1 *Assume π_1 and π_2 are two read or write operations such that the* prop-fix *event of π_1 precedes the* query-phase-start(π_2) *event in α. If $\mathit{prop\text{-}cmap}(\pi_1) \cap \mathit{query\text{-}cmap}(\pi_2) \neq \emptyset$, then $tag(\pi_1) \leq tag(\pi_2)$ and if π_2 is a write operation, then $tag(\pi_1) < tag(\pi_2)$.*

Proof (sketch). There exists some configuration c in the intersection of $\mathit{prop\text{-}cmap}(\pi_1)$ and $\mathit{query\text{-}cmap}(\pi_2)$. Therefore there exists some read-quorum R of configuration c, and some write-quorum, W, of configuration c, such that π_1 updates W before π_2 reads from R. Then by the quorum intersection property, the result follows. $\Box$

Next, consider the case where $\min(\mathit{prop\text{-}cmap}(\pi_1)) > \max(\mathit{query\text{-}cmap}(\pi_2))$. It turns out, by the following lemma, that this case is impossible: a later operation always learns of a configuration at least as large as that of an earlier operation.

Lemma 4.2 *Assume π_1 and π_2 are two read or write operations such that the* prop-fix *event of π_1 precedes the* query-phase-start *event of π_2 in α. Then: $\min(\{\ell : \mathit{prop\text{-}cmap}(\pi_1)(\ell) \in C\}) \leq \max(\{\ell : \mathit{query\text{-}cmap}(\pi_2)(\ell) \in C\})$.*

[1]Operation phases must occasionally restart when the initiating node has a severely out of date $cmap$. This restart mechanism, while relevant to the detailed proof, is not discussed in this paper.

The only remaining case is when $\mathit{prop\text{-}cmap}(\pi_1)$ and $\mathit{query\text{-}cmap}(\pi_2)$ are disjoint and $\max(\mathit{prop\text{-}cmap}(\pi_1)) < \min(\mathit{query\text{-}cmap}(\pi_2))$. The rest of the discussion assumes this relationship between the $cmap$s, and shows the appropriate relationship between the tags.

Let s_2 be the index of the smallest configuration in $\mathit{query\text{-}cmap}(\pi_2)$. Then the following lemma demonstrates that there exists a configuration upgrade operation, γ, that precedes π_2, and that upgrades configuration $c(s_2)$.

Lemma 4.3 *Let π_2 be a read or write operation whose* query-fix *event occurs in α. Let s_2 be the smallest index such that $\mathit{query\text{-}cmap}(\pi_2)(s_2) \in C$. Assume $s_2 > 0$. Then there exists a configuration upgrade operation γ that upgrades configuration $c(s_2)$, such that the* cfg-upg-prop-fix(γ) *event precedes the* query-phase-start(π_2) *event in α.*

Proof (sketch). This follows from the realization that if s_2 is the smallest non-removed configuration, then there must have been a prior operation that upgraded s_2 and removed the smaller configurations. $\Box$

Lemma 4.3 implies that $tag(\gamma) \leq tag(\pi_2)$: the former accesses a write-quorum of $c(s_2)$ in the propagation phase, and the latter access a read-quorum of $c(s_2)$ in the query phase. By the quorum intersection properties of read and write quorums, and the fact that γ completes before π_2 begins, the claim follows. Similarly, it follows that if π_2 is a write operation, $tag(\gamma) < tag(\pi_2)$.

Next we construct a sequence, possible containing repeated elements, of configuration upgrade operations, $\gamma_0, \ldots, \gamma_{s_2-1}$, where $\gamma_{s_2-1} = \gamma$, with the following properties: First, for all $s < s_2$, configuration upgrade operation γ_s removes configuration $c(s)$. Next, consider two elements in the sequence, γ_s and γ_{s+1}, for all $s < s_2 - 1$. If γ_s and γ_{s+1} are distinct upgrade operation then (i) γ_s completes before γ_{s+1} begins, and (ii) the configuration that is the target of γ_s is removed by γ_{s+1}. These last two properties allow us to use the sequence to ensure the propagation of tags: if two consecutive upgrade operations, γ_s and γ_{s+1}, are distinct operations, then the tag and value are passed from the former to the latter; the former accesses a write-quorum of $target(\gamma_s)$, and the latter accesses a read-quorum of $target(\gamma_s)$.

Lemma 4.4 *If a* cfg-upgrade$_i$ *event for configuration upgrade operation γ occurs in α such that $s_2 - 1 \in \mathit{removal\text{-}set}(\gamma)$, then there exists a sequence (possibly containing repeated elements) of configuration upgrade operations $\gamma_0, \gamma_1, \ldots, \gamma_{s_2-1}$, where $\gamma_{s_2-1} = \gamma$, with one element in the sequence for every configuration index $< s_2$, with the following properties:*

1. *$\forall\, s : 0 \leq s < s_2,\ s \in \mathit{removal\text{-}set}(\gamma_s)$.*

2. *$\forall\ s : 0 \leq s < s_2 - 1$, if $\gamma_s \neq \gamma_{s+1}$, then a* cfg-upg-prop-fix *event of γ_s and a* cfg-upgrade *event of γ_{s+1} occur in α, and the* cfg-upg-prop-fix *event of γ_s precedes the* cfg-upgrade *event of γ_{s+1}.*

3. *$\forall\ s : 0 \leq s < s_2 - 1$, if $\gamma_s \neq \gamma_{s+1}$, then $target(\gamma_s) \in removal\text{-}set(\gamma_{s+1})$.*

Proof (sketch). We construct the sequence in reverse order, first defining $\gamma_{s_2-1} = \gamma$, and then at each step defining the preceding element. The induction is a backward induction on ℓ, for $\ell = s_2 - 1$ down to $\ell = 0$, maintaining the three properties at each step of the induction.

For the inductive step, we assume that γ_ℓ has been defined and that the three hypotheses hold for $s \geq \ell$. We proceed to define $\gamma_{\ell-1}$. If γ_ℓ removes configuration $c(\ell - 1)$, then let $\gamma_{\ell-1} = \gamma_\ell$, and all the properties hold for $s = \ell - 1$.

On the other hand, if configuration $c(\ell)$ is the configuration with the smallest index removed by γ_ℓ, then there is an upgrade operation, $\gamma_{\ell-1}$, that removes configuration $c(\ell - 1)$, does not remove $c(\ell)$ and completes before γ_ℓ begins. It follows immediately that Properties 1 and 2 are satisfied for $s = \ell - 1$. Since $\gamma_{\ell-1}$ does not remove $c(\ell)$, $c(\ell)$ is the target of the upgrade operation and Property 3 follows. □

As discussed above, the properties of this sequence immediately imply that tags are monotonic: later configuration upgrade operations in the sequence have tags no smaller than earlier configuration upgrade operations.

Lemma 4.5 *Let $\gamma_\ell, \ldots, \gamma_k$ be a sequence of configuration upgrade operations constructed by Lemma 4.4. Then $\forall\ s : 0 \leq s < k,\ tag(\gamma_s) \leq tag(\gamma_{s+1})$.*

We now state and prove the main theorem:

Theorem 4.6 *Assume π_1 and π_2 are two read or write operations, such that the* prop-fix *event of π_1 precedes the* query-phase-start(π_2) *event in α. Then $tag(\pi_1) \leq tag(\pi_2)$, and if π_2 is a write then $tag(\pi_1) < tag(\pi_2)$.*

Proof. Assume that π_1 and π_2 occur at i_1 and i_2 respectively. By assumption, π_1 completes before π_2 begins. Define $cm_1 = prop\text{-}cmap(\pi_1)$, and $cm_2 = query\text{-}cmap(\pi_2)$. If both operations share a configuration, Lemma 4.1 implies the result. Assume, then, that cm_1 and cm_2 are disjoint. Let s_1 be the largest element in cm_1, and let s_2 be the smallest element in cm_2. Then Lemma 4.2 implies that $s_1 < s_2$.

Lemma 4.3 defines upgrade operation γ, which precedes π_2. Construct a sequence of configuration upgrade operations using Lemma 4.4, $\gamma_0, \ldots, \gamma_{s_2-1}$, where $\gamma_{s_2-1} = \gamma$. We now consider γ_{s_1} from this sequence. Lemma 4.5 implies that $tag(\gamma_{s_1}) \leq tag(\gamma_{s_2-1})$. We now show that the tag of π_1 must be no larger than the tag of γ_{s_1}.

The propagation phase of π_1 accesses a write-quorum W of configuration $c(s_1)$, whereas the query phase of γ_{s_1} accesses a read-quorum R of configuration $c(s_1)$. Since $W \cap R \neq \emptyset$, we may fix some $j \in W \cap R$. Let m_1 be the message sent from j to i_1 in the propagation phase of γ_{s_1}. Let m_2 be the message sent from j to the process running γ_{s_1} in the query phase of γ_{s_1}.

We claim that j sends m_1, its message for π_1, before it sends m_2, its message for γ_{s_1}. If not, then the information about configuration $s_1 + 1$ would be conveyed by j to i_1, who would include it in cm_1, contradicting the choice of s_1.

Since j sends m_1 before it sends m_2, j conveys tag information from π_1 to γ_{s_1}, ensuring that $tag(\pi_1) \leq tag(\gamma_{s_1})$. We already showed that $tag(\gamma_{s_1}) \leq tag(\pi_2)$, and if π_2 is a write operation then $tag(\gamma_{s_1}) < tag(\pi_2)$. Combining the inequalities yields both conclusions. □

Having shown that the tags are monotonically increasing, it follows immediately that the tags induce a partial-order $\prec$ that meets the necessary and sufficient requirements for atomic consistency.

5. Conditional Performance Analysis

In this section, we discuss the latency bounds for the new algorithm. We show that RAMBO II allows the system to recover rapidly after a period of unreliable network connectivity or bursty reconfigurations. For a full statement of the results, and for the proofs, omitted here, see [9].

We first define what it means for a system to stabilize and exhibit "normal behavior" from some point onward. Let d be the maximum message delay bound when the network is stable. The constant d is also the interval at which gossip messages are sent. Assume α is an admissible timed execution and α' a finite prefix of α. Define $\ell time(\alpha')$ to be 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 is delivered, then it is delivered by the maximum of time $t + d$ and time $\ell time(\alpha') + d$.

The analysis takes into account actions of the *Recon* automaton, which we do not present in this paper. In particular, the *Recon* automaton produces decide and report events when consensus is reached on a new configuration; the former occur at members of the old configuration, the latter at nodes that learn of the new configuration.

Configuration-Viability. As in all quorum-based algorithms, liveness depends on all the nodes in some quorums remaining alive. We say that a configuration $c(k)$ is *installed* when a decide$(c(k))$ event has occurred at every non-failed member of the prior configuration, $c(k - 1)$, notifying it that configuration $c(k)$ has been accepted by the *Recon* service. We say that an execution α is *(α',e,τ)-configuration-viable* if for every configuration, $c(k)$, that is chosen, there exists a read-quorum, R, and a write-quorum, W, such that no process in $R \cup W$ fails before the maximum of (i) time τ after every configuration $\leq k+1$ is installed, and (ii) $\ell time(\alpha') + e + \tau$.

By assuming that an execution is *(α',e,τ)-configuration-viable*, we ensure that the algorithm has time $\geq \tau$ after a new configuration is installed to remove old configurations. Also, the algorithm has at least time $e + \tau$ after the system stabilizes, that is, after α', to remove old configurations.

Configuration-viability is a reasonable assumption: whenever viability is threatened, a reconfiguration occurs before too many members of a configuration fail. We claim in Theorem 5.3 that $(\alpha', e, 23d)$*-configuration-viability* is sufficient to ensure that read and write operations complete.

Other Assumptions. We need a few other reasonable assumptions on executions: (1) $(\alpha', 8d)$*-recon-spacing*: after α', when the system stabilizes, reconfigurations are not too frequent; in particular, at least time $8d$ elapses between the $\mathsf{report}(c)_i$ event and any following $\mathsf{recon}(c,*)_i$ event. Also, every $\mathsf{recon}(c,*)_i$ event is preceded in α by a number of $\mathsf{report}(c)_*$ events, one for each node of some write-quorum of c. (2) (α', e)*-join-connectivity*: the network is connected so that nodes learn about each other rapidly; if two nodes both perform join-acks at or before time $t-e$, and $t \geq \ell time(\alpha') + e$, then by time t they know about each other. (3) *(α',e+d)-recon-readiness*: if some node i is a member of configuration c and a $\mathsf{recon}(*,c)$ event occurs, then i performs a join-ack at least time $e+d$ prior to the recon event. These last two properties ensure that all the nodes in installed, non-removed configurations are aware of each other.

Configuration Management Latency Results. First we examine how long a configuration upgrade operation takes to complete. This allows us to bound how long an obsolete configuration can remain in the system. For all the claims in this section, assume that α is an α'*-normal* execution, satisfying $(\alpha', e, 23d)$*-configuration-viability*, $(\alpha', 12d)$*-recon-spacing*, *(α',e)-join-connectivity*, and *(α',e+d)-recon-readiness*.

Lemma 5.1 *If the* config-upgrade$_i$ *action is enabled at time t, where $t > \ell time(\alpha')$, and if all necessary quorums are available, and if i does not fail, it completes the operation by time $t + 4d$.*

In general, a configuration upgrade operation takes time $\leq 4d$: two rounds of communication for each phase of the operation. Even though many configurations are being removed, the operation still takes at most $4d$ time to complete.

The next lemma bounds the time by which any configuration is removed. This result is not true of the original RAMBO algorithm: RAMBO might take arbitrarily long to remove an obsolete configuration, even after the network stabilizes and messages are reliably, rapidly delivered.

Lemma 5.2 *Assume that all configuration with index $\leq k$ are installed by time $t \geq \ell time(\alpha') + e + d$, and that node $i \in members(c(k-1))$ does not fail until after $t + 12d$. Then by time $t + 12d$, $cmap(h)_j$ is marked $\pm$ for all non-failed j that perform a* join-ack *by time $t - e$ and for all h such that $0 \leq h < k$.*

Lemma 5.2 follows from a tricky inductive argument. If all old configurations were previously removed, only recent configurations remain to be removed. Recent configurations, however, do not fail before they are removed, as guaranteed by configuration-viability. The base case depends on the guarantee that no configuration fails before $\ell time(\alpha') + e + 23d$. The main theorem then follows:

Theorem 5.3 *Let $t > \ell time(\alpha') + e + 9d$. Assume a* join-ack$_i$ *occurs prior to time $t - e - 9d$, and that i does not fail in α until after time $t + 8d$. Then if a read or write operation starts at node i at time t at node i, it completes by time $t + 8d$.*

Proof (sketch). Consider some configuration $c(k)$ such that $cmap(k)_i \in C$ at time t. Lemma 5.2 guarantees that at time t, all "old" configurations have been removed from $cmap_i$; only "recent" configurations remain. Therefore configuration $c(k)$ is a "recent" configuration, meaning that configuration $c(k+1)$ is installed no earlier than time $t - 15d$, and configuration-viability guarantees that configuration $c(k)$ does not fail before time $t + 8d$.

Since every non-removed configuration remains viable, and at most one reconfiguration occurs during each phase of the operation, the result follows by an argument similar to that used for Lemma 5.1; the read or write operation completes by time $t + 8d$. ☐

The conclusion is that if the execution satisfies configuration-viability, join-connectivity, recon-readiness, and recon-spacing, then the algorithm continues to make progress, despite prior bad network behavior or bursty reconfiguration. The strongest assumption is that of $(\alpha', e, 23d)$*-configuration-viability*, which requires configurations to survive time $23d$ after a new configuration is installed. We have bounded the length of this viability by a *constant* time, depending only on the network message delay bound, d. In comparison, the original RAMBO algorithm had no bound on the required configuration-viability.

6. Implementation and Preliminary Evaluation

Musial and Shvartsman [19] developed a prototype distributed implementation that incorporates both the original

RAMBO configuration management algorithm [15] and the new algorithm presented in this paper. The system was developed by manually translating the Input/Output Automata specification to Java code. To mitigate the introduction of errors during translation, the implementers followed a set of precise rules, similar to [4], that guided the derivation of Java code from Input/Output Automata notation. The system is undergoing refinement and tuning, however an initial evaluation of the performance of the two algorithms has been performed in a local-area setting.

The platform consists of a Beowulf cluster with 13 machines running Linux (Red Hat 7.1). The machines are Pentium processors in the range from 90 MHz to 900 MHz, interconnected via a 100 Mbps Ethernet switch. The implementation of the two algorithms shares most of the code and all low-level routines. Any difference in performance is traceable to the distinct configuration management discipline used by each algorithm. Given several very slow machines, we do not evaluate absolute performance and instead focus initially on comparing the two algorithms.

The preliminary results in Figure $3(a)$ show the average latency of read/write operations as the frequency of reconfigurations grows from about two to twenty reconfigurations per one gossip period. In order to handle such frequent reconfigurations, we chose a large gossip interval (8 seconds). This interval is much larger than the round-trip message delay, thus allowing us to reduce the effects of network congestion encountered when reconfiguring very frequently. The results show that the overall latency of read/write operations for the new algorithm is progressively improved. As expected, the decrease in latency becomes substantial for bursty reconfigurations (at 20 reconfigurations per gossip interval). For less frequent reconfigurations the latency is similar, at about 4 gossip intervals depending on the settings (not shown). This is expected and consistent with our analysis, since the two algorithms are essentially identical when $cmap$s contain one or two configurations. Figure $3(b)$ shows the average number of configurations in $cmap$s as a function of reconfiguration frequency. This further explains the difference in performance, since the average number of configurations in $cmap$s is lower in the new algorithm as the frequency of reconfigurations increases.

Finally we observe that the modest number of machines used in this study favored the original algorithm. This is because the machines are often members of multiple configurations, thus the number of messages needed to reach fixed-points by the read/write operations of the original algorithm is much lower than is expected when each processor is a member of a few configurations. Full performance evaluation is currently in progress. We are investigating how the performance depends on the number of machines and various timing parameters.

7. Conclusions and Future Work

We have presented RAMBO II, a new algorithm improving on the original RAMBO by Lynch and Shvartsman. While RAMBO performs well in the context of benign network behavior and infrequent reconfiguration, RAMBO II recovers rapidly after periods of arbitrary asynchrony and message loss. Additionally, RAMBO II performs better during frequent reconfiguration. As a result, RAMBO II is more reliable in long running, dynamic systems, and has shorter read and write operation latency during instability.

One important open problem is to design an algorithm that chooses new configurations and when to initiate reconfiguration. The improved configuration management protocol in RAMBO II makes it more practical to consider the design of such algorithms. In the earlier RAMBO algorithm, the requirements of a configuration-choosing algorithm are unclear. This paper shows that the configuration chooser must provide exactly $(\alpha', e, 23d)$*-configuration-viability*.

Based on the analysis of Karger and Liben-Nowell[13], we might assume that the long running, dynamic, system has a bounded half-life, defined as the time in which either half the processes fail or the number of active processes doubles. It is then possible to design a configuration choosing algorithm that initiates a reconfiguration a fixed number of times in every half-life. By choosing appropriate quorums and appropriate numbers of reconfigurations, $(\alpha', e, 23d)$*-configuration-viability* should be possible.

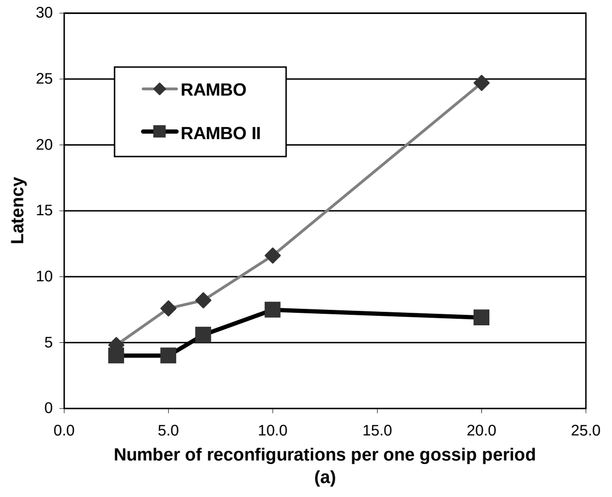
There are a number of other open problems. How can the algorithm recover if viability is compromised and too many quorums fail? Is it possible for read operations to return more rapidly? Can the join protocol be improved, to allow more rapid integration into the system? In general, how can RAMBO II be adapted for peer-to-peer or mobile settings? Does a similar strategy support stronger data objects, rather than just read or write data? What are the lower bounds on how fast a reconfigurable system can recover when the network stabilizes?

In conclusion, we have presented a new algorithm for atomic memory in highly dynamic environments. We showed that it is correct, and that operations terminate rapidly under certain well-defined conditions. This, then, creates the framework for introducing and solving many interesting open problems.

Acknowledgments. We thank Peter Musial for valuable feedback that improved the quality of the formal presentation, and for the preliminary experimental results comparing the new algorithm to the RAMBO algorithm.

References

[1] *Communications of the ACM*, 39(4), 1996. (Special section on group communication).



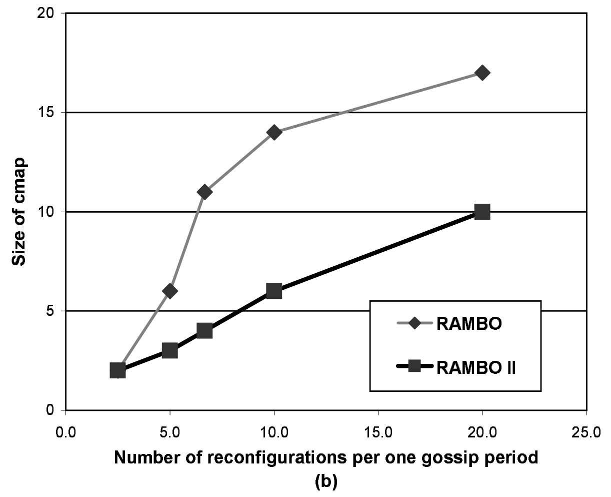


Figure 3. Preliminary empirical evaluation of: (a) the average operation latency measured as the number of gossip intervals, and (b) the average number of configurations in $cmap$'s, as functions of reconfiguration frequency, measured as number of reconfigurations per one reconfiguration period.

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

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

[4] O. Cheiner and A. Shvartsman. Implementing and evaluating an eventually-serializable data service as a distributed system building block. In *Networks in Distributed Computing*, volume 45 of *DIMACS Series on Disc. Mathematics and Theoretical Computer Science*, pages 43–71. AMS, 1999.

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

[6] El Abbadi, Skeen, and Cristian. An efficient fault-tolerant protocol for replicated data management. In *Proceedings of the 4th Annual ACM Symposium on Principles of Databases*, pages 215–228, 1985.

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

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

[9] S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO II: Implementing atomic memory in dynamic networks, using an aggresive reconfiguration strategy. Technical Report LCS-TR-890, Massachusetts Institute Technology, 2003.

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

[11] 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.

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

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

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

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

[16] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. Technical Report LCS-TR-856, M.I.T., 2002.

[17] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

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

[19] P. Musial and A. Shvartsman. Implementing RAMBO. In Progress.

[20] R. D. Prisco, A. Fekete, N. A. Lynch, and A. A. Shvartsman. A dynamic primary configuration group communication service. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 64–78, September 1999.

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

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

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