Developing a Consistent Domain-Oriented Distributed Object Service*

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

Abstract

This paper presents a new algorithm for a reconfigurable distributed domain-oriented *atomic object service, called DO-RAMBO, which stands for Domain-Oriented Reconfigurable Atomic Memory for Basic Objects. This service is suitable for inclusion as a middleware system service for distributed applications requiring atomic read/write data. The implementation substantially extends and refines the abstract RAMBO algorithm of Lynch and Shvartsman that supports individual atomic objects. In this paper* domains *are introduced to allow the users to group related atomic objects. The new implementation manages configurations on the basis of domains, significantly improving the utility and the performance of the resulting service. DO-RAMBO guarantees consistency under asynchrony, message loss, node crashes, new node arrivals, and node departures. We present the formal algorithm development for DO-RAMBO and give analytical and preliminary empirical results that illustrate the benefit of the new approach.*

1 Introduction

This paper presents a formal development of a practical distributed service supporting shared read/write atomic objects in dynamic network settings. Users of the service can efficiently group objects in users' scope of interest in user-defined domains. This service is suitable for maintaining consistent long-lived survivable data in a dynamic networks, in which participants may join, leave, or fail during the course of computation. Such settings are becoming increasingly common in modern distributed applications that rely on multitudes of communicating, computing devices. 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.

An approach to implementing read/write objects for dynamic networks was developed by Lynch and Shvartsman [13], and extended by Gilbert *et al.* [10] and Georgiou *et al.* [7]. They implement a consistent distributed memory service called RAMBO (Reconfigurable Atomic Memory for Basic Objects) that maintains atomic (linearizable) data in dynamic environments. In order to achieve availability in the presence of failures, the objects are replicated at several network locations. To maintain consistency in the presence of small and transient changes, the algorithm uses *configurations* consisting of *quorums* of locations. To accommodate larger and more permanent changes, the algorithm supports *reconfiguration*, by which the configurations are modified. Any configuration may be installed at any time. Obsolete configurations can be removed from the system without interfering with the ongoing read and write operations [10]. The algorithm tolerates asynchrony, node arrivals, departures and failures, and message loss.

Motivation for the current development. The original RAMBO algorithms [13, 10, 7] are specified using the Input/Output Automata (IOA) formalism [14, 11]. This enables one to reason formally about the properties of the service. The service is parameterized by object name, that is, the service is specified individually for each object instance. Multiple objects are supported by composing multiple instances of the service, one for each object. While composition preserves atomicity, the resulting service is impractical for supporting large numbers of objects because this requires running multiple instances of the service, one instance per object, which introduces substantial processing and messaging overhead. For example, bookkeeping communication is carried out in the background individually for each object, and configuration upgrades and removal of obsolete configurations must also be done on a per-object basis. With this approach, the penalty for the mathematical

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

[†]Dept. of Computer Science, University of Cyprus, 75 Kallipoleos Str., P.O. Box 20537, CY-1678, Nicosia, Cyprus. Email: chryssis@ucy.ac.cy

[‡]Dept. of Computer Science and Engineering, University of Connecticut, 371 Fairfield Rd., U-2155, Storrs, CT 06269, USA. Email: piotr@cse.uconn.edu

[§]CSAIL, Massachusetts Institute of Technology, The Stata Center, Cambridge, MA 02139, USA. Email: alex@theory.csail.mit.edu

simplicity of the formal specification is the reduced practicality of the resulting system.

In many settings applications may use multiple related objects, e.g., the objects may represent data values of interest to certain users. In such cases it is highly desirable to eliminate redundancy by allowing a collection of objects to share configurations and related processing. In this work we investigate an approach where multiple related objects are grouped into a *domain*, so that reconfiguration is performed on the per-domain basis instead of on the per-object basis. While this is a conceptually sensible approach, formally specifying such a solution and proving it correct is fairly involved. To assess the practicality of the solution, it is also important to experiment with a working system that implements the desired service in a network.

Contributions. We present a new algorithm implementing reconfigurable, domain-oriented, atomic distributed object service, called Domain-Oriented Reconfigurable Atomic Memory for Basic Objects, or DO-RAMBO. The algorithm borrows from the abstract RAMBO algorithms [13, 10, 7] that implement individual reconfigurable objects. We introduce the notion of *domains* that allow the users to group related objects. Users join the system by means of join requests. The objects in domains are then accessed by means of read and write operations. Users request reconfiguration by means of recon operations. The algorithm manages configurations on the basis of domains, which significantly improves the practicality of the service.

We use Input/Output Automata [14] (IOA) to specify the algorithms and reason about correctness. The formal development is presented in two steps. First, based on the ideas in [13, 10], we present and prove correct an intermediate version. Here we implement domains, yet the resulting algorithm is not practical for long-lived applications because it involves messages that may grow in size without bound. In the second step, we make the algorithm practical using the approach in [7] for implementing long-lived data services. Presenting the development in two steps substantially simplifies the proofs of correctness, and separates the functional issues of domain implementation from the performance issues associated with dynamic network settings and the need to bound the demands for communication bandwidth. We perform conditional latency analysis that shows that, under reasonable network behavior assumptions, the read and write operations take at most time 8δ, where δ is the maximum message delay (unknown to the algorithm). We developed a complete implementation of the DO-RAMBO service on a network of workstations. This development is an example of an approach to software engineering in which formal algorithm design is followed by a methodical translation of the abstract algorithm specification in IOA to distributed Java code using the techniques we earlier documented in [15].

Background. Several approaches have been used to implement consistent data in (static) distributed systems. Starting with the work of Gifford [9] and Thomas [16], many algorithms have used collections of intersecting sets of objects replicas (such as quorums) to solve the consistency problem. Upfal and Wigderson [17] use majority sets of readers and writers to emulate shared memory. Vitányi and Awerbuch [4] use matrices of registers where the rows and the columns are written and respectively read by specific processors. Attiya, Bar-Noy and Dolev [3] use majorities of processors to implement shared objects in static message passing systems. Extension for limited reconfiguration of quorum systems have also been explored [6, 12]. Virtually synchronous services [5], and group communication services (GCS) in general [1], can also be used to implement consistent data services, e.g., by implementing a global totally ordered broadcast. While the universe of processors in a GCS can evolve, in most implementations, forming a new view takes a substantial time, and client operations are interrupted during view formation. In our algorithm, as in [13, 10], reads and writes can make progress during reconfiguration.

Document structure. In Section 2 we present the specification and the algorithms for the reconfigurable domain-oriented object service. Proof of atomicity is in Section 3. In Section 4 we refine the algorithm making it suitable for long-lived executions. Conditional performance analysis and the experimental results are presented in Section 5. We conclude the paper in Section 6. Due to lack of space, several low-level details and proofs are omitted (they can be found in [8]).

2 The DO-RAMBO Algorithm

In this section we present the architecture of DO-RAMBO, then focus on the key new components. We begin by presenting RAMBO, since at the component level the structure of DO-RAMBO follows that of RAMBO. In later sections we describe in detail the new components of DO-RAMBO. As presented in [13, 10], RAMBO is given for a single object. Since atomicity is preserved under composition, single-object services can be composed to yield a complete shared memory. However, as discussed earlier, doing this introduces performance overheads making the resulting service impractical for supporting large numbers of objects.

In order to achieve fault tolerance and availability, RAMBO uses reconfigurable quorum configurations, where any quorum configuration may be installed, and atomicity is preserved in all executions. Old configurations are removed and the new configurations are updated with the latest object information, without any interference from ongoing reads and writes. Multiple configurations may be removed con-

currently.

Read and write operations consist of two phases. In the first phase, the node initiating a read or write operation contacts at least one read-quorum of each installed configuration. The quorum intersection property ensures that after this phase the most up to date information about the object is obtained. In the next phase this information (in case of a write, the new value) is propagated to appropriate write-quorums, ensuring consistency. Next, we present each of these operations in detail for the DO-RAMBO algorithm.

2.1 DO-RAMBO: architecture and interface

The overall architecture is given in Figure 1, following the earlier model of RAMBO. The main external distinction is that DO-RAMBO automata are parameterized by a domain name, instead of an object name. The algorithm is composed of several components, formally expressed as Input/Output Automata [14].

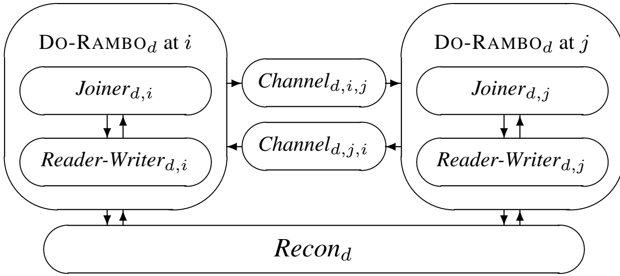

Figure 1. DO-RAMBO$_d$ component architecture depicting the automata at some representative nodes i and j, the channels, and the *Recon* service.

Let D be the set of domain identifiers. For $d \in D$ we define X_d to be the set of object identifiers for domain d. Let I be the set of network locations. For each domain d and each participating network location i, the system includes $Joiner_{d,i}$ automata, which handle joining of new participants, and $Reader\text{-}Writer_{d,i}$ automata, which handle reading, writing, and upgrading configurations. The *Reader-Writer* and *Joiner* automata have access to asynchronous channels $Channel_{d,i,j}$ providing communication from location i to location j, implemented as a typical unidirectional asynchronous channel that does not corrupt messages, but that may reorder and lose messages. The *Reader-Writer* automata interact with an arbitrary implementation of the *Recon* service that is responsible for emitting a totally-ordered sequence of configurations based on user requests (this service is exactly as specified in [13] and we do not discuss it further). The *Joiner* automata implement a very simple protocol that allows new participants to join the system. The details can be found in [13]. The only difference is that in DO-RAMBO nodes join the service for a domain of objects, and not for a single object.

Data types:
I, a set of processes, D, a set of domains, V, a set of legal values
X_d, a set of object identifiers from domain d, where $d \in D$
C, a set of configurations, each consisting of members, read/write-quorums
Input:
$\mathsf{join}(\mathsf{rambo}, J)_{d,i},\ J$ a finite subset of $I - \{i\}$, $i \in I$,
such that if $i = i_0$ then $J = \emptyset$, $d \in D$
$\mathsf{read}(x)_{d,i},\ i \in I,\ x \in X_d,\ d \in D$
$\mathsf{write}(x, v)_{d,i},\ v \in V,\ i \in I,\ x \in X_d,\ d \in D$
$\mathsf{recon}(c, c')_{d,i},\ c, c' \in C,\ i \in members(c),\ i \in I,\ d \in D$
$\mathsf{fail}_{d,i},\ i \in I,\ d \in D$
Output:
$\mathsf{join\text{-}ack}(\mathsf{rambo})_{d,i},\ i \in I,\ d \in D$
$\mathsf{read\text{-}ack}(x, v)_{d,i},\ v \in V,\ i \in I,\ x \in X_d,\ d \in D$
$\mathsf{write\text{-}ack}(x)_{d,i},\ i \in I,\ x \in X_d,\ d \in D$
$\mathsf{recon\text{-}ack}(b)_{d,i},\ b \in \{ok, nok\}, i \in I,\ d \in D$
$\mathsf{report}(c)_{d,i},\ c \in C,\ i \in I,\ d \in D$

Figure 2. DO-RAMBO$_d$: External signature.

The heart of the DO-RAMBO system is the *Reader-Writer* automata that implement read and write operations, perform upgrade to new and remove obsolete configurations. We present this in Section 2.2. The external interface of the service is given in Figure 2. Processes join the system via join/join-ack events. Read (write) operations correspond to read/read-ack (write/write-ack) events. Participants submit reconfiguration requests using the recon action, which is acknowledged via the recon-ack event. Participants learn about new configurations via the report event. We model node crashes using an external fail event. In the presentation that follows we will deal with a single domain (only to reduce notational clutter) and suppress explicit mention of d where it is clear from the context.

2.2 Reader-Writer automata

We now present in detail the $Reader\text{-}Writer_i$ automata: signature, state, transitions, and operation protocols.

Signature and state. The signature and state of $Reader\text{-}Writer_i$ appear in Figure 3. The state variables are used as follows. The $status$ variable keeps track of the progress of the component as it joins the protocol. When $status = \mathsf{idle}$, $Reader\text{-}Writer_i$ does not respond to any inputs (except for join) and does not perform any locally controlled actions. When $status = \mathsf{joining}$, $Reader\text{-}Writer_i$ is receptive to inputs but still does not perform any locally controlled actions. When $status = \mathsf{active}$, the automaton participates fully in the protocol.

The $world$ variable is used to keep track of all processes that are known to have attempted to join the system. The $value$ array contains the latest known value for the local replica of each object, e.g., $value(x)$ is a value for the local replica of some object x. The tag array holds the associated tag of each object, e.g., $tag(x)$ is the latest known tag for the object x (tags are pairs consisting of a sequence number and location id, comparable lexicographically). The $cmap(\cdot)$ variable contains information about configurations: If $cmap(k) = \bot$, it means that $Reader\text{-}Writer_i$ has not yet learned what the kth configuration iden-

Signature:

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

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

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

State:

- $status \in \{\mathsf{idle}, \mathsf{joining}, \mathsf{active}\}$, initially idle
- $world$, a finite subset of I, initially $\emptyset$
- $value(x) \in V_x, x \in X_d$, initially $\forall\, x \in X_d$: $value(x) = (v_0)_x$
- $tag \in X \rightarrow T$, initially $\forall\, x \in X_d$: $tag(x) = (0, i_0)$
- $cmap \in CMap$, initially $cmap(0) = c_0$,
  $cmap(k) = \bot$ for $k \geq 1$
- $pnum1 \in X_d \rightarrow \mathbb{N}$, initially $\forall\, x \in X_d$: $pnum1(x) = 0$
- $pnum2 \in X_d \times I \rightarrow \mathbb{N}$, initially $\forall\, x \in X_d, \forall j \in I$,
  where $j \neq i$: $pnum2(x,j) = 0$
- $failed$, a Boolean, initially $false$

- $op(x)$, an array of records (one for each object $x \in X_d$) with fields:
  - $type \in \{\mathsf{read}, \mathsf{write}\}$
  - $phase \in \{\mathsf{idle}, \mathsf{query}, \mathsf{prop}, \mathsf{done}\}$, initially idle
  - $pnum \in \mathbb{N}$
  - $cmp \in CMap$
  - acc, a finite subset of I
  - $val \in V_x$
- upg, a record with fields:
  - $phase \in \{\mathsf{idle}, \mathsf{query}, \mathsf{prop}\}$, initially idle
  - $pnum(x) \in \mathbb{N}$, $\forall\, x \in X_d$: $pnum(x) = 0$
  - $cmap \in CMap$
  - $acc(x)$, a finite subset of I, $\forall\, x \in X_d$
  - $target \in \mathbb{N}$

Figure 3. $Reader\text{-}Writer_i$: Signature and state

tifier is. If $cmap(k) = c \in C$, it means that *Reader-Writer*$_i$ has learned that the kth configuration identifier is c, and the kth configuration was not included in a local configuration upgrade operation. If $cmap(k) = \pm$, it means that *Reader-Writer*$_i$ performed a configuration upgrade operation that included k^{th} configuration identifier. *Reader-Writer*$_i$ learns about configuration identifiers either directly, from the *Recon* service, or indirectly, from other *Reader-Writer* processes. The value of $cmap$ is always in $Usable$, that is, $\pm$ for some finite prefix of $\mathbb{N}$, followed by an element of C, followed by elements of $C \cup \{\bot\}$, with only finitely many elements of C. When *Reader-Writer*$_i$ processes a read or write operation, it uses all the configurations whose identifiers appear in its $cmap$ up to the first $\bot$.

The $pnum1$ array and $pnum2$ matrix are used to implement a handshake that identifies "recent" messages in regards to a specific object. *Reader-Writer*$_i$ uses $pnum1$ array to count the total number of operation "phases" it has initiated overall per object, including phases occurring in read, write, and configuration upgrade operations. (A "phase" here refers to either a query or propagate phase, as described below.) For every j, including $j = i$ and some object x, *Reader-Writer*$_i$ uses $pnum2(x,j)$ to record the largest number of a phase that i has learned that j has started.

For each object x, the record $op(x)$ contains information about the locally-initiated read or write operation in progress. The record upg contain information about the locally-initiated configuration upgrade in progress. A node can process read/write operations concurrently with configuration upgrades. The *type* subfield records the type of the operation in progress, either a read or a write. The $cmap$ subfield records the configuration map associated with the operation on x. 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. The $pnum$ subfield records the phase number when the phase begins, allowing the initiator to determine which responses correspond to the phase. The phase of the operation is indicated by $phase$ subfield. The acc subfield records which nodes have responded during the current phase. The like named subfields of upg record are defined analogously. The $upg.target$ subfield records the identifier of configuration that is the target of current upgrade operation.

Transitions. Transitions pertaining to reading, writing, and configuration upgrade are presented in Figure 4.

Joining. Since the join protocol is simple, we omit the related to it transitions (see [13]), and instead we briefly describe the join process. When $status = \mathsf{idle}$ and $\mathsf{join}(\mathsf{rw})_i$ input occurs, then: if $i = i_0$ and is the domain's initiator then $status$ becomes active and *Reader-Writer*$_i$ is now ready for conducting operations; otherwise, $status$ becomes joining, making *Reader-Writer*$_i$ receptive to inputs only. In both cases, *Reader-Writer*$_i$ records itself as a member of its own $world$. From this point on, *Reader-Writer*$_i$ also adds to its $world$ any process from which it receives a *join* message (these messages are originated by the *Joiner* automata).

After *Reader-Writer*$_i$ receives a $\mathsf{recv}(*)_{*,i}$ message (see Figure 4) from another process while $status = \mathsf{joining}$, then $status$ becomes active. At this point, process i can perform a join-ack(rw) and has acquired enough information to begin participating fully.

Information propagation. Information is propagated between *Reader-Writer* processes in the background, using send and recv actions. Each message sent by process i is per object (we describe in Section 5 how to remove this requirement) and includes: an object identifier obj, the latest known $value(obj)$ and $tag(obj)$, $world$, $cmap$, and two phase numbers—the current phase number of i,

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

Input recv$(\langle W, cm, obj, v, t, pns, pnr\rangle)_{j,i}$
Effect:
if $\neg failed$ and $status \neq$ idle then
  $status \leftarrow$ active
  $world \leftarrow world \cup W$
  $cmap \leftarrow update(cmap, cm)$
  if $t > tag(obj)$ then
    $(value(obj), tag(obj)) \leftarrow (v, t)$
  $pnum2(obj, j) \leftarrow \max(pnum2(obj, j), pns)$
  if $op(obj).phase \in \{$query, prop$\}$ and
   $pnr \geq op(obj).pnum$ then
    $op(obj).cmp \leftarrow$
      $extend(op(obj).cmp, truncate(cm))$
    if $op(obj).cmp \in Truncated$ then
      $op(obj).acc \leftarrow op(obj).acc \cup \{j\}$
    else
      $pnum1(obj) \leftarrow pnum1(obj) + 1$
      $op(obj).acc \leftarrow \emptyset$
      $op(obj).cmp \leftarrow truncate(cmap)$
  if $upg.phase \in \{$query, prop$\}$ and
   $pnr \geq upg.pnum(obj)$ then
    $upg.acc(obj) \leftarrow upg.acc(obj) \cup \{j\}$

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

Input read$(x)_i$
Effect:
if $\neg failed$ and $status \neq$ idle then
  $pnum1(x) \leftarrow pnum1(x) + 1$
  $op(x) \leftarrow \langle$read, query, $pnum1(x),$
    $truncate(cmap), \emptyset, op.(x).value\rangle$

Input write$(x, v)_i$
Effect:
if $\neg failed$ and $status \neq$ idle then
  $pnum1(x) \leftarrow pnum1(x) + 1$
  $op(x) \leftarrow \langle$write, query, $pnum1(x),$
    $truncate(cmap), \emptyset, v\rangle$

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

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

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

Output write-ack$(x)_i$
Precondition:
$\neg failed$
$status =$ active
$op(x).type =$ write
$op(x).phase =$ done
Effect:
$op(x).phase =$ idle

Internal cfg-upgrade$(k)_i$
Precondition:
$\neg failed$
$status =$ active
$upg.phase =$ idle
$cmap(k) \in C$
$\forall l \in \mathbb{N}, l < k : cmap(l) \neq \bot$
Effect:
for all $x \in X$ do
  $pnum1(x) \leftarrow pnum1(x) + 1$
  $upg.pnum(x) \leftarrow pnum1(x)$
  $upg.acc(x) \leftarrow \emptyset$
$upg.phase \leftarrow$ query
$upg.target \leftarrow k$
$upg.cmap \leftarrow cmap$

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

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

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

Figure 4. $Reader\text{-}Writer_i$: Read/write and configuration upgrade transitions

$pnum1(obj)$, and the latest known phase number of the receiver, $pnum2(obj, j)$. These background messages may be sent at any time, once the process is active. They are sent only to processes in the sender's $world$ set.

When *Reader-Writer*$_i$ receives a message, $status$ is set to active. The incoming world information, in W, is merged with the local $world$ set. Also, the local $cmap$ is updated with the incoming configuration information, cm. That is, for each k, if $cmap(k) = \bot$ and $cm(k)$ is a configuration identifier $c \in C$, then process i sets its $cmap(k)$ to c. Also, if $cmap(k) \in C \cup \{\bot\}$, and $cm(k) = \pm$ then *Reader-Writer*$_i$ sets its $cmap(k)$ to $\pm$, indicating that this configuration has been removed. Since, messages are per object, obj indicates identifier of this object, and is used to update the remaining state variables. *Reader-Writer*$_i$ compares the incoming tag t to its own $tag(obj)$. If t is strictly greater, it represents a more recent version of this object; in this case, $tag(obj)$ is replaced with t and $value(obj)$ with value v. *Reader-Writer*$_i$ also updates its $pnum2(obj, j)$ component for the sender j to reflect new information about the phase number of the sender for the object whose identifiers is obj, which appears in the pns component of the message.

The last sequence of updates depends on the following: if *Reader-Writer*$_i$ is conducting a phase of a read, write, or configuration upgrade, and the incoming message is "recent", then sender j is replying to a message that i sent in the current phase. Phase numbers are used to perform this check: if the incoming phase number pnr is at least as large

as the current operation phase number ($op(obj).pnum$ or $upg.pnum(obj)$), then the message is recent. If these conditions are met then $op(obj)$ and upg records are updated.

Read and write operations. A read or write operation on object x consists of a query phase and a propagation phase. In each phase, *Reader-Writer*$_i$ obtains recent tag and $value$ for x, and $cmap$ information from "enough" processes (as we explain below) by exchanging messages in the background, as described above.

For an object x, when *Reader-Writer*$_i$ starts either a query or a propagation phase of a read or write, it sets $op(x).cmp$ to $truncate(cmap)$, which is defined to include all the configuration identifiers in the local $cmap$ up to the first $\perp$.When a new $CMap$, cm, is received during the phase, $op(x).cmp$ is "extended" by adding all newly-discovered configuration identifiers, up to the first $\perp$ in cm. If adding these new configuration identifiers does not create a "gap", that is, if the extended $op(x).cmp$ is in $Truncated$, then the phase continues using the new $op(x).cmp$. On the other hand, if a "gap" is created (that is, the result is not in $Truncated$), then *Reader-Writer*$_i$ can infer that it has been using out-of-date configuration identifiers. In this case, it restarts the phase using the best currently known $CMap$, information, which is obtained by computing $truncate(cmap)$ for the latest local $cmap$.

In between restarts, while process i is engaged in a single attempt to complete a phase, it never removes configuration identifiers from $op(x).cmp$. In particular, if process i learns during a phase that a configuration identifier in $op(x).cmp(k)$ has been included in some configuration upgrade, it does not remove it from $op(x).cmp$, but continues to include it in conducting the phase.

The query phase of a read or write operation terminates when a *query fixed point* is reached. This happens when *Reader-Writer*$_i$ receives recent responses from some read-quorum of each configuration in $op(x).cmp$. Let t denote process i's $tag(x)$ at the query fixed point. Then we know that t is at least as great as the $tag(x)$ value that each process in each of these read-quorums had at the start of this phase.

If the operation is a read, then process i at this point fixes its current value as the value to be returned to its client. However, before returning this value, process i performs the propagation phase, whose purpose is to make sure that "enough" *Reader-Writer* processes have acquired tags that are at least t (and associated values). Again, the information is propagated in the background, and $op(x).cmp$ is managed as described above. The propagation phase ends once a *propagation fixed point* is reached, when *Reader-Writer*$_i$ has received recent responses from some write-quorum of each configuration in the current $op(x).cmp$. When this occurs, we know that the $tag(x)$ of each process in each of these write-quorums is at least t.

Processing for a write operation, for object x, starting with a $\mathsf{write}(x, v)_i$ event is similar to that for a read. The query phase is conducted exactly as for a read, but processing after the query fixed point is different. Suppose t, process i's $tag(x)$ at the query fixed point, is of the form (n, j). Then *Reader-Writer*$_i$ defines the tag for its write operation to be the pair $(n+1, i)$. *Reader-Writer*$_i$ sets its local $tag(x)$ to $(n+1, i)$ and its $value(x)$ to v, the value it is currently writing. Then, it performs its propagation phase. The purpose of the propagation phase is to ensure that "enough" processes acquire tags that are at least as great as the new tag $(n+1, i)$. The propagation phase is conducted and concluded exactly as for a read operation.

New configurations and configuration upgrade. 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 query phase terminates, i.e. the cfg-upg-query-fix point is reached, when for each object in the domain fresh responses from at least one read quorum and at least one write quorum of each old configuration are collected. In the second phase, the latest object information obtained in the query phase is propagated to the members of the write-quorum of the new configuration. This means that the upg-cfg-prop-fix event occurs when fresh responses for each object in the domain from members of the write-quorum of the new configuration are collected. This ensures that the latest domain information is propagated to the new configuration.

Note, in DO-RAMBO the upgrade operation is conducted on behalf of all objects in the domain, hence the query and propagation phases are based on fresh responses for each object from appropriate quorums.

The complete algorithm. The complete implementation is the composition of the $Joiner_i$ and *Reader-Writer*$_i$ automata for all i, all the channels, and any automaton whose traces satisfy the *Recon* safety specification—with all the non-external actions of DO-RAMBO hidden. (Recall that the specification of *Recon* service [13] is essentially unchanged: the only difference is that the *Recon* service is parameterized by domains instead of objects.)

3 Atomic Consistency

We now state the key lemmas that lead to the main result. Throughout the rest of this paper, we consider "good"

executions of the algorithm. In general, the assumptions we present require well-formed requests: clients follow the protocols for joining and to initiating reconfiguration; clients initiate only one operation at a time on any object; clients wait for appropriate acknowledgments before proceeding.

Definitions. In this section, we assume that α is an arbitrary, good execution of the algorithm. We also assume that $\pi(x)_1$ and $\pi(x)_2$ are two read or write operations on some object x from domain d (i.e., $x \in X_d$) that occur at i_1 and i_2 respectively, where i_1 and i_2 are participants of DO-RAMBO$_d$ service. Additionally, we assume that $\pi(x)_1$ completes before $\pi(x)_2$ begins in α. In the case when the ordering of operations is not important we denote a read or write operation on x as $\pi(x)$. For every $\pi(x)$, the query-fix(x) (resp. prop-fix(x)) event occurs immediately after the query (resp. prop) phase of $\pi(x)$ completes. For every configuration upgrade operation γ, the cfg-upg-query-fix and cfg-upg-prop-fix events are defined analogously.

Next we introduce history variables. First, the $\mathit{query\text{-}cmap}(\pi(x))$ is a map from integer indices to $C \cup \{\bot, \pm\}$, initially undefined. It is set in the query-fix(x) step of $\pi(x)$, to the value of $op(x).cmp$ in the pre-state. (If configuration with index ℓ equals $\bot$, $c(\ell) = \bot$, then this means that this configuration has not been installed. On the other hand, if $c(\ell) = \pm$ then this configuration has been upgraded.) The history variable $\mathit{prop\text{-}cmap}(\pi(x))$ is defined analogously for the propagation phase of operation $\pi(x)$.

The query-phase-start($\pi(x)$), initially undefined is defined in the query-fix(x) step of $\pi(x)$, to be the unique earlier event at which the collection of query results was started and not subsequently restarted (the last time $op(x).acc$ set is assigned $\emptyset$). This is either a read(x), write($x, *$), or recv($*, *, x, *, *, *, *$) event. The event prop-phase-start($\pi(x)$) is defined analogously, but with respect to the propagation phase.

For every read or write operation $\pi(x)$ at node i, we define the history variable $tag(\pi(x))$ to be the value of $tag(x)_i$ when the query-fix(x) event occurs for $\pi(x)$ at node i. If $\pi(x)$ is a read operation then $tag(\pi(x))$ is the largest tag that node i encounters during the query phase. If $\pi(x)$ is a write operation, $tag(\pi(x))$ is the new tag that is chosen by i for performing the write.

Similarly, for a configuration upgrade operation γ at node i, we define $tag(x, \gamma)$ to be the tag of object x at node i (i.e., $tag(x)_i$) when the cfg-upg-query-fix event occurs, that is, the largest tag encountered for object x at node i during the query phase of γ.

The last history variable is $\mathit{removal\text{-}set}(\gamma)$, defined for the configuration upgrade operation γ. It is a subset of $\mathbb{N}$, initially undefined, and records the configuration identifiers of the configurations that are marked for removal (configurations with identifier less than $upg.target$ for γ).

Correctness. We show atomicity using the framework of Lemma 13.16 in [11]. Recall that α is an arbitrary, good execution of the algorithm. We need to show that in α if all invoked read/write operations complete, then these operations on x can be partially ordered by an ordering $\prec_x$, so that with regard to each object $x \in X_d$ the following properties are satisfied. *(P1)*: $\prec_x$ totally orders all write operations in α. *(P2)*: $\prec_x$ orders every read operation in α with respect to every write operation in α. *(P3)*: for each read operation, if there is no preceding write operation in $\prec_x$, then the initial value is returned; else, the read operation returns the value of the unique write operation immediately preceding it in $\prec_x$. *(P4)*: if some operation, $\pi(x)_1$, completes before another operation, $\pi(x)_2$, begins in α, then $\pi(x)_2$ does not precede $\pi(x)_1$ in $\prec_x$. If such ordering $\prec_x$ can be constructed for α, then the algorithm guarantees atomic consistency.

We define $\prec_x$ in terms of the lexicographic order on tags of operations $\pi(x)$. As *(P1)* to *(P3)* are essentially immediate, we focus on *(P4)*. To demonstrate that our algorithm implements atomic objects, we have to show that $tag(\pi(x)_1) \leq tag(\pi(x)_2)$, and the strict inequality if $\pi(x)_2$ is a write operation.

First we examine the behavior of sequential read and write operations. The first lemma describes propagation of tag information, in the case where the propagation phase of the first operation and the query phase of the second operation share an active configuration.

Lemma 3.1 *Let $\pi(x)_1$ and $\pi(x)_2$ be as defined above and $k \in \mathbb{N}$, such that the* prop-fix(x) *event of $\pi(x)_1$ precedes the* query-phase-start($\pi(x)_2$) *event in α. If $\mathit{prop\text{-}cmap}(\pi(x)_1) \cap \mathit{query\text{-}cmap}(\pi(x)_2) \neq \emptyset$, then $tag(\pi(x)_1) \leq tag(\pi(x)_2)$ and if $\pi(x)_2$ is a write then $tag(\pi(x)_1) < tag(\pi(x)_2)$.*

The next lemma says that when two read or write operations on x, $\pi(x)_1$ and $\pi(x)_2$, execute sequentially, the smallest configuration index used in the propagation phase of $\pi(x)_1$ is no larger than the largest index used in the query phase of $\pi(x)_2$.

Lemma 3.2 *Let $\pi(x)_1$ and $\pi(x)_2$ be as defined above, such that the* prop-fix(x) *event of $\pi(x)_1$ precedes the* query-phase-start($\pi(x)_2$) *event in α. Then:* $\min(\{\ell : \mathit{prop\text{-}cmap}(\pi(x)_1)(\ell) \in C\}) \leq \max(\{\ell : \mathit{query\text{-}cmap}(\pi(x)_2)(\ell) \in C\})$.

The only remaining case is when $\mathit{prop\text{-}cmap}(\pi(x)_1)$ and $\mathit{query\text{-}cmap}(\pi(x)_2)$ are disjoint and $\mathit{max}(\mathit{prop\text{-}cmap}(\pi(x)_1)) \leq \mathit{min}(\mathit{query\text{-}cmap}(\pi(x)_2))$. In the rest of the discussion this relationship between the *cmap*s is assumed. Next, we proceed to show the appropriate relationship between the *tag*s.

The next lemma shows that if, for some read/write operation $\pi(x)$, k is the index of the smallest active configuration

in $query\text{-}cmap(\pi(x))$, then some configuration upgrade operation γ with target k precedes $\pi(x)$ and updates $c(k)$.

Lemma 3.3 *Let $\pi(x)$ be as previously defined and that* query-fix(x) *event occurs in α. Let k be the smallest element such that $query\text{-}cmap(\pi(x))(k) \in C$. Assume $k > 0$. Then there must exist a configuration upgrade operation γ such that $upg.target$ of γ equals k, and the* cfg-upg-prop-fix *event of γ precedes the* query-phase-start$(\pi(x))$.

Lemma 3.3 implies that $tag(x, \gamma) \leq tag(\pi(x)_2)$. Tags are propagated from the configuration upgrade operation to the following read or write operation via update of $c(k)$. The first operation updates some write-quorum of $c(k)$ in its propagation phase and the later accesses some read-quorum of $c(k)$ in the query phase. By the intersection properties of the read and write quorums and the fact that γ completes before $\pi(x)_2$ begins, the claim follows. Similarly, it follows that if $\pi(x)_2$ is a write operation then $tag(x, \gamma) < tag(\pi(x)_2)$.

Now, we construct a sequence of preceding upgrade operations satisfying certain properties. Assuming that some configuration with index k is removed by the specified upgrade operation. For every configuration with an index smaller than k, we choose a single upgrade operation—that removes this configuration—to add to the sequence. Therefore the constructed sequence may well contain the same configuration upgrade operation multiple times, if the operation removed multiple configurations. If two elements in the sequence are distinct upgrade operations, then the earlier operation completes before the later operation is initiated. Also, the target of an upgrade operation in the sequence is removed by the next distinct upgrade operation in the sequence. As a result of these properties, the configuration upgrade process obeys a sequential discipline. The sequential nature of configuration upgrade has a nice consequence for propagation of tags: for any sequence of upgrade operations as described here, $tag(x, \gamma_s)$, where $x \in X_d$, is nondecreasing in s.

Lemma 3.4 *Let $\gamma_\ell, \ldots, \gamma_k$ be a sequence of configuration upgrade operations such that:*

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

Then $\forall s, x : 0 \leq s < s_2 - 1, x \in X_d$, $tag(x, \gamma_s) \leq tag(x, \gamma_{s+1})$.

Lemmas 3.1 to 3.4 are used to show the key theorem:

Theorem 3.5 *Let $\pi(x)_1$ and $\pi(x)_2$ be as previously defined and that the* prop-fix(x) *event of $\pi(x)_1$ precedes the* query-phase-start$(\pi(x)_2)$ *event in α. Then $tag(\pi(x)_1) \leq tag(\pi(x)_2)$, and if $\pi(x)_2$ is a write then $tag(\pi(x)_1) < tag(\pi(x)_2)$.*

Proof. (Sketch). The proof is similar to that of Theorem 4.6 of [10]. Let $cm_1 = prop\text{-}cmap(\pi(x)_1)$ and $cm_2 = prop\text{-}cmap(\pi(x)_2)$. If both cm_1 and cm_2 share a configuration, then the result follows from Lemma 3.1. Now assume that cm_1 and cm_2 are disjoint. Let ℓ_1 be the largest element in cm_1, and ℓ_2 the smallest element in cm_2. Per Lemma 3.2, $\ell_1 < \ell_2$. Lemma 3.3 with $\pi(x) = \pi(x)_2$ and $k = \ell_2$ defines an upgrade operation γ which precedes $\pi(x)_2$. Using Lemma 3.4 we construct a sequence of configuration upgrades $\gamma_0, \ldots, \gamma_{\ell_2 - 1}$ such that $\gamma_{\ell_2 - 1} = \gamma$. Consider γ_{ℓ_1} from this sequence. From Lemma 3.4 we have that $tag(x, \gamma_{\ell_1}) \leq tag(x, \gamma_{\ell_2 - 1})$. Now continuing with exactly the same reasoning as in Theorem 4.6 of [10] we get that $tag(\pi(x)_1) \leq tag(x, \gamma_{\ell_1})$. We already showed that $tag(x, \gamma_{\ell_1}) \leq tag(\pi(x)_2)$, and if $\pi(x)_2$ is a write operation then $tag(x, \gamma_{\ell_1}) < tag(\pi(x)_2)$. Combining all the above inequalities, the result follows. □

Theorem 3.5 shows that the tags of operations on x are monotonically increasing. It follows that the tags induce a partial order $\prec_x$ that meets the necessary and sufficient requirements for atomic consistency. Since the property holds for any x, it must hold for all $x \in X_d$. The main result follows:

Theorem 3.6 DO-RAMBO *implements atomic read/write objects.*

4 Long-Lived DO-RAMBO

In this section we make DO-RAMBO suitable for long lived executions, by allowing processes to gracefully leave the service, and by using an incremental gossip mechanism to reduce the size of gossip messages.

The long-lived version of RAMBO algorithm, called LL-RAMBO, is presented in [7]. LL-RAMBO supports graceful departures and incremental gossip, which we now briefly describe. Prior to departure, a process sends notification messages to some subset of processes in its $world$. Once these messages are sent the process simply stops participating in the service. A node that receives the departure notifications, marks the sender as departed, hence preventing any further communication with that node. The departed information is included in the gossip messages and shared among non-failed participants. This improvement reduces the extra communication burden created by processes sending gossip messages to processes that left the service. The incremental gossip protocol trades the local processing for decreased communication cost. Each process keeps track of information that is known by each process in its $world$. This knowledge is used when sending gossip messages: only new

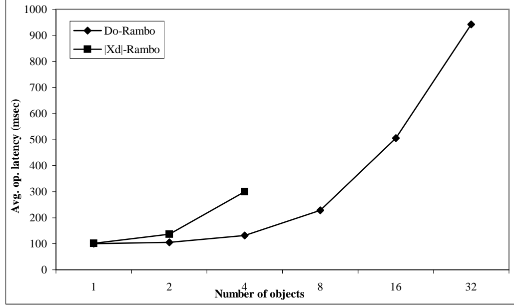

Figure 5. DO-RAMBO vs. the composition of $|X_d|$ instances of RAMBO.

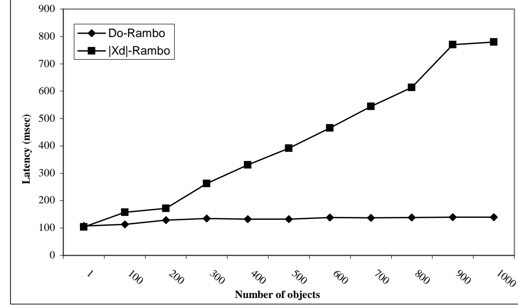

Figure 6. DO-RAMBO vs. RAMBO for a single "super-object" of $|X_d|$ objects.

or unacknowledged information is sent. The proof of correctness of LL-RAMBO is shown by forward simulation [7].

We now make two observations. First, in LL-RAMBO the leave protocol is per object, similarly to the join service. In the case of DO-RAMBO the leave protocol is per domain. Second, the incremental gossip mechanism is used on $world$ (and $departed$) variables; state variables that pertain to the entire domain. Therefore, as in [7] we augment DO-RAMBO by adding the leave protocol and incremental gossip mechanism and obtain long-lived DO-RAMBO.

Using the approach from [7] we prove the following by forward simulation.

Theorem 4.1 *Long-lived* DO-RAMBO *implements atomic read/write objects.*

In the remainder of the paper, when let DO-RAMBO denote the long-lived version of the service.

5 Analysis, Implementation, and Evaluation

We now present a conditional analysis of operation latency in DO-RAMBO, and the preliminary empirical results obtained from our implementations of RAMBO and DO-RAMBO on a LAN, comparing the performance of the two implementations in two different settings.

Conditional Analysis. A conditional analysis of RAMBO read and write operation latency is presented in [13, 10, 7]. Here we show that under the same conditions DO-RAMBO has the same operation latency as RAMBO.

An execution is said to be in *steady state* when the following conditions hold: (a) the local clocks of all automata progress at exactly the rate of real time, (b) all messages sent prior to and during the steady state are delivered withing bounded time of δ, (c) the sending pattern is restricted, where each automaton sends messages at the first possible time and at regular intervals of δ, as measured on the local clock, (d) the non-send locally controlled events occur instantaneously and just once, and (e) reconfiguration is infrequent and the installed configurations are not disabled due to failures and departures—also, at the time of a configuration being installed, all of its members have successfully joined the system and have learned about each other. We now state the latency bounds on the read/write operations under the steady state assumptions in DO-RAMBO.

Theorem 5.1 *Let α be a steady state execution of* DO-RAMBO. *Assume i is a process that successfully joined the system prior to time t and does not fail or depart in α until after time $t + 8\delta$. Then if a read or write operation starts at process i for object x at time t, it completes by time $t + 8\delta$.*

Implementations. We implemented RAMBO and DO-RAMBO on a network-of-workstations. We now describe our implementations along with the initial experimental results. These preliminary results support our expectation that grouping objects into domain leads to improved performance.

We manually translated the IOA 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 [15]. The platform consists of a cluster with ten machines running Linux. The machines are various Pentium processors up to 900 MHz interconnected via a 100Mbps Ethernet switch.

Each instance of RAMBO (resp. DO-RAMBO) uses a single socket to receive messages over TCP/IP, and maintains a list of open, outgoing connections to each process in its world. Both algorithms use identical communication routines. The implementation of *Joiner* and *Recon* services is also identical.

The *Reader-Writer* service is implemented as described in the previous sections. Management of *common* state variables to RAMBO and DO-RAMBO, such as $world$, $cmap$, is identical. However, we make one simple optimization in the implementation of DO-RAMBO. In the specification of DO-RAMBO we assume that each gossip message is per object (contains *value*, *tag*, and *object identifier* of a single object). In the implementation our messages may include information about multiple objects (at least one). This simple optimization trivially preserves correctness.

Experiments. We designed two experiments as follows: There are ten processes that do not leave the system and a single configuration is installed that includes all of these processes as members. The configuration does not change

over time and consists of majorities; here we consider any majority configuration with at least six processes. In the first experiment we compare the performance of DO-RAMBO with $|X_d|$ objects to that of a $|X_d|$ instances of RAMBO, where all processes perform concurrent read and write operations on all objects in the domain. Figure 5 presents average latency of read/write operations (over all objects and all machines) as the number of objects grows from 1 to 32. We note that collecting data for the composition of RAMBO instances when the number of objects is 8 or larger ($8\times$RAMBO) was not possible, as our hardware was not capable of executing more than eight instances of RAMBO at the same time. A possible explanation of this phenomenon is the rapidly growing communication burden between the individual RAMBO automata. The second experiment is designed to compare the performance of DO-RAMBO to a single RAMBO instance that encapsulates the entire domain into a single object that we refer as a "super-object", where we choose a single object and let all processes perform read/write operations on that object concurrently. Figure 6 presents the average latency of read/write operations (over all machines) as the number of objects in the domain increases from 1 to 1000.

6 Discussion

RAMBO [13] is an atomic memory service for highly dynamic networks. Several proposals were recently made to make this service more practical [10, 7, 15]. An implementation of RAMBO is presented in [15]. These successive improvements are aimed at improving the performance of RAMBO implementations, but support only a single object per system instance. To support multiple shared atomic objects one has to use a composition of multiple RAMBO instances, one per object. This approach is very inefficient. In this paper we presented a specification and an efficient implementation of an atomic memory service that supports multiple related objects by grouping them into domains. We proved that the algorithms implement atomic objects. We methodically derived a real implementation of the service for a network-of-workstations, and we presented a preliminary comparison of its performance to the performance of the similar implementation of the prior RAMBO service.

In designing practical distributed object services, we also aim to make them useful in a broad set of distributed applications that incorporate atomic objects (cf. [2]). Thus we consider our services suitable as middleware. Our approach to middleware differs from common practice: although middleware frameworks such as CORBA, DCE and Java/JINI support construction of distributed systems from components, their specification capability is limited to the formal definition of interfaces and informal descriptions of behavior. These are not enough to support careful reasoning about the behavior of systems that are built using such services. Moreover, current middleware provides only rudimentary support for fault-tolerance. In contrast, our services are precisely defined, with respect to both their interfaces and their behavior.

References

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

[2] J. Albrecht and S. Yasushi. RAMBO for dummies. Technical Report HPL-2005-39, HP Labs, 2005.

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

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

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

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

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

[8] C. Georgiou, P. Musial, and A. Shvartsman. Developing a consistend domain-oriented distributted object service. `http://www.cse.uconn.edu/~piotr/pubs/TRs/GMS_NCA05F.pdf`, 2005.

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

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

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

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

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

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

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

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

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