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
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 δ0, 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-
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.](image)

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 \( \text{Joiner}_{d,i} \) automata, which handle joining of new participants, and \( \text{Reader-Writer}_{d,i} \) automata, which handle reading, writing, and upgrading configurations. The Reader-Writer and Joiner automata have access to asynchronous channels \( \text{Channel}_{d,i,j} \) 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.

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-Writer_i automata: signature, state, transitions, and operation protocols.

**Signature and state.** The signature and state of Reader-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 = idle, Reader-Writer_i does not respond to any inputs (except for join) and does not perform any locally controlled actions. When status = joining, Reader-Writer_i is receptive to inputs but still does not perform any locally controlled actions. When status = 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(\( k \)) variable contains information about configurations: If cmap(\( k \)) = \( \perp \), it means that Reader-Writer_i has not yet learned what the \( k \)th configuration iden-
that included

configuration upgrade operation. If

that is, ±

C

of

many elements of

configuration identifier is.

Reader-Writer

k

eration that included

signature:

Input:

read(x), x ∈ X_d

write(x, v), x ∈ X_d, v ∈ V_e

new-config(c, k), c ∈ C, k ∈ N^+

recv(join), i, j ∈ I − {i}

recv(m_e), m_e ∈ M, j ∈ I

join(rw),

fail_i

State:

status ∈ {idle, joining, active}, initially idle

world, a finite subset of I, initially ∅

tag ∈ X → T, initially ∀ x ∈ X_d: tag(x) = (v_0)_x

cmap ∈ CMap, initially cmap((0)) = c_0,

cmap(k) = ⊥ for k ≥ 1

pnum1 ∈ X_d → N, initially ∀ x ∈ X_d: pnum1(x) = 0

pnum2 ∈ X_d → N, initially ∀ x ∈ X_d: ∀ j ∈ I,

where j ≠ i: pnum2(x, j) = 0

failed, a Boolean, initially false

Output:

join-ack(rw),

read-ack(x, v), x ∈ X_d, v ∈ V_e

write-ack(x), x ∈ X_d

send(m_e), j ∈ I

op(x), an array of records (one for each object x ∈ X_d) with fields:

type ∈ {read, write}

phase ∈ {idle, query, prop, done}, initially idle

pnum ∈ N

cmap ∈ CMap

prop, a record with fields:

phase ∈ {idle, query, prop}, initially idle

pnum(x) ∈ N, ∀ x ∈ X_d: pnum(x) = 0

cmap ∈ CMap

prop(x), a finite subset of I

world, cmap, config-upgrade

target ∈ N

Figure 3. Reader-Writer: Signature and state

tifier is. If cmap(k) = c ∈ C, it means that Reader-Writer

has learned that the k^{th} configuration identifier is c, and the k^{th}

configuration was not included in a local configuration upgrade operation. If cmap(k) = ±, it means that Reader-Writer

performed a configuration upgrade operation that included k^{th} configuration identifier. Reader-Writer

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, ± for some finite prefix of N, followed by an element of C, followed by elements of C ∪ {⊥}, with only finitely many elements of C. When Reader-Writer, processes a read or write operation, it uses all the configurations whose identifiers appear in its cmap up to the first ±.

The pnum1 and pnum2 matrix are used to implement a handshake that identifies “recent” messages in regards to a specific object. Reader-Writer, 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, 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 = idle and join(rw), input occurs, then: if i = i_0 and is the domain’s initiator then status becomes active and Reader-Writer, is now ready for conducting operations; otherwise, status becomes joining, making Reader-Writer, receptive to inputs only. In both cases, Reader-Writer, records itself as a member of its own world. From this point on, Reader-Writer, 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, receives a recv(*),_,i message (see Figure 4) from another process while status = 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,
\[ \text{Input output}(\mathcal{W}, \text{cm}, \text{obj}, v, t, \text{pns}, \text{prn}) \], i, j

Effect:
- \text{status} = \text{active}
- \text{Effect}:

\[ \text{Input rec}(\mathcal{W}, \text{cm}, \text{obj}, v, t, \text{pns}, \text{prn}) \], i, j

Effect:
- \text{if } \neg \text{failed} \text{ and } \text{status} \neq \text{idle} \text{ then }
  \text{state} = \text{active}
- \text{Effect}:

\[ \text{Internal query-fix}(x), i \]

Precondition:
- \text{~failed}
- \text{status} = \text{active}
- \text{Effect:}

\[ \text{Internal prop-fix}(x), i \]

Precondition:
- \text{~failed}
- \text{status} = \text{active}
- \text{Effect:}

\[ \text{Internal query-fix}(x), i \]

Precondition:
- \text{~failed}
- \text{status} = \text{active}
- \text{Effect:}

\[ \text{Internal prop-fix}(x), i \]

Precondition:
- \text{~failed}
- \text{status} = \text{active}
- \text{Effect:}

\[ \text{Internal ckg-upgrade}(k), i \]

Precondition:
- \text{~failed}
- \text{status} = \text{active}
- \text{Effect:}

\[ \text{Internal ckg-upgrade}(k), i \]

Precondition:
- \text{~failed}
- \text{status} = \text{active}
- \text{Effect:}

\[ \text{Reader-Writer}: \text{Read/write and configuration upgrade transitions} \]

\[ \text{pnnum1}(\text{obj}), \text{and the latest known phase number of the receiver, pnnum2(\text{obj}, \text{j}). These background messages may be sent at any time, once the process is active. They are sent only to processes in the sender’s world set.} \]

When \text{Reader-Writer}, receives a message, \text{status} is set to active. The incoming world information, in \W, is merged with the local \world set. Also, the local \cm is updated with the incoming configuration information, \cm. That is, for each \( k \), if \cm(k) = \perp and \cm(k) is a configuration identifier \( c \in C \), then process \( i \) sets its \cm(k) to \( c \). Also, if \cm(k) \( \in C \cup \{\perp\} \), and \cm(k) = \perp then \text{Reader-Writer}, sets its \cm(k) to \perp, 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. \text{Reader-Writer}, compares the incoming \tag(k) to its own \( \text{tag}(\text{obj}) \). If \( t \) is strictly greater, it represents a more recent version of this object; in this case, \( \text{tag}(\text{obj}) \) is replaced with \( t \) and \value(\text{obj}) with \( v \). \text{Reader-Writer}, also updates its \pnnum2(\text{obj}, \text{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 \text{pns} component of the message.

The last sequence of updates depends on the following:
- if \text{Reader-Writer}, 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 \( \text{prn} \) is at least as large
as the current operation phase number \((\text{op}(\text{obj}), \text{pn}\text{um} \text{ or } \text{upg.pnum}(\text{obj}))\), then the message is recent. If these conditions are met then \(\text{op}(\text{obj})\) and \(\text{upg}\) 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, obtains recent \(\text{tag}\) and \(\text{value}\) for \(x\), and \(\text{cmap}\) 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 \(\text{op}(\text{obj}).\text{cmp}\) to \(\text{truncate}(\text{cmap})\), which is defined to include all the configuration identifiers in the local \(\text{cmap}\) up to the first \(\bot\). When a new \(\text{CMap}\), \(\text{cn}\), is received during the phase, \(\text{op}(\text{obj}).\text{cmp}\) is “extended” by adding all newly-discovered configuration identifiers, up to the first \(\bot\) in \(\text{cn}\). If adding these new configuration identifiers does not create a “gap”, that is, if the extended \(\text{op}(\text{obj}).\text{cmp}\) is in \(\text{Truncated}\), then the phase continues using the new \(\text{op}(\text{obj}).\text{cmp}\). On the other hand, if a “gap” is created (that is, the result is not in \(\text{Truncated}\)), 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 \(\text{CMap}\) information, which is obtained by computing \(\text{truncate}(\text{cmap})\) for the latest local \(\text{cmap}\).

In between restarts, while process \(i\) is engaged in a single attempt to complete a phase, it never removes configuration identifiers from \(\text{op}(\text{obj}).\text{cmp}\). In particular, if process \(i\) learns during a phase that a configuration identifier in \(\text{op}(\text{obj}).\text{cmp}(k)\) has been included in some configuration upgrade, it does not remove it from \(\text{op}(\text{obj}).\text{cmp}\), 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 \(\text{op}(\text{obj}).\text{cmp}\). Let \(t\) denote process \(i\)’s \(\text{tag}(x)\) at the query fixed point. Then we know that \(t\) is at least as great as the \(\text{tag}(x)\) 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 \(\text{op}(\text{obj}).\text{cmp}\) 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 \(\text{op}(\text{obj}).\text{cmp}\). When this occurs, we know that the \(\text{tag}(x)\) of each process in each of these write-quorums is at least \(t\).

Processing for a write operation, for object \(x\), starting with a \(\text{write}(x, v)_i\) 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 \(\text{tag}(x)\) at the query fixed point, is of the form \((n, j)\). Then Reader-Writer\(_i\) defines the \(\text{tag}\) for its write operation to be the pair \((n + 1, i)\). Reader-Writer\(_i\) sets its local \(\text{tag}(x)\) to \((n + 1, i)\) and its \(\text{value}(x)\) 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 \(\text{tag}(n + 1, i)\). 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 \(\text{cfg-upg-query-fix}\) 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 \(\text{upg-cfg-prop-fix}\) 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 \(\text{Joiner}_i\) 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 initiatiing 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 $\alpha$ 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 $\alpha$. 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 $\gamma$, the cfg-upg-query-fix and cfg-upg-prop-fix events are defined analogically.

Next we introduce history variables. First, the query-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($x$).cmap in the pre-state. (If configuration with index $\ell$ 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 prop-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($x, \ast$), or recv($\ast, \ast, x, \ast, \ast, \ast, \ast$) 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 $\gamma$ 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 $\gamma$.

The last history variable is removal-set($\gamma$), defined for the configuration upgrade operation $\gamma$. 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 $\gamma$).

Correctness. We show atomicity using the framework of Lemma 13.16 in [11]. Recall that $\alpha$ is an arbitrary, good execution of the algorithm. We need to show that in $\alpha$ 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 $\alpha$. (P2): $\prec_x$ orders every read operation in $\alpha$ with respect to every write operation in $\alpha$. (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 $\alpha$, then $\pi(x)_2$ does not precede $\pi(x)_1$ in $\prec_x$. If such ordering $\prec_x$ can be constructed for $\alpha$, 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 $\alpha$. If prop-cmap($\pi(x)_1$) $\cap$ query-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 $\alpha$. Then: $\min(\{\ell : \text{prop-cmap}(\pi(x_1))(\ell) \in C\}) \leq \max(\{\ell : \text{query-cmap}(\pi(x_2))(\ell) \in C\})$.

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

The next lemma shows that if, for some read/write operation $\pi(x)$, $k$ is the index of the smallest active configuration
in \(query-cmap(\pi(x))\), then some configuration upgrade operation \(\gamma\) 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 \(\alpha\). Let \(k\) be the smallest element such that \(query-cmap(\pi(x))(k) \in C\). Assume \(k > 0\). Then there must exist a configuration upgrade operation \(\gamma\) such that \(upg\_target\ of \(\gamma\) equals \(k\), and the \(cfg\_upg\_prop-fix\ event\ of \(\gamma\) 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 \(\gamma\) 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 \(\gamma_{s}, \ldots, \gamma_{k}\) be a sequence of configuration upgrade operations such that:
1. \(\forall s : 0 \leq s \leq s_2, s \in removal-set(\gamma_s)\),
2. \(\forall s : 0 \leq s < s_2 - 1, \text{ if } \gamma_s \neq \gamma_{s+1}, \text{ then the } cfg\_upg\_prop\_fix\ event \ of \(\gamma_s\) and the \(cfg\_upgrade\ event\ of \(\gamma_{s+1}\) occur in \(\alpha\), and the \(cfg\_upg\_prop\_fix\ event\ of \(\gamma_s\) precedes the \(cfg\_upgrade\ event\ of \(\gamma_{s+1}\), and
3. \(\forall s : 0 \leq s < s_2 - 1, \text{ if } \gamma_s \neq \gamma_{s+1}, \text{ then target}(\gamma_s) \in removal-set(\gamma_{s+1})\).

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

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 \(\alpha\). 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 \(cm_{11} = prop\_cmap(\pi(x)1)\) and \(cm_{22} = prop\_cmap(\pi(x)2)\). If both \(cm_{11}\) and \(cm_{22}\) share a configuration, then the result follows from Lemma 3.1. Now assume that \(cm_{11}\) and \(cm_{22}\) are disjoint. Let \(\ell_1\) be the largest element in \(cm_{11}\), and \(\ell_2\) the smallest element in \(cm_{22}\). 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 \(\gamma\) 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 \(\gamma_{\ell_1}\) 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 \(\prec\) 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
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 with bounded time of $\delta$, (c) the sending pattern is restricted, where each automaton sends messages at the first possible time and at regular intervals of $\delta$, 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—all, 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 $\alpha$ 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 $\alpha$ 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 major 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×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