Developing a Consistent Domain-Oriented Distributed Object Service

Chryssis Georgiou, Peter M. Musial, 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 empirical results that illustrate the benefit of the new approach.

Index Terms

C.2.4 and H.3.4.b: Distributed systems, F.3.1: Specifying and verifying and reasoning about programs, G.4.a: Algorithm and design analysis, G.4.g: Reliability and robustness.

I. 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 the scope of interest into user-defined domains. This service is suitable for maintaining consistent long-lived survivable data in 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 [1], and extended by Gilbert *et al.* [2], [3] and Georgiou *et al.* [4]. Their atomic (linearizable) distributed memory service is called RAMBO (Reconfigurable Atomic

This work is supported in part by the NSF Grants 9988304, 9984778, 0121277, and 0311368. A preliminary version of this work appears in Proc. of the 4th IEEE International Symposium on Network Computing and Applications, 2005, pages 149–158.

C. Georgiou is with Dept. of Computer Science, University of Cyprus, 75 Kallipoleos Str., P.O. Box 20537, CY-1678, Nicosia, Cyprus. Email: chryssis@cs.ucy.ac.cy

P.M. Musial is with Naval Postgraduate School, Computer Science Department, 1411 Cunningham Rd., GE-314, Monterey, CA 93943, USA. Email: pmmusial@nps.edu

A.A. Shvartsman is with Department of Computer Science & Engineering, University of Connecticut, 371 Fairfield Rd., Unit 2155, Storrs CT 06269, USA. Email: aas@cse.uconn.edu

Memory for Basic Objects). 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 new configurations are installed and obsolete configurations are removed from the system concurrently with the ongoing read and write operations. The service tolerates asynchrony, node arrivals, departures and failures, and message loss.

A. Motivation for the Current Development

The original RAMBO algorithms [1]–[4] are specified using the Input/Output Automata formalism [5], [6], enabling 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. The resulting service is impractical for supporting large numbers of objects because this requires running multiple instances of the service, one instance per object, introducing substantial processing and messaging overhead. For example, bookkeeping communication is carried out in the background individually for each object, and reconfiguration 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.

B. 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 [1]–[4] that implement individual reconfigurable objects. We introduce the notion of *domains* that allow the service 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 to specify the algorithms and reason about correctness. Building on ideas from [1]–[3], we present and prove the correctness of our new algorithm. Note that the presented algorithm in not practical for long-lived applications because it involves messages that may grow in size without bound. A long-lived, practical version of the algorithm can be obtained by applying the exact technique we developed in [4]. We omit such details from this presentation, to focus on the domain-based approach which is the contribution of this work.

We perform conditional latency analysis that shows that, under reasonable network behavior assumptions, the read and write operations take at most time 8δ and configuration upgrade takes at most 4δ, 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 our techniques [7]. We compare the performance of the implementation of DO-RAMBO with the one of RAMBO on a network of workstations; the obtained experimental results illustrate the performance benefits of DO-RAMBO.

C. Related Work

Several approaches have been used to implement consistent data in (static) distributed systems. Starting with the work of Gifford [8] and Thomas [9], many algorithms have used collections of intersecting sets of replicas to solve the consistency problem. Upfal and Wigderson [10] use majority sets of readers and writers to emulate shared memory. Vitányi and Awerbuch [11] use matrices of registers where the rows and the columns are written and respectively read by specific processors. Attiya, Bar-Noy and Dolev [12] use majorities of processors to implement shared objects in static message passing systems. Extension for limited reconfiguration of quorum systems have also been explored [13], [14]. Virtually synchronous services [15], and group communication services (GCS) in general [16], 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 substantial time, and client operations are interrupted during view formation. In our algorithm, as in [1]–[3], reads and writes can make progress during reconfiguration. Finally, consensus algorithms can be used directly to implement an atomic data service by allowing participants to agree on a global total ordering of all operations [17]. In contrast, we use consensus to agree only on the sequence of configurations and not on the individual operations. Also, in our algorithm, the termination of consensus affects the termination of reconfiguration, but not of read and write operations.

D. Document Structure

In Section II we present the specification and the algorithms for our object service. Proof of atomicity is in Section III. Conditional analysis of performance is presented in Section IV. Experimental results are presented in Section V. Section VI contains the concluding remarks.

II. THE DO-RAMBO ALGORITHM

In this section we first overview the DO-RAMBO service and its goals, and then we present its architecture and components in detail. DO-RAMBO aims to provide a robust and practical atomic memory service in dynamic systems. The service maintains atomicity in the presence of arbitrary node crashes, with fault-tolerance implemented through replication. The service uses quorums to ensure consistency, where the members of quorum sets are the object replica owners. In order to achieve availability in dynamic systems, DO-RAMBO service uses reconfiguration that introduces new quorum systems and removes obsolete quorum systems. The *configurations* used by the service consist of a unique identifier, a set of node identifiers, a set of read-quorums, and a

set of write-quorums, where each quorum is a subset of node identifiers. Here every write-quorum intersects every other write-quorum and every read-quorum intersects every write-quorum.

A. Atomicity

We now state a definition of atomicity for a read/write memory service following [6]. For any execution, if all the read and the write operations complete, then the operations on object x can be partially ordered by an ordering $\prec$,so that the following properties are satisfied:

P1. No operation has infinitely many other operations ordered before it.

P2. The order $\prec$ is consistent with the external order of invocation and responses, that is, there do not exist operations π_1 and π_2 such that π_1 completes before π_2 starts, yet $\pi_2 \prec_x \pi_1$.

P3. All write operations on x are totally ordered and every read operation on x is ordered with respect to all the writes on x.

P4. Every read operation on object x ordered after any write on x returns the value of the last write on x preceding it in the partial order; any read operation on x ordered before all writes on x returns the initial value of x.

The original RAMBO specification [1]–[3] is given for a single object, where the complete shared memory is obtained through atomicity-preserving composition of individual objects. Doing so introduces performance overheads making the resulting service impractical for large numbers of objects. The goal of DO-RAMBO is to provide atomicity and reconfigurability for a complete shared memory in a practical implementation.

B. DO-RAMBO *in a Nutshell*

The DO-RAMBO service consists of two components, the *Joiner* component and the *Reader-Writer* component that implements the main features of the service. DO-RAMBO relies on an external *Recon* service to provide a consistent sequence of quorum configurations. We now briefly introduce each of these, with the more detailed presentation following later in this section.

Each participant of DO-RAMBO runs an instance of the *Joiner* and *Reader-Writer* component and participates in the *Recon* service. The architecture of DO-RAMBO is depicted in Figure 1. The participants in *Joiner* and *Reader-Writer* components, and *Recon* service communicate with each other via communication channels that may lose, delay, and reorder messages.

The *Joiner* component implements a simple protocol that allows new participants to join the service. The join protocol is as follows. If a node is the first to initiate the service, then it is considered to be a creator and the *Joiner* component is used to initiate the *Reader-Writer* component and the *Recon* service. Otherwise, a node provides a seed set of possible participants of the service (for the specific domain) and sends a join request. Receipt of a join request by an active service participant is followed by an acknowledgment. Once a join acknowledgment message is received the new node may participate in the service and to host object replicas.

The *Reader-Writer* component implements a read and write protocol and a configuration upgrade protocol that removes old configurations. Read and write operations consist of two phases. In the first phase, the node initiating the operation contacts at least one read-quorum of each usable configuration. The quorum intersection property ensures that 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 of known configurations, ensuring

consistency. Obsolete configurations are removed from the system by the configuration upgrade protocol that consists of two phases during which the latest replica information is transfered from the write quorums of the configurations being removed to the write quorums of the configuration being updated. Multiple configurations may be removed concurrently.

The reconfiguration process involves installation of new configurations, where the consistent sequence of the configurations is established by an external *Recon* service. Our service does not depend on any specific implementation of *Recon*, however, it is required that the sequence of configurations emitted by *Recon* be without gaps and be totally ordered. At any time, an active participant of the DO-RAMBO service can submit new configuration to be considered as a next to be installed. The *Recon* service decides which of the proposed configurations will be installed and notifies the participants about its decision. It is important to point out that progress of read and write operations is independent on any ongoing configuration installation.

We next define notation and needed data types and present the components in detail.

C. Data Types

We assume two distinguished elements, $\perp$ and $\pm$, which are not in any of the basic types. For any type A, we define new types $A_\perp = A \cup \{\perp\}$ and $A_\pm = A \cup \{\perp, \pm\}$. If A is a partially ordered set, we augment its ordering by assuming that $\perp < a < \pm$ for every $a \in A$. We assume the following specific data types, distinguished elements, and functions.

- I, the totally-ordered set of *locations* or *nodes*.
- D, the set of *domains*. For $d \in D$, $(i_0)_d$ denotes the unique node that can create domain d.
- X_d, the set of *object identifiers* of domain d.
- For each $x \in X_d$: V_x, the set of values that object x may take on. $(v_0)_x \in V_x$, the initial value of x.
- T_d, the set of *tags* of the domain d, defined as $\mathbb{N} \times I$.
- C_d, the set of *configuration identifiers* for domain d. We denote by $(c_0)_d \in C_d$, the *initial configuration identifier* for d. We assume only the trivial partial order on C_d, in which all elements are incomparable; in $C_{d_\pm}$, all elements of C_d are still incomparable.
- For each $c \in C_d$ we define:
  - $members(c)$, a finite subset of I.
  - $read\text{-}quorums(c)$, a set of finite subsets of $members(c)$.
  - $write\text{-}quorums(c)$, a set of finite subsets of $members(c)$.

  We assume the following constraints:
  - $members((c_0)_d) = \{(i_0)_d\}$. That is, the initial configuration for domain d has only a single member, who is the creator (initiator) of d.
  - For every c, every $R \in read\text{-}quorums(c)$, and every $W \in write\text{-}quorums(c)$, $R \cap W \neq \emptyset$.

We now define operations on C_d.

- $update$, a binary function on $C_{d_\pm}$, defined by $update(c, c') = \max(c, c')$ if c and c' are comparable (in the augmented partial ordering of $C_{d_\pm}$), $update(c, c') = c$ otherwise.
- $extend$, a binary function on $C_{d_\pm}$, defined by $extend(c, c') = c'$ if $c = \perp$ and $c' \in C_d$, and $extend(c, c') = c$ otherwise.
- $CMap$, the set of *configuration maps*, defined as mappings from $\mathbb{N}$ to $C_{d_\pm}$, $\mathbb{N} \to C_{d_\pm}$. We extend the $update$ and $extend$ operators element-wise to binary operations on $CMap$.
- $truncate$, a unary function on $CMap$, defined by $truncate(cm)(k) = \perp$ if there exists $\ell \leq k$ such that $cm(\ell) = \perp$, $truncate(cm)(k) = cm(k)$ otherwise. This truncates configuration map cm by removing all the configuration identifiers that follow a $\perp$.
- $Truncated$, the subset of $CMap$ such that $cm \in Truncated$ iff $truncate(cm) = cm$.

- $Usable$, the subset of $CMap$ such that $cm \in Usable$ iff the pattern occurring in cm consists of a prefix of finitely many $\pm$s, followed by an element of C_d, followed by an infinite sequence of elements of $C_{d_\perp}$ in which all but finitely many elements are $\perp$.

D. DO-RAMBO*: Architecture and Interface*

The architecture of DO-RAMBO is given in Figure 1, where the components are defined as Input/Output Automata [5], following the model of RAMBO. The main external distinction is that DO-RAMBO automata are parameterized by a domain name, instead of an object name.

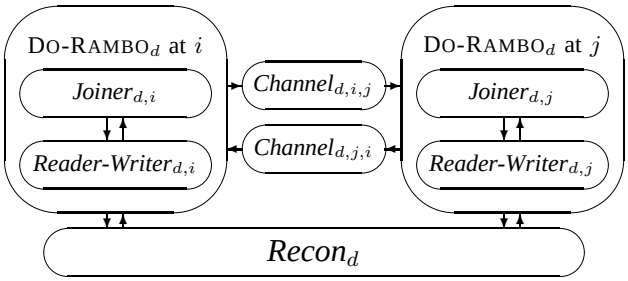

Fig. 1. DO-RAMBO$_d$ component architecture shown at some representative nodes i and j.

For each domain d and each participating node i, the system includes *Joiner*$_{d,i}$ automata that handle joining of new participants, and *Reader-Writer*$_{d,i}$ automata that handle reading, writing, and upgrading configurations. *Reader-Writer*$_d$ and *Joiner*$_d$ automata have access to channels *Channel*$_{d,i,j}$ providing communication from node i to node j, implemented as a typical unidirectional asynchronous channel that does not corrupt messages, but that may reorder and lose messages. *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 as specified in [1]). The *Joiner*$_d$ automata implement a very simple protocol that allows new participants to join the system. The only difference is that in DO-RAMBO nodes join the service for a domain of objects, and not for a single object.

The heart of the system is the *Reader-Writer* automata that implement read and write operations, perform upgrade to new and remove obsolete configurations. The external interface of the service is given in Figure 2. Nodes join the system via join/join-ack events. Read and write operations correspond to read/read-ack and write/write-ack events respectively. 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 sequel 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.

E. Joiner Automata

The service is "bootstrapped" using a protocol that allows nodes to join the service. The *Joiner*$_{d,i}$ component implements this protocol at node i for the domain d. Signature, state, and transitions of the component are specified in Figure 3. The state variables are as follows. The *status* variable keeps track of the component as it joins the DO-RAMBO$_d$ service. When $status =$ idle then the component does not perform any local actions. When $status =$ joining, *Joiner*$_i$ sends the join signal to the local *Recon*$_i$ and *Reader-Writer*$_i$ components and awaits

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:

$\mathsf{join\text{-}ack}(\mathsf{rambo})_{d,i}$, $i \in I$, $d \in D$
$\mathsf{read\text{-}ack}(x, v)_{d,i}$, $v \in V$, $i \in I$, $x \in X_d$, $d \in D$
$\mathsf{write\text{-}ack}(x)_{d,i}$, $i \in I$, $x \in X_d$, $d \in D$
$\mathsf{recon\text{-}ack}(b)_{d,i}$, $b \in \{ok, nok\}, i \in I$, $d \in D$
$\mathsf{report}(c)_{d,i}$, $c \in C$, $i \in I$, $d \in D$

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

acknowledgment, which when received allow $status$ to become active. The *child-status* is a mapping from $\{\mathsf{recon}, \mathsf{rw}\} \rightarrow \{\mathsf{idle}, \mathsf{joining}, \mathsf{active}\}$ and it keeps track of the local *Recon*$_i$ and *Reader-Writer*$_i$ components as they join the protocol. Prior to *Joiner*$_i$ initiating the join protocol with each component $child\text{-}status[*] = \mathsf{idle}$. Once *Joiner*$_i$ sends a join signal to *Recon*$_i$ or *Reader-Writer*$_i$ component, the corresponding $child\text{-}status$ variable becomes joining. When an acknowledgment is received, the corresponding $child\text{-}status$ variable becomes active. Variable *hints* is a placeholder for the set of node identifiers that *Joiner*$_i$ component is seeded with.

When *Joiner*$_i$ receives a $\mathsf{join}(\mathsf{rambo}, J)_i$ request from its environment, where J is a set of seed processor identifiers, it sends join messages to the processes in J with the hope that they are already participating in the service, and so can help in the attempt to join. Also, it submits join requests to the local *Reader-Writer*$_i$ and *Recon*$_i$ components and waits for acknowledgments. In the next section we describe *Reader-Writer* automata and how they handle join messages.

Signature:

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

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

State:

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

Transitions:

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

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

Input $\mathsf{fail}_{d,i}$
Effect:
  $failed \leftarrow \mathsf{true}$

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

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

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

Fig. 3. *Joiner*$_{d,i}$: Signature, state, and transitions

Data Types:

M, a set of messages, defined as $\langle W, cm, obj, v, t, pns, pnr \rangle$, where $W \subset I$, $cm \in CMap$, $obj \in X$, $v \in V_{obj}$, $t \in T$, and $pns, pnr \in \mathbb{N}$

Signature:

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

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

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

State:

- $status \in \{\mathsf{idle}, \mathsf{joining}, \mathsf{active}\}$, initially idle
- $world$, a finite subset of I, initially $\emptyset$
- $value(x) \in V_x$, $x \in X$, initially $\forall\ x \in X$: $value(x) = (v_0)_x$
- $tag \in X \to T$, initially $\forall\ x \in X$: $tag(x) = (0, i_0)$
- $cmap \in CMap$, initially $cmap(0) = c_0$, $cmap(k) = \perp$ for $k \geq 1$
- $pnum1 \in X \to \mathbb{N}$, initially $\forall\ x \in X_d$: $pnum1(x) = 0$
- $pnum2 \in X \times I \to \mathbb{N}$, initially $\forall\ x \in X, \forall j \in I$, where $j \neq i$: $pnum2(x, j) = 0$
- $failed$, a Boolean, initially $false$
- $op(x)$, an array of records (one for each object $x \in X$) 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
  - $pnum(x) \in \mathbb{N}$, $\forall\ x \in X$: $pnum(x) = 0$
  - $cmap \in CMap$
  - $acc(x)$, a finite subset of I, $\forall\ x \in X$
  - $target \in \mathbb{N}$

Fig. 4. *Reader-Writer*$_{d,i}$: Signature and state

F. Reader-Writer Automata

We now define the *Reader-Writer*$_i$ automata, their signature, state, and transitions.

1) Signature and state: The signature and state variables are given in Figure 4. Variable $status$ keeps track of the progress of the component as it joins the protocol. When $status = \mathsf{idle}$, *Reader-Writer*$_i$ does not respond to any inputs (except for join) and does not perform any locally controlled actions. When $status = \mathsf{joining}$, *Reader-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. Variable $world$ keeps track of all nodes that are known to have attempted to join the system. Array $value$ contains the latest known value of each object, i.e., $value(x)$ is the value of the local replica of x. Array tag holds the associated tag of each object, i.e., $tag(x)$ is the latest known tag of object x. Tags are pairs consisting of a sequence number and location id, comparable lexicographically. Variable $cmap(\cdot)$ contains information about configurations: If $cmap(k) = \perp$, it means that the k^{th} configuration is not yet known. If $cmap(k) = c \in C$, it means that *Reader-Writer*$_i$ has learned that the k^{th} configuration identifier is c. If $cmap(k) = \pm$, it means that some configuration upgrade operation removed the k^{th} configuration. *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 $C \cup \{\perp\}$, 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 $\perp$.

Array $pnum1$ and matrix $pnum2$ are used to 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, record $op(x)$ contains information about the latest locally-initiated read or write operation. Record upg contains information about the latest locally-initiated configuration upgrade. A node can perform read/write operations concurrently with configuration upgrades. Subfield $type$ records the type of operation, 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.

Reader-Writer transitions are given in Figures 5 and 6, and we next describe its operation.

2) Joining: When ***Reader-Writer***$_i$'s state variable is $status = \mathsf{idle}$ and $\mathsf{join(rw)}_i$ input occurs, then: if i is the domain's initiator, denoted by the value i_0, 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 5) from another process while $status = \mathsf{joining}$, then $status$ becomes active. At this point, process i can perform a $\mathsf{join\text{-}ack(rw)}$ and has acquired enough information to begin participating fully.

3) 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 V 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, $pnum1(obj)$, 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 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 $cmap(k) = \perp$ and $cm(k)$ is a configuration identifier $c \in C$, then process i sets its $cmap(k)$ to c. Also, if $cmap(k) \in C \cup \{\perp\}$, and $cm(k) = \pm$ then ***Reader-Writer***$_i$ sets its $cmap(k)$ to $\pm$, indicating that this configuration has been removed. The object identifier obj 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

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

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

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

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

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

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

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

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

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

Fig. 5. ***Reader-Writer***$_{d,i}$: Read/write transitions

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.

4) Read and write operations: Each read and write operation on object x consists of a query phase and a propagation phase. In each phase, ***Reader-Writer***$_i$ communicates with "enough" nodes (as we explain below) through information propagation in the background.

For an object x, when ***Reader-Writer***$_i$ starts a phase of a read or write, it sets $op(x).cmp$ to $truncate(cmap)$ that includes all configuration identifiers in $cmap$ up to the first $\perp$. When a

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

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
    $cmap(l) \leftarrow \pm$

Fig. 6. *Reader-Writer*$_{d,i}$: Configuration upgrade transitions

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 $op(x).cmp$. Else if a "gap" is present (i.e., the result is not in $Truncated$), then the configuration map is out-of-date. In this case, the phase is "restarted" using the best currently known $CMap$ information that is obtained by computing $truncate(cmap)$.

Other than restarts, node i never removes configuration identifiers from $op(x).cmp$ in processing a phase. In particular, if node 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 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 node 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 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

Input:	Output:
$\mathsf{join(recon)}_{d,i}$, $i \in I$, $d \in D$	$\mathsf{join\text{-}ack(recon)}_{d,i}$, $i \in I$, $d \in D$
$\mathsf{recon}(c, c')_{d,i}$, $c, c' \in C$, $i \in members(c)$, $d \in D$	$\mathsf{recon\text{-}ack}(b)_{d,i}$, $b \in \{\mathsf{ok}, \mathsf{nok}\}$, $i \in I$, $d \in D$
$\mathsf{fail}_{d,i}$, $i \in I$, $d \in D$	$\mathsf{report}(c)_{d,i}$, $c \in C$, $i \in I$, $d \in D$
	$\mathsf{new\text{-}config}(c, k)_{d,i}$, $c \in C$, $k \in \mathbb{N}^+$, $i \in I$, $d \in D$

Fig. 7. *Recon*$_{d,i}$: External signature

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.

5) New configurations and configuration upgrade: Configurations go through three stages: proposal, installation, and upgrade. The *install* stage requires interaction with the external *Recon* service. The external interface to *Recon* is depicted in Figure 7. Recall that *Recon* is responsible for emitting a consistent sequence of configurations chosen from the configurations submitted by the participants, but the exact implementation of this service is immaterial. The *Recon* service is activated via $\mathsf{join(recon)}$, where the corresponding $\mathsf{join\text{-}ack(recon)}$ event indicates readiness of the service. New configurations are submitted into *Recon* service (i.e., *proposed*) though the $\mathsf{recon}(c, c')$ event, where c' is the new configuration and c is the latest configuration known to the node emitting the proposal. Providing c as a parameter serves the following functions: (i) as a guard, where the submitting node must be a member of c, (ii) members of c will decide on the next configuration (where c' is included as one of the choices), and (iii) ensures total ordering of installations. When the configuration installation request completes, *Reader-Writer* is notified via $\mathsf{recon\text{-}ack}(b)$ event, where b is ok when installation of c' was successful and nok otherwise. Successfully installed configurations are reported to the *Reader-Writer* service via the report event. The *Recon* service is as specified in [1], except that the specification of *Recon* is parameterized by *domains* instead of *objects*. Since otherwise the implementation details of the *Recon* service are not essential to this presentation, we do not discuss it further.

The configuration is *upgraded* when every configuration with a smaller index has been removed. Once a configuration has been upgraded, it is responsible for maintaining the data. Upgrades are performed by the configuration upgrade operations (see Figure 6). The operation requires two phases, a query phase and a propagate phase. The query phase completes with event $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}$ when for each object in the domain fresh responses are collected from at least one read-quorum and at least one write-quorum of each old configuration. 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 event $\mathsf{upg\text{-}cfg\text{-}prop\text{-}fix}$ occurs when fresh responses for each object in the domain are collected from a write-quorum of the new configuration, ensuring that the latest domain information is propagated to the new configuration.

Note that 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.

G. The complete algorithm

The complete implementation $\mathcal{S}$ is the composition of the $Joiner_i$, ***Reader-Writer***$_i$ automata for all i, all the channels, and any automaton whose traces satisfy the *Recon* safety specification [1], with all the non-external actions of DO-RAMBO hidden.

H. Environment Well-Formedness

We assume that the clients of the service submit well-formed requests: clients follow the protocol to join and to initiate reconfigurations; clients initiate only one operation at a time on each object; clients wait for appropriate acknowledgments before proceeding.

First we state the well-formedness assumptions of DO-RAMBO$_d$, where $d \in D$, in terms of the following conditions.

For every $x \in X_d$, and $i \in I$:

- No $\mathsf{join}(\mathsf{rambo}, *)_{d,i}$, $\mathsf{read}(x)_{d,i}$, $\mathsf{write}(x, *)_{d,i}$ event is preceded by a $\mathsf{fail}_{d,i}$ event.
- At most one $\mathsf{join}(\mathsf{rambo}, *)_{d,i}$ event occurs.
- Any $\mathsf{read}(x)_{d,i}$, $\mathsf{write}(x, *)_{d,i}$, or $\mathsf{recon}(*, *)_{d,i}$ event is preceded by a $\mathsf{join\text{-}ack}(\mathsf{rambo})_{d,i}$ event.
- Any $\mathsf{read}(x)_{d,i}$, $\mathsf{write}(x, *)_{d,i}$, or $\mathsf{recon}(*, *)_{d,i}$ event is preceded by an -ack event for any preceding event of any of these kinds.
- For every c, at most one $\mathsf{recon}(*, c)_{d,*}$ event occurs.
- For every c, c', and i, if a $\mathsf{recon}(c, c')_{d,i}$ event occurs, then it is preceded by: (1) a $\mathsf{report}(c)_{d,i}$ event, and (2) a $\mathsf{join\text{-}ack}(\mathsf{rambo})_{d,j}$ event for every $j \in members(c')$.

The following are the well-formedness assumptions for ***Recon***$_d$. For every i:

- No $\mathsf{join}(\mathsf{recon})_{d,i}$ or $\mathsf{recon}(*, *)_{d,i}$ event is preceded by a $\mathsf{fail}_{d,i}$ event.
- At most one $\mathsf{join}(\mathsf{recon})_{d,i}$ event occurs.
- Any $\mathsf{recon}(*, *)_{d,i}$ event is preceded by a $\mathsf{join\text{-}ack}(\mathsf{recon})_{d,i}$ event.
- Any $\mathsf{recon}(*, *)_{d,i}$ event is preceded by an -ack for any preceding $\mathsf{recon}(*, *)_{d,i}$ event.
- For every c, at most one $\mathsf{recon}(*, c)_{d,*}$ event occurs.
- For every c and c' if a $\mathsf{recon}(c, c')_{d,i}$ event occurs, then it is preceded by: (1) a $\mathsf{report}(c)_{d,i}$ event, and (2) a $\mathsf{join\text{-}ack}(\mathsf{recon})_{d,j}$ event for every $j \in members(c')$.

In the rest of this paper we deal with *"good" executions* of implementations $\mathcal{S}$, viz. executions where the environment is well-formed, and where the communication channels behave correctly, delivering only the messages that were sent, but possibly reordering and losing some messages.

III. PROOF OF ATOMICITY

We now prove the correctness of DO-RAMBO: we prove that the service implements atomic read/write memory. In Section III-A, notation and basic invariants and lemmas are presented that are used in Section III-B to prove atomicity.

A. Notation and Basic Lemmas

We start by showing a simple result about the well-formedness of the DO-RAMBO service.

Theorem 1: In any good execution of DO-RAMBO the following guarantees are provided. For every $d \in D$ and i:

- Any join-ack(rambo)$_{d,i}$ (resp., recon-ack$(*)_{d,i}$) event has a preceding join(rambo, $*)_{d,i}$ (resp., recon$(*,*)_{d,i}$) event with no intervening invocation or response action for d and i.
- Any read-ack$(x,*)_{d,i}$ (resp., write-ack$(x)_{d,i}$) event has a preceding read$(x)_{d,i}$ (resp., write$(x,*)_{d,i}$) event with no intervening invocation or response action for d, x and i.

Proof. This simple result follows from code inspection under the assumptions about the client well-formedness as stated in Section II-H. □

In the rest of this section we restate the results from RAMBO [1]–[3], and we introduce certain history variables[1] to the global state of system $\mathcal{S}$. Some of the notation in the proofs has been modified to allow us to reason about the new algorithm. Several of the original lemmas in [1]–[3] are restated using new notation and their proofs are updated accordingly. The results that pertain to the individual object are essentially unchanged. To avoid unnecessary restatement in what follows, we omit any proofs that are essentially the same as [1]–[3]. We refer the reader to the cited papers for these proofs, and here we focus on presenting new lemmas and new or re-constructed proofs that also constitute the contribution of this work. Definitions and meaning of data types used in this section are found in Section II-C.

In our presentation we are dealing with executions of implementation $\mathcal{S}$ and read, write, and configuration upgrade *operations* occurring in the executions. (Recall that we elide the mention of domains, unless the identity of a domain is material.) Read and write operations are performed on objects in a domain, and are uniquely identified by their starting events, specifically, a read operation on x at node i is defined by its read$(x)_i$ event, and a write operation is similarly defined by its write$(x,v)_i$ event. We will use notation $\pi(x)$ to denote a read or a write operation on x. A configuration upgrade operation is performed for a domain, and it is defined by the corresponding cfg-upgrade$_i$ event.

We introduce the following history variables:

- *in-transit*, a set of messages, initially $\emptyset$. A message is added to the set when it is sent by any *Reader-Writer*$_i$ to any *Reader-Writer*$_j$. No messages is every removed from this set.
- $c(k) \in C$, for every $k \in \mathbb{N}$, initially undefined. This is set when the first new-config$(c,k)_i$ occurs, for some c and i. It is set to the c that appears as the first argument of this action.
- $tag(\pi(x)) \in T$, initially undefined. This is set to the value of $tag(x)$ at the process running $\pi(x)$, at the point right after $\pi(x)$'s query-fix(x) event occurs. If $\pi(x)$ is a read operation this is the highest tag that it encounters during the query phase. If $\pi(x)$ is a write operation, this is the new tag that is selected for performing the write.
- $query\text{-}cmap(\pi(x))$, a $CMap$, initially undefined. This is set in the query-fix(x) step of $\pi(x)$, to the value of $op(x).cmap$ in the pre-state.
- $R(\pi(x),k)$, for $k \in \mathbb{N}$, a subset of I, initially undefined. This is set in the query-fix(x) step of $\pi(x)$, for each k such that $query\text{-}cmap(\pi(x))(k) \in C$. It is set to an arbitrary $R \in read\text{-}quorums(c(k))$ such that $R \subseteq op(x).acc$ in the pre-state.
- $prop\text{-}cmap(\pi(x))$, a $CMap$, initially undefined.
- $W(\pi(x),k)$, for $k \in \mathbb{N}$, a subset of I, initially undefined. This is set in the prop-fix(x) step of $\pi(x)$, for each k such that $prop\text{-}cmap(\pi(x))(k) \in C$. It is set to an arbitrary $W \in write\text{-}quorums(c(k))$ such that $W \subseteq op(x).acc$ in the pre-state.

[1]History variables are used to aid reasoning about properties of the algorithm and are not used by the algorithm.

- $tag(x, \gamma) \in T$, initially undefined. This is set to the value of $tag(x)$ at the process running γ, at the point right after γ's cfg-upg-query-fix event occurs.
- $removal\text{-}set(\gamma)$, a subset of $\mathbb{N}$, initially undefined. This is set in the cfg-upgrade step of γ, to the set $\{\ell : \ell < k, cmap(\ell) \neq \pm\}$.
- $R(\gamma, \ell)$, for $\ell \in \mathbb{N}$, a subset of I, initially undefined. This is set in the cfg-upg-query-fix step of γ, for all $\ell \in removal\text{-}set(\gamma)$, to an arbitrary $R \in read\text{-}quorums(c(\ell))$ such that $R \subseteq upg(x).acc$ in the pre-state, for each $x \in X_d$.
- $W_1(\gamma, \ell)$, for $\ell \in \mathbb{N}$, a subset of I, initially undefined.This is set in the cfg-upg-query-fix step of γ, for all $\ell \in removal\text{-}set(\gamma)$, to an arbitrary $W \in write\text{-}quorums(c(\ell))$ such that $W \subseteq upg(x).acc$ in the pre-state, for each $x \in X_d$.
- $W_2(\gamma)$, a subset of I, initially undefined.This is set in the cfg-upg-prop-fix step of γ, to an arbitrary $W \in write\text{-}quorums(c(k))$ such that $W \subseteq upg(x).acc$ in the pre-state, for all $x \in X_d$.

In any good execution α, we define the following events (more precisely, we are giving additional name to some existing events):

- query-phase-start$(\pi(x))$, initially undefined. This 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. This is either a read(x), write$(x, *)$, or recv$(*, *, x, *, *, *, *)$ event.
- prop-phase-start$(\pi(x))$, initially undefined. This is defined in the prop-fix(x) step of $\pi(x)$, to be the unique earlier event at which the collection of propagation results was started and not subsequently restarted. This is either a query-fix(x) or recv$(*, *, x, *, *, *, *)$ event.

1) Configuration map invariants: Here we give invariants describing the kinds of configuration maps that may appear in various places in the state of $\mathcal{S}$.

We begin with a lemma saying that various operations yield or preserve the "usable" property.

Lemma 1 ([1]–[3]): The following hold:

1) If $cm, cm' \in Usable$ then $update(cm, cm') \in Usable$.
2) If $cm \in Usable$, $k \in \mathbb{N}$, $c \in C$, and cm' is identical to cm except that $cm'(k) = update(cm(k), c)$, then $cm' \in Usable$.
3) If $cm, cm' \in Usable$ then $extend(cm, cm') \in Usable$.
4) If $cm \in Usable$ then $truncate(cm) \in Usable$.

The following invariant describes some properties of $cmap_i$ that hold while ***Reader-Writer***$_i$ is conducting a configuration upgrade operation.

Invariant 1 ([1]–[3]): If $upg.phase_i \neq$ idle and $upg.target_i = k$, then:

1) $\forall\ \ell : \ell \leq k \implies cmap(\ell)_i \in C \cup \{\pm\}$.
2) If $k_1 = min\{\ell : \ell \leq k \wedge upg.cmap(\ell) \neq \pm\}$ then $k_1 = 0$ or $cmap(k_1 - 1)_i = \pm$.

Next we describe the patterns of C, $\perp$, and $\pm$ values that may occur in configuration maps in various places in the system state. We use the dot notation to indicate components of state, for example, $s.cmap_i$ indicates that value of $cmap_i$ in state s.

Invariant 2: Let cm be a *CMap* that appears as one of the following:

1) The cm component of some message in *in-transit*.
2) $cmap_i$ for any $i \in I$.

3) $op(x).cmap_i$ for some $i \in I$ and for some $x \in X_d$ for which $op(x).phase_i \neq$ idle.
4) $query\text{-}cmap(\pi(x))$ or $prop\text{-}cmap(\pi(x))$ for any operation $\pi(x)$ on any object x.
5) $upg.cmap_i$ for some $i \in I$ for which $upg.phase_i \neq$ idle.

Then $cm \in Usable$.

Proof. By induction on the length of a finite execution.

Base: Part 1 holds because $in\text{-}transit$ is empty initially. Part 2 holds because initially, for every i, $cmap(0)_i = c_0$ and $cmap(k)_i = \perp$; the resulting $Cmap$ is in ***Usable***. Parts 3 and 5 hold vacuously, because in the initial state, all $op(x).phase$ and $upg.phase$ values are idle. Part 4 also holds vacuously, because initially, all $query\text{-}cmap$ and $prop\text{-}cmap$ variables are undefined.

Inductive step: Let s and s' be the states before and after the new event, respectively. We consider Parts 1-5 one by one.

For Part 1, the interesting case is a $\mathsf{send}_{i,*}$ event that puts a message containing cm in ***in-transit***. The precondition on the send action implies that cm is set to $s.cmap_i$. The inductive hypothesis, Part 2, implies that $s.cmap_i \in Usable$, which suffices.

For Part 2, fix i. The interesting cases are those that may chage $cmap_i$, namely, $\mathsf{new\text{-}config}_i$, $\mathsf{recv}_{*,i}$ for a gossip (non-join) message, and $\mathsf{cfg\text{-}upg\text{-}prop\text{-}fix}_i$.

1) $\mathsf{new\text{-}config}(c, *)_i$. This part of the proof is as in [2]; we refer the reader there for details.
2) $\mathsf{recv}(\langle *, cm, *, *, *, *, * \rangle)_{*,i}$. This part of the proof is also as in [2].
3) $\mathsf{cfg\text{-}upg\text{-}prop\text{-}fix}(k)_i$. This part of the proof is also as in [2].

For Part 3, we consider actions that modify $op(x).cmap_i$, namely, read_i, write_i, recv_i, and $\mathsf{query\text{-}fix}_i$, for some object $x \in X_d$.

1) $\mathsf{read}(x)_i$, $\mathsf{write}(x, *)_i$, and $\mathsf{query\text{-}fix}(x)_i$: By inductive hypothesis, $s.cmap_i \in Usable$. The new step sets $s'.op(x).cmap_i$ to $truncate(s.cmap_i)$; since $s.cmap_i \in Usable$, Lemma 1, Part 4, implies that this is also usable.
2) $\mathsf{recv}(\langle *, cm, x, *, *, *, * \rangle)_{*,i}$: This step may alter $op(x).cmap_i$ only if $s.op(x).phase_i \in \{\mathsf{query}, \mathsf{prop}\}$, and then in only two ways: by setting it either to $extend(op(x).cmap_i, truncate(cm))$ or to $truncate(update(s.cmap_i, cm))$. The inductive hypothesis implies that ***truncate***, ***extend***, and ***update*** all preserve usability. Therefore, $s'.op(x).cmap_i \in Usable$.

For Part 4, we consider $\mathsf{query\text{-}fix}(x)_i$ and $\mathsf{prop\text{-}fix}(x)_i$, of some read or write operation $\pi(x)$.

1) $\mathsf{query\text{-}fix}(x)_i$. This sets $s'.query\text{-}cmap(\pi(x))_i$ to the value of $s.op(x).cmap_i$. Since by inductive hypothesis the latter is usable, so is $s'.query\text{-}cmap(\pi(x))_i$.
2) $\mathsf{prop\text{-}fix}(x)_i$. This sets $s'.prop\text{-}cmap(\pi(x))_i$ to the value of $s.op(x).cmap_i$. Since by inductive hypothesis the latter is usable, so is $s'.prop\text{-}cmap(\pi(x))_i$.

For Part 5, the actions to consider are $\mathsf{cfg\text{-}upgrade}(k)_i$ and $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_i$. These set $s'.upg.cmap_i$ to the value of $s.cmap_i$. Since by the inductive hypothesis the later is usable so is $s'.upg.cmap_i$. □

We now strengthen Invariant 2 to say more about the form of the ***CMap***s that are used for read and write operations:

Invariant 3: Let cm be a $CMap$ that appears as $op(x).cmap_i$ for some $i \in I$ for which $op(x).phase \neq$ idle, or as $query\text{-}cmap(\pi(x))$ or $prop\text{-}cmap(\pi(x))$ for any operation $\pi(x)$ on object $x \in X_d$. Then:

1) $cm \in Truncated$.
2) cm consists of finitely many $\pm$ entries followed by finitely many entries from C followed by a infinite number of $\perp$ entries.

Proof. We prove that the desired properties hold for a cm that is $op(x).cmap_i$. The same properties for $query\text{-}cmap(\pi(x))$ and $prop\text{-}cmap(\pi(x))$ follows by the way they are defined, from $op(x).cmap_i$.

To prove Part 1 we proceed by induction. In the initial sate, $op(x).phase_i = \mathsf{idle}$, which makes the claim vacuously true. For the inductive step we consider all actions that alter $op(x).cmap_i$:

1) $\mathsf{read}(x)_i$, $\mathsf{write}(x,*)_i$, or $\mathsf{query\text{-}fix}(x)_i$: These set $op(x).cmap_i$ to $truncate(cmap_i)$, which is necessarily in *Truncated*.
2) $\mathsf{recv}(\langle *, cm, x, *, *, *, * \rangle)_{*,i}$: This first sets $op(x).cmap_i$ to a preliminary value and then tests if the result is in *Truncated*. If it is, we are done. If not, then this step resets $op(x).cmap_i$ to $truncate(cmap_i)$, which is in *Truncated*.

To see Part 2, note that $cm \in Usable$ by Invariant 2. The fact that $cm \in Truncated$ then follows from the definition of *Usable* and Part 1. □

2) Phase guarantees: Lemmas presented here discuss the effects of query and propagation phases of read/write and configuration upgrade operations. In more detail, we describe the information flow that must occur during these phases to allow operation completion.

Note that the case $j = i$ is treated uniformly with the case where $j \neq i$ because *Reader-Writer* algorithm treats communication from a location to itself exactly the way as communication between two different locations. We first consider the query phase of a configuration-upgrade.

Lemma 2: Suppose that a $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_i$ event for configuration upgrade operation γ occurs in α and $k' \in removal\text{-}set(\gamma)$. Suppose $j \in R(\gamma, k') \cup W_1(\gamma, k')$.
Then for each $x \in X_d$ there exist messages m_x from i to j and m'_x from j to i such that:

1) *obj* component of messages m_x and m'_x is equal to x.
2) m_x is sent after the $\mathsf{cfg\text{-}upgrade}(k)_i$ event of γ.
3) m'_x is sent after j receives m_x.
4) m'_x is received before the $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_i$ event of γ.
5) In any state after j receives m_x, $cmap(\ell)_j \neq \perp$ for all $\ell \leq k$.
6) $tag(x, \gamma) \geq t$, where t is the value of $tag(x)_j$ in any state before j sends message m'_x.

Proof. The phase number discipline, applied to object x, implies the existence of the claimed messages m_x and m'_x. For Part 5, the precondition of $\mathsf{cfg\text{-}upgrade}(k)$ implies that, when the $\mathsf{cfg\text{-}upgrade}(k)_i$ event of γ occurs, $cmap(\ell)_i \neq \perp$ for all $\ell \leq k$. Therefore, j sets $cmap(\ell)_j \neq \perp$ for all $\ell \leq k$ when it receives m_x. Monotonicity of $cmap_j$ ensures that this property persists.

For Part 6, let t be the value of $tag(x)_j$ in any state before j sends message m'_x. Let t' be the value of $tag(x)_j$ in the state just before j sends m'_x. Then $t \leq t'$, by monotonicity. The *tag* component of m'_x is equal to t', by the code for send. Since i receives this message before the $\mathsf{cfg\text{-}upg\text{-}query\text{-}fix}(k)_i$, it follows that $tag(x, \gamma)$ is set by i to a value $\geq t$. □

Next, we consider the propagation phase of a configuration upgrade.

Lemma 3: Suppose that a $\mathsf{cfg\text{-}upg\text{-}prop\text{-}fix}(k)_i$ event for a configuration upgrade operation γ occurs in α. Suppose that $j \in W_2(\gamma)$.
Then for each $x \in X_d$ there exist messages m_x from i to j and m'_x from j to i such that:

1) *obj* component of messages m_x and m'_x is equal to x.
2) m_x is sent after the cfg-upg-query-fix$(k)_i$ event of γ.
3) m'_x is sent after j receives m.
4) m'_x is received before the cfg-upg-prop-fix$(k)_i$ event of γ.
5) In any state after j receives m, $tag(x)_j \geq tag(x, \gamma)$, for all $x \in X_d$.

Proof. The phase number discipline, on individual object, implies the existence of the claimed messages m_x and m'_x. For Part 5, when j receives m_x, it sets $tag(x)_j$ to be $\geq tag(x, \gamma)$. Monotonicity of $tag(x)_j$ ensures that this property persists in later states. □

Next, we consider the query phase of read/write operations.

Lemma 4: Suppose that in α a query-fix$(x)_i$ event occurs for a read or write operation $\pi(x)$ on object x. Let $k, k' \in \mathbb{N}$. Suppose $query\text{-}cmap(\pi(x))(k) \in C$ and $j \in R(\pi(x), k)$.
Then there exist messages m_x from i to j and m'_x from j to i such that:
1) obj field of messages m and m' equals x.
2) m_x is sent after the query-phase-start$(\pi(x))$ event.
3) m'_x is sent after j receives m.
4) m'_x is received before the query-fix(x) event of $\pi(x)$.
5) If t is the value of the $tag(x)_j$ in any state before j sends m'_x, then:
   (a) $tag(\pi(x)) \geq t$, and (b) if $\pi(x)$ is a write operation then $tag(\pi(x)) > t$.
6) If $cmap(\ell)_j \neq \perp$ for all $\ell \leq k'$ in any state before j send m'_x, then $query\text{-}cmap(\pi(x))(\ell) \in C$ for some $\ell \geq k'$.

Proof. The phase number discipline, on individual object, implies the existence of the claimed messages m and m'. For Part 5, the tag component of message m'_x is $\geq t$, so it receives a tag that is $\geq t$ during the query phase of $\pi(x)$. Therefore, $tag(\pi(x)) \geq t$. Also, if $\pi(x)$ is a write, the effects of the query-fix(x) imply that $tag(\pi(x)) > t$.

Finally, we show Part 6. In the cm component of message m'_x, $cm(\ell) \neq \perp$ for all $\ell \leq k'$. Therefore, $truncate(cm)(\ell) = cm(\ell)$ for all $\ell \leq k'$, so $truncate(cm) \neq \perp$ for all $\ell \leq k'$.

Let cm' be the configuration map $extend(op(x).cmap_i, truncate(cm))$ computed by i during the effects of the recv event for m'_x. Since i does not reset $op(x).acc$ to $\emptyset$ in this step, by definition of the query-phase-start$(pi(x))$ event, it follows that $cm' \in Truncated$, and cm' is the value of $op(x).cmap_i$ just after the recv step.

Fix ℓ, $0 \leq \ell \leq k'$. We claim that $cm'(\ell) \neq \perp$. We consider cases:
1) $op(x).cmap(\ell)_i \neq \perp$ just before the recv step. Then the definition of $extend$ implies that $cm' \neq \perp$, as needed.
2) $op(x).cmap(\ell)_i = \perp$ just before the recv step and $truncate(cm)(\ell) \in C$. Then the definition of $extend$ implies that $cm'(\ell) \in C$, which implies that $cm'(\ell) \neq \perp$, as needed.
3) $op(x).cmap(\ell)_i = \perp$ just before the recv step and $truncate(cm)(\ell) \notin C$. Since $truncate(cm))(\ell) \neq \perp$, it follows that $truncate(cm)(\ell) \notin C$. By the case assumption, $op(x).cmap(\ell)_i = \perp$ just before the recv step. Since by Invariant 3, $op(x).cmap_i \in Truncated$, it follows that $op(x).cmap(\ell') = \perp$ before the recv step. Then by definition of $extend$, we have that $cm'(\ell) = \perp$ while $cm'(\ell) \in C$. This implies that $cm' \notin Truncated$, which contradicts the fact, already shown,that $cm' \in Truncated$. So this case cannot arise.

Since this argument holds for all ℓ, $0 \leq \ell \leq k'$, it follows that $cm'(\ell) \neq \perp$ for all $\ell \leq k'$. Since $cm'(\ell) \neq \perp$ for all $\ell \leq k'$, Invariant 2 implies that $cm' \in Usable$, which implies by definition

of $Usable$ that $cm'(\ell) \in C$ for some $\ell \geq k'$. That is, $op(x).cmap_i(\ell) \in C$ for some $\ell \geq k'$ immediately after the recv step. This implies that $query\text{-}cmap(\pi(x))(\ell) \in C$ for some $\ell \geq k'$, as needed. □

And finally, we consider the propagation phase of read and write operations.

Lemma 5: Suppose that a prop-fix$(x)_i$ event for a read or a write operation $\pi(x)$ on object x occurs in α. Suppose $prop\text{-}cmap(\pi(x))(k) \in C$ and $j \in W(\pi(x), k)$.
Then there exist messages m_x from i to j and m'_x from j to i such that:

1) obj field of messages m_x and m'_x equals x.
2) m_x is sent after the prop-phase-start$(\pi(x))$ event.
3) m'_x is sent after j receives m_x.
4) m'_x is received before the prop-fix(x) event of $\pi(x)$.
5) In any state after j receives m_x, $tag(x)_j \geq tag(\pi(x))$.
6) If $cmap(\ell)_j \neq \perp$ for all $\ell < k'$ in any state before j sends m'_x, then $prop\text{-}cmap(\pi(x))(\ell) \in C$ for some $\ell \geq k'$.

Proof. The phase number discipline, on individual object, implies the existence of the claimed messages m_x and m'_x. For Part 5, let $m_x.tag$ be the tag in message m_x. Since m_x is sent after event prop-phase-start$(\pi(x))$, which is not earlier than query-fix$(x)_i$, it must be that $m_x.tag \geq tag(\pi(x))$. Therefore, by the effects of recv, just after j receives m_x, $tag(x)_j \geq m.tag \geq tag(\pi(x))$. Then monotonicity of $tag(x)_j$ implies that $tag(x)_j \geq tag(\pi(x))$ in any state after j receives m_x.

For Part 6, the proof is analogous to the proof of part 5 of Lemma 4. In fact, it is identical except for the final conclusion, which now says that $prop\text{-}cmap(\pi(x))(\ell) \in C$ for some $\ell \geq k'$. □

B. Atomic Consistency

We now proceed to prove atomicity of the service in stages. First, in Section III-B.1 we present lemmas describing the relationship between configuration upgrade operations. We show in detail how object information is propagated during the configuration upgrade operation. Section III-B.2 describes the relationship between read/write operations and configuration upgrade operations. Section III-B.3 then considers two read or write operations on the same object, culminating in Lemma 14 that says that tags are monotone with respect to non-concurrent read or write operations on an object. Finally, Section III-B.4 uses the tags to define a partial order on operations that has sufficient properties (given in Section II-A) to claim atomicity.

1) Behavior of configuration upgrade: Here we present lemmas describing information flow between configuration upgrade operations to assert the existence of a sequence of configuration upgrade operations with certain properties. In particular, the key property is that the tag of each object in the domain is monotonically increasing with respect to the specific sequence of upgrade operations, guaranteeing that the value/tag information is propagated to newer configurations. Observe that the statements and proofs in this section, with the exception of the proof of Lemma 8, remain unchanged (when compared to [1]–[3]). The reason is that the configuration upgrade is performed on an entire domain (hence on all objects simultaneously). Proof of Lemma 8 needs to be modified since it requires reasoning about the tag information of individual objects, hence we update the proof to reflect modifications of DO-RAMBO.

The first lemma shows that if all configuration upgrade operations remove two particular configurations together, then those two configurations are always in the same state in all *cmaps*.

Lemma 6 ([1]–[3]): Suppose that $k > 0$, and α is an execution in which no cfg-upg-prop-fix(k) event occurs. Suppose that cm is a $CMap$ appearing as one of the following in any state in α:

1) The cm component of some message in *in-transit*.
2) $cmap_i$ for any $i \in I$.

If $cm(k-1) = \pm$ then $cm(k) = \pm$.

The following corollary says that if a cfg-upgrade(k) event for an upgrade operation γ occurs in an execution, then there is some previous configuration upgrade γ' (that completes before γ starts) where the target of γ' is the configuration with the smallest index removed by γ.

Corollary 1: Let γ be a configuration upgrade operation, initiated by a cfg-upgrade$(k)_i$ event in α, and let $k_1 = \mathit{min}\{removal - set(\gamma)\}$. That is, k_1 is the smallest element such that $upg\text{-}cmap(\gamma)(k_1) \in C$. Assume $k_1 > 0$. Then a cfg-upg-prop-fix$(k_1)_j$ event for some configuration upgrade operation γ' occurs in α for some j such that the cfg-upg-prop-fix$_j$ event of γ' precedes the cfg-upgrade$(k)_i$ event in α.

The next lemma says that for a given configuration upgrade operation γ, there exists a sequence of preceding upgrade operations satisfying certain properties. The lemma begins by 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 that configuration—to add to the sequence. Therefore the constructed sequence may well contain the same configuration upgrade operation multiple times, if the operation has removed multiple configurations. If two elements in the sequence are distinct upgrade operations, then the earlier operation in the sequence 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. As a result of these properties, the configuration upgrade process obeys a sequential discipline.

Lemma 7 ([1]–[3]): If a cfg-upgrade$_i$ event for upgrade operation γ occurs in α such that $k \in removal\text{-}set(\gamma)$, then there exists a sequence (possibly containing repeated elements) of configuration upgrade operations $\gamma_0, \gamma_1, \ldots, \gamma_k$ with the following properties:

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

The sequential nature of configuration upgrade has a nice consequence for propagation of tags: for any sequence of upgrade operations (as in Lemma 7), $tag(x, \gamma_s)$ is nondecreasing in s.

Lemma 8: Let $\gamma_\ell, \ldots, \gamma_k$ be a sequence of configuration upgrade operation such that:

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

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

Proof. If $\gamma_s = \gamma_{s+1}$, then it is trivially true that $tag(x, \gamma_s) \leq tag(x, \gamma_{s+1})$, for all x. Therefore assume that $\gamma_s \neq \gamma_{s+1}$. This implies that cfg-upg-prop-fix event of γ_s precedes the cfg-upgrade event of γ_{s+1}. Let k be the largest element in $removal\text{-}set(\gamma_s)$. We know by assumption that $k+1 \in removal\text{-}set(\gamma_{s+1})$. Therefore, $W_2(\gamma_s)$, a write-quorum of configuration $c(k+1)$, has at least one element in common with $R(\gamma_{s+1}, k+1)$; label this process j. By Lemma 3, and the monotonicity of $tag(x)_j$, of each object x, after the cfg-upg-prop-fix event of γ_s we know that $tag(x)_j \geq tag(x, \gamma_s)$, again for each $x \in X_d$. Then by Lemma 2 for any x we know that $tag(x, \gamma_{s+1}) \geq tag(x, \gamma_s)$. Therefore, $tag(x, \gamma_s) \leq tag(x, \gamma_{s+1})$. □

The next result follows immediately from the above lemma by induction.

Corollary 2: Let $\gamma_\ell, \ldots, \gamma_k$ be a sequence of configuration upgrade operation such that:

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

Then $\forall\ s, s', x : 0 \leq s \leq s' < k, x \in X_d,\ tag(x, \gamma_s) \leq tag(x, \gamma_{s'})$.

2) Behavior of a read or a write following a configuration upgrade: Now we describe the relationship between an upgrade operation and a following read or write operation. The following three lemmas relate $removal\text{-}set$ of a preceding configuration upgrade operation with the $query\text{-}cmap$ of a later read or write operation.

The first lemma shows that if, for some read or write operation $\pi(x)$, k is the smallest index such that $query\text{-}cmap(\pi(x))(k) \in C$, then some configuration upgrade operation with target k precedes the read or write operation.

Lemma 9: For some object x, let $\pi(x)$ be a read or write operation whose 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 $k = target(\gamma)$, and the cfg-upg-prop-fix event of γ precedes the query-phase-start$(\pi(x))$.

Proof. Follows from Lemma 6. Let s be the state just before event query-phase-start$(\pi(x))$. By definition, $query\text{-}cmap(\pi(x)) = s.cmap_i$. Since $s.cmap(k-1)_i = \pm$ and $s.cmap(k)_i \neq \pm$, there must exist such a configuration upgrade for k by the contrapositive of Lemma 6. □

Second, if some upgrade removing k does complete before the query-phase-start event of a read or write operation, then some configuration with index $\geq k+1$ must be included in the $query\text{-}cmap$ of a latter read or write operation.

Lemma 10: Let γ be a configuration upgrade operation such that $k \in removal\text{-}set(\gamma)$. Let $\pi(x)$ be a read or write operation on object x whose query-fix(x) event occurs in α. Suppose that the cfg-upg-prop-fix event of γ precedes the query-phase-start$(\pi(x))$ event in α. Then $query\text{-}cmap(\pi(x))(\ell) \in C$ for some $\ell \geq k+1$.

Proof. Suppose for the sake of contradiction that $query\text{-}cmap(\pi(x))(\ell) \notin C$ for all $\ell \geq k+1$. Fix $k' = \max(\{\ell' : query\text{-}cmap(\pi(x))(\ell') \in C\})$. Then $k' \leq k$.

Let $\gamma_0, \ldots, \gamma_k$ be the sequence of upgrade operations whose existence is asserted by Lemma 7, where $\gamma_k = \gamma$. Then, by this construction, $k' \in \mathit{removal\text{-}set}(\gamma_{k'})$, and the cfg-upg-prop-fix event of $\gamma_{k'}$ does not come after the cfg-upg-prop-fix event of γ in α. By assumption, the cfg-upg-prop-fix event of γ precedes the query-phase-start$(\pi(x))$ event in α. Therefore, the cfg-upg-prop-fix event of $\gamma_{k'}$ precedes the query-phase-start$(\pi(x))$ event in α.

Then, since $k' \in \mathit{removal\text{-}set}(\gamma_{k'})$, write-quorum $W_1(\gamma_{k'}, k')$ is defined. Since $\mathit{query\text{-}cmap}(\pi(x))(k') \in C$, the read-quorum $R(\pi(x), k')$ is defined. Choose $j \in W_1(\gamma_{k'}, k') \cap R(\pi(x), k')$. Assume that $k_t = \mathit{target}(\gamma_{k'})$. Notice that $k' < k_t$. Then Lemma 2 and monotonicity of cmap imply that, in the state just prior to the cfg-upg-query-fix event of $\gamma_{k'}$, $\mathit{cmap}(\ell)_j \neq \perp$ for all $\ell \leq k_t$. Then Lemma 4 implies that $\mathit{query\text{-}cmap}(\pi(x))(\ell) \in C$ for some $\ell \geq k_t$. But this contradicts the choice of k'. □

The next lemma describes the propagation of object tag information from a configuration upgrade operation to a following read or write operation.

Lemma 11: Let α be an execution with a configuration upgrade operation γ. Assume that $k = \mathit{target}(\gamma)$. Let $\pi(x)$ be a read or write operation on object x with event query-fix(x) in α. Suppose that event cfg-upg-prop-fix of γ precedes event query-phase-start$(\pi(x))$. Suppose also that $\mathit{query\text{-}cmap}(\pi(x))(k) \in C$. Then:
(1) $\mathit{tag}(x, \gamma) \leq \mathit{tag}(\pi(x))$, and (2) If $\pi(x)$ is a write operation then $\mathit{tag}(x, \gamma) < \mathit{tag}(\pi(x))$.

Proof. The propagation phase of γ accesses write quorum $W_2(\gamma)$ of $c(k)$, whereas the query phase of $\pi(x)$ accesses read-quorum $R(\pi(x), k)$. Since both are quorums of configuration $c(k)$, they have a nonempty intersection, hence choose $j \in W_2(\gamma) \cap R(\pi(x), k)$.

Lemma 3 implies that, in any state after the cfg-upg-prop-fix event for γ, $\mathit{tag}(x)_j \geq \mathit{tag}(x, \gamma)$. Since the cfg-upg-prop-fix event of γ precedes the query-phase-start$(\pi(x))$ event, we have that $t \geq \mathit{tag}(x)$, where t is defined to be the value of $\mathit{tag}(x)_j$ just before the query-phase-start$(\pi(x))$ event. Then Lemma 4 implies that $\mathit{tag}(\pi(x)) \geq t$, and if $\pi(x)$ is a write operation, then $\mathit{tag}(\pi(x)) > t$. Combining the inequalities yields both conclusions of the lemma. □

3) Behavior of sequential reads and writes: For two read or write operations on some object x that execute sequentially, we can prove certain relationships between their *query-cmaps*, *prop-cmaps*, and *tags*. Lemma 12 says that when two read or write operations on x execute sequentially, the smallest configuration index used in the propagation phase of the first operation is no higher than the largest index used in the query phase of the second.

Lemma 12: Let $\pi(x)_1$ and $\pi(x)_2$ be two read or write operations on object x, such that:
1) The prop-fix(x) event of $\pi(x)_1$ occurs in α.
2) The query-fix(x) event of $\pi(x)_2$ occurs in α.
3) The prop-fix(x) event of $\pi(x)_1$ precedes the query-phase-start$(\pi(x)_2)$ event.
Then $\mathit{min}(\{\ell : \mathit{prop\text{-}cmap}(\pi(x)_1)(\ell) \in C\}) \leq \mathit{max}(\{\ell : \mathit{query\text{-}cmap}(\pi(x)_2)(\ell)\})$.

Proof. Suppose for the sake of contradiction that $\mathit{min}(\{\ell : \mathit{prop\text{-}cmap}(\pi(x)_1)(\ell) \in C\}) > k$, where k is defined to be $\mathit{max}(\{\ell : \mathit{query\text{-}cmap}(\pi(x)_2)(\ell)\})$. Then in particular, $\mathit{prop\text{-}cmap}(\pi(x)_1)(k) \notin C$. The form of $\mathit{prop\text{-}cmap}(\pi(x)_1)$, as expressed in Invariant 3, implies that $\mathit{prop\text{-}cmap}(\pi(x)_1)(k) = \pm$.

This implies that some cfg-upg-prop-fix event for some upgrade operation γ such that $k \in \mathit{removal\text{-}set}(\gamma)$ occurs prior to the prop-fix(x) of $\pi(x)_1$, and hence prior to the

query-phase-start$(\pi(x)_2)$ event. Lemma 10 then implies that $query\text{-}cmap(\pi(x)_2)(\ell) \in C$ for some $\ell \geq k+1$. But this contradicts the choice of k. □

The next 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 a configuration.

Lemma 13: Assume $\pi(x)_1$ and $\pi(x)_2$ are two read or write operations on some object x, and:

1) The prop-fix(x) event of $\pi(x)_1$ occurs in α.
2) The query-fix(x) event of $\pi(x)_2$ occurs in α.
3) The prop-fix(x) event of $\pi(x)_1$ precedes the query-phase-start$(\pi(x)_2)$ event.
4) $prop\text{-}cmap(\pi(x)_1)(k)$ and $query\text{-}cmap(\pi(x)_2)(k)$ are both in C, for some $k \in \mathbb{N}$.

Then: (1) $tag(\pi(x)_1) \leq tag(\pi(x)_2)$, and (2) If $\pi(x)_2$ is a write then $tag(\pi(x)_1) < tag(\pi(x)_2)$.

Proof. The hypothesis imply that $prop\text{-}cmap(\pi(x)_1)(k) = query\text{-}fix(\pi(x)_2)(k) = c(k)$. Then $W(\pi(x)_1, k)$ and $R(\pi(x)_2, k)$ are both defined in α. Since they are both quorums of configuration $c(k)$, they have a nonempty intersection; choose $j \in W(\pi(x)_1, k) \cap R(\pi(x)_2, k)$.

Lemma 5 implies that, in any state after the prop-fix(x) event of $\pi(x)_1$, $tag(x)_j \geq tag(\pi(x)_1)$. Since the prop-fix$(x)$ event of $\pi(x)_1$ precedes the query-phase-start$(\pi(x)_2)$ event, we have that $t \geq tag(\pi(x)_1)$, where t is defined to be the value of $tag(x)_j$ just before the query-phase-start$(\pi(x)_2)$ event. Then Lemma 4 implies that $tag(\pi(x)_2) \geq t$, and if $\pi(x)_2$ is a write operation, then $tag(\pi(x)_2) > t$. Combining the inequalities yields both conclusions. □

The following lemma is similar to the previous one, but it does not assume that the propagation phase of the first operation and the query phase of the second operation share a configuration. The focus of the proof is on the situation where all configuration indices used in the query phase of the second operation are greater than those used in the propagation of the first operation.

Lemma 14: Assume $\pi(x)_1$ and $\pi(x)_2$ are two read or write operations on object x, and:

- prop-fix(x) of $\pi(x)_1$ occurs in α.
- query-fix(x) of $\pi(x)_2$ occurs in α.
- prop-fix(x) event of $\pi(x)_1$ precedes the $query\text{-}phase\text{-}start(\pi(x)_2)$ event.

Then: (1) $tag(\pi(x)_1) \leq tag(\pi(x)_2)$, and (2) If $\pi(x)_2$ is a write then $tag(\pi(x)_1) < tag(\pi(x)_2)$.

Proof. Let i_1 and i_2 be the indices of the processes that run operations $\pi(x)_1$ and $\pi(x)_2$, respectively. Let $cm_1 = prop\text{-}cmap(\pi(x)_1)$ and $cm_2 = query\text{-}cmap(\pi(x)_2)$. If there exists k such that $cm_1(k) \in C$ and $cm_2(k) \in C$, then Lemma 13 implies the conclusions of the lemma. So from now on, we assume that no such k exists.

Lemma 12 implies that $\min(\{\ell : cm_1(\ell) \in C\}) \leq \max(\{\ell : cm_2(\ell) \in C\})$. Invariant 3 implies that the set of indices used in each phase consists of consecutive integers. Since the intervals have no indices in common, it follows that $s_1 < s_2$, where s_1 is defined to be $\max(\{\ell : cm_1(\ell) \in C\})$ and s_2 to be $\min(\{\ell : cm_2(\ell) \in C\})$.

Lemma 9 implies that there exists a configuration upgrade operation that we will call γ_{s_2-1} such that $s_2 = target(\gamma_{s_2-1})$, and the cfg-upg-prop-fix of γ_{s_2-1} precedes the query-phase-start$(\pi(x)_2)$ event. Then by Lemma 11, $tag(x, \gamma_{s_2-1}) \leq tag(\pi(x)_2)$, and if $\pi(x)_2$ is a write operation than $tag(x, \gamma_{s_2-1}) < tag(\pi(x)_2)$.

Next we will demonstrate a chain of configuration upgrade operation with non-decreasing tags. Lemma 7, in conjunction with the already defined γ_{s_2-1}, implies the existence of a sequence of configuration upgrade operations $\gamma_0, \ldots, \gamma_{s_2-1}$ such that:

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

As a special case of above first property, since $s_1 \leq s_2 - 1$, we know that $s_1 \in \mathit{removal\text{-}set}(\gamma_{s_1})$. Then Corollary 2 implies that $\mathit{tag}(x, \gamma_{s_1}) \leq \mathit{tag}(x, \gamma_{s_2-1})$.

It remains to show that the tag of $\pi(x)_1$ is no greater than the tag of γ_{s_1}. Therefore we focus on the relationship between operation $\pi(x)_1$ and configuration upgrade γ_{s_1}. The propagation phase of $\pi(x)_1$ accesses write-quorum $W(\pi(x)_1, s_1)$ of configuration $c(s_1)$, whereas the query phase of γ_{s_1} accesses read-quorum $R(\gamma_{s_1}, s_1)$ of configuration $c(s_1)$. Since $W(\pi(x)_1, s_1) \cap R(\gamma_{s_1}, s_1) \neq \emptyset$, we may fix some $j \in W(\pi(x)_1, s_1) \cap R(\gamma_{s_1}, s_1)$. Let message $m_{x,1}$ from i_1 to j and message $m'_{x,1}$ from j to i_1 be as in Lemma 5 for the propagation phase of γ_{s_1}.

Let message $m_{x,2}$ from the process running γ_{s_1} to j and message $m'_{x,2}$ from j to the process running γ_{s_1} be the message whose existence is asserted in Lemma 2 for the query phase of γ_{s_1}.

We claim that j sends $m'_{x,1}$, its message for $\pi(x)_1$, before it sends $m'_{x,2}$, its message for γ_{s_1}. Suppose for the sake of contradiction that j sends $m'_{x,2}$ before it sends $m'_{x,1}$. Assume that $s_t = \mathit{target}(\gamma_{s_1})$. Notice that $s_t > s_1$, since $s_1 \in \mathit{removal\text{-}set}(\gamma_{s_1})$. Lemma 2 implies that in any state after j receives m_2, before j sends $m'_{x,2}$, $\mathit{cmap}(k)_j \neq \perp$ for all $k \leq s_t$. Since j sends $m'_{x,2}$ before it sends $m'_{x,1}$, monotonicity of *cmap* implies that just before j sends $m'_{x,1}$, $\mathit{cmap}(k)_j \neq \perp$ for all $k \leq s_t$. Then Lemma 5 implies that $\mathit{prop\text{-}cmap}(\pi(x)_1)(\ell) \in C$ for some $\ell \geq s_t$. But this contradicts the choice of s_1, since $s_1 < s_t$. This implies that j sends $m'_{x,1}$ before it sends $m'_{x,2}$.

Since j sends $m'_{x,1}$ before it sends $m'_{x,2}$, Lemma 5 implies that, at the time j sends $m'_{x,2}$, $\mathit{tag}(\pi(x)_1) \leq \mathit{tag}(x)_j$. Lemma 2 implies that $\mathit{tag}(\pi(x)_1) \leq \mathit{tag}(x, \gamma_{s_1})$. From above, we know that $\mathit{tag}(x, \gamma_{s_1}) \leq \mathit{tag}(x, \gamma_{s_2-1})$, and $\mathit{tag}(x, \gamma_{s_2-1}) \leq \mathit{tag}(x, \gamma_{\pi(x)_2})$, and if $\pi(x)_2$ is a write then $\mathit{tag}(x, \gamma_{s_2-1}) < \mathit{tag}(x, \gamma_{\pi(x)_2})$. Combining the inequalities yields both conclusions. □

4) Atomicity: We now proceed to prove atomicity of DO-RAMBO by showing that in any good execution, properties P1, P2, P3, and P4 (stated in Section II-A) hold for any object.

Let β be a trace of $\mathcal{S}$, the system that implements DO-RAMBO, where all read and write operations on some object $x \in X_d$ complete. Consider any particular good execution α of $\mathcal{S}$ whose trace is β. We define a partial order $\prec_x$ on read and write operations on x in β, in terms of the operation tags in α. Namely, we totally order the writes in order of their tags, and we order each read with respect to all writes as follows: a read with tag t is ordered after all writes with tags $\leq t$ and before all writes with tags $> t$.

Lemma 15: The ordering $\prec_x$ is well-defined, for all $x \in X_d$.

Proof. The key is to show that no two write operations on some object x get assigned the same tag. This is obviously true for two writes that are initiated at different locations, because the low-order tiebreaker identifiers are different. For two writes at the same location, for the same object x, Lemma 14 implies that the tag of the second is greater than the tag of the first. This suffices. □

Lemma 16: The order $\prec_x$, for all $x \in X_d$, satisfies properties P1, P2, P3, and P4.

Proof. We begin with property P2, the most interesting one. We consider two operations $\pi(x)_1$ and $\pi(x)_2$ on object x. Now, suppose for the sake of contradiction that $\pi(x)_1$ completes before $\pi(x)_2$ starts, yet $\pi(x)_2 \prec_x \pi(x)_1$. We consider two cases:

1) $\pi(x)_2$ is a write operation. Since $\pi(x)_1$ completes before $\pi(x)_2$, Lemma 14 implies that $tag(\pi(x)_2) > tag(\pi(x)_1)$. On the other hand, the fact that $\pi(x)_2 \prec_x \pi(x)_1$ implies that $tag(\pi(x)_2) \leq tag(\pi(x)_1)$, hence contradiction.
2) $\pi(x)_2$ is a read operation. Since $\pi(x)_1$ completes before $\pi(x)_2$ starts, Lemma 14 implies that $tag(\pi(x)_2) \geq tag(\pi(x)_1)$. On the other hand, the fact that $\pi(x)_2 \prec_x \pi(x)_1$ implies that $tag(\pi(x)_2) < tag(\pi(x)_1)$, hence a contradiction.

Since we have a contradiction in either case, property P2 must hold.

Property P1 follows from property P2. Properties P3 and P4 are straightforward. □

Finally, we tie everything together and show safety of our implementation $\mathcal{S}$, assuming the environment safety assumptions (Section II-H).

Theorem 2: Let β be a trace of system $\mathcal{S}$ that implements DO-RAMBO. Then β satisfies the atomicity guarantee for each object x.

Proof. Assume that all read and write operations complete in β. Let α be a good execution of $\mathcal{S}$ whose trace is β. For all objects $x \in X_d$ define the ordering $\prec_x$, on the read and write operations for each object x in β as above using the execution α. Then Lemma 16 says that $\prec_x$ satisfies the four conditions in the definition of atomicity for each $x \in D$. Thus, β satisfies the atomicity condition for all objects as needed. □

IV. CONDITIONAL OPERATION LATENCY ANALYSIS

A conditional analysis of RAMBO read, write, and configuration upgrade operation latency is presented in [1]–[4]. Here we show that under the same conditions, these operations in DO-RAMBO have the same latency. We start by giving relevant definitions (following [2], [3]). Let δ denote the maximum message delivery latency. Also let δ be the interval at which the gossip messages are sent. Assume α is an admissible timed execution, and α' a finite prefix of α. Let $\ell time(\alpha')$ denote the time of the last event in α'. We say α is an α'-*normal* execution if (i) after α', the local clocks of all automata progress at exactly the rate of real time, (ii) no message sent in α after α' is lost, and (iii) if a message is sent at time t in α and it is delivered, then it is delivered by the time $\max\{t + \delta, \ell time(\alpha') + \delta\}$.

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

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

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

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

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

As in [2], [3], we assume that α is an α'-$normal$ execution, satisfying $(\alpha'$,*e*,$23\delta)$-*configuration-viability*, $(\alpha', 8\delta)$-*recon-spacing*, (α', e)-*join-connectivity*, and $(\alpha', e + \delta)$-*recon-readiness*.

The following theorems give the latency bounds on read, write, and configuration upgrade operations under the stated timing assumptions. These results apply to DO-RAMBO because in terms of messaging, our read-write protocol for an object is identical to that of RAMBO. Moreover, the configuration upgrade operation is similar to the previous RAMBO algorithms, where the only differences are semantic: we manage information per domain as opposed to managing it per object. Hence, we forgo detailed proofs of the following theorems as they are identical to those in [1]–[4] (except for the notation in domain vs. object indexing).

Theorem 3 ([1]–[4]): Let α be an α'-$normal$ execution of DO-RAMBO satisfying join-connectivity, recon-readiness, recon-spacing, and configuration-viability. Let $t > \ell time(\alpha') + e + \delta$. Assume i is a process that received a $\mathsf{join\text{-}ack}_i$ prior to time $t - e - \delta$, and neither fails nor departs 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 $t + 8\delta$.

Recall that message delay is bounded by δ and local processing takes zero time. Since after time t messages are not lost and nodes do not fail and the node initiating an operation has already joined the service, the thesis of Theorem 3 follows from the following observation. The bound of $t + 8\delta$ represents the sum of a maximum duration of the two phases comprising a read or a write operation. Each phase can be interrupted by an ongoing reconfiguration where a new quorum system is detected while processing messages for the current phase. From timing assumptions, each phase can be interrupted exactly once, hence the result follows.

Theorem 4 ([1]–[4]): Let α be an α'-$normal$ execution of DO-RAMBO satisfying join-connectivity, recon-readiness, recon-spacing, configuration-viability. Assume that $t > \ell time(\alpha') + e + \delta$, and that a $\mathsf{cfg\text{-}upgrade}(k)_i$ occurs at time t at node i. Assume that node i does not depart or fail before $t + 4\delta$. Then $\mathsf{cfg\text{-}upg\text{-}ack}(k)_i$ occurs no later than time $t + 4\delta$.

Configuration upgrade proceeds independently from configuration installation and any read/write operations. The duration of configuration upgrade is bounded by the maximum duration of the two phases involved in this operation, hence the thesis of Theorem 4 follows.

V. IMPLEMENTATION AND EVALUATION

We now present the empirical results obtained from our implementations of RAMBO and DO-RAMBO on a LAN, comparing the performance of the two implementations using three different experimental settings. We note that our two implementations differ only in the introduction of domains in DO-RAMBO, while all low-level sequencing of control and communication carried out in response to client requests is essentially the same in both systems. Thus we believe that

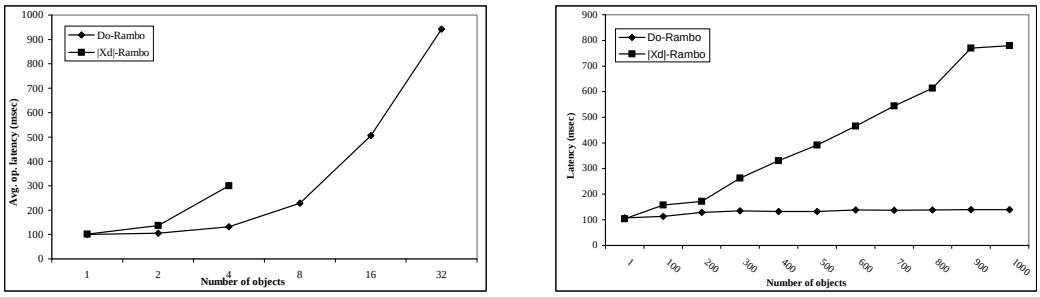

Fig. 8. Left:DO-RAMBO vs. the composition of $|X_d|$ instances of RAMBO; Right: DO-RAMBO vs. RAMBO for a single "super-object" of $|X_d|$ objects.

our experimental results indeed reveal differences in performance that are due to the domain-oriented approach implemented in DO-RAMBO. The results presented in this section support our expectation that grouping objects into domains leads to improved performance.

We manually translated the Input/Output Automata specifications of RAMBO and DO-RAMBO 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 [7]. The target platform consists of a cluster with nodes running Linux that are dedicated to the project. The nodes are various Pentium processors up to 900 MHz interconnected via a 100Mbps Ethernet switch.

Each instance of RAMBO and 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. Management of *common* state variables in RAMBO and DO-RAMBO, such as $world$, $cmap$, is identical. The *Reader-Writer* service is implemented as described in this paper. However, we make one simple optimization in the implementation of DO-RAMBO relative to its specification. In the specification of DO-RAMBO we assume that each gossip message is per object (containing *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. It is worth to mention that the memory location in our experiments is implemented as a Java Integer.

Experiment 1: *Grouping objects into a domain under a stable configuration.* This experiment is designed to 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. To eliminate the effects of reconfiguration (that are likely to further benefit DO-RAMBO), a single stable configuration is used in this experiment.

In this experiment, there are ten nodes that do not leave the system and a single configuration is installed that includes all of these nodes as members. The configuration does not change over time and consists of majorities, of at least six nodes each.

As the domain increases, additional instances of RAMBO service are needed to support new objects added to the domain. For domain size one a single RAMBO service suffices and we expect to see same performance as that of DO-RAMBO with $|X_d| = 1$. However, each addition of RAMBO introduces overhead that DO-RAMBO removes by consolidating all objects into a single domain. Therefore, we expect that DO-RAMBO will outperform the composition of RAMBO services as the size of domain increases.

Figure 8 (left) presents average latency of read/write operations (over all objects and all nodes) as the number of objects grows from 1 to 32. The data points represent averages collected over a series of runs. 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 network platform was not capable of executing concurrently more than eight instances of RAMBO.

A possible explanation of this phenomenon is the rapidly growing communication burden within the increasing number of RAMBO components. The performance comparison of the two systems substantiates our claim that DO-RAMBO is a more practical system.

Experiment 2: *Performance modeling of a domain in* RAMBO *under a stable configuration.* This experiment is designed to compare the performance of DO-RAMBO to a single RAMBO instance that encapsulates objects of the entire domain in a single object that we call super-object. This is done to allow RAMBO to "model" a domain with the goal of measuring its performance. *In this experiment we choose a single object from the domain on which a single chosen process performs read and write operations.*

Note that this experiment is designed to measure performance only — the semantics of objects is changed when using the super-object approach to model domains in RAMBO. With a super-object, a write to a single object is accomplished by reading the super-object (the entire modeled domain), modifying the value of the object, and writing the super-object. Therefore, if two writers are attempting to concurrently perform write operations on two different objects within a super-object, then one write can possibly undo the effects of the other write on the different object. However, conducting the experiment is still meaningful single-writer/multiple-reader systems, where the writer issues one write at the time per domain.

The setup for this experiment is as in Experiment 1. Here each of the ten nodes is a member of the configuration installed and used during data collection, where this configuration does not change over time. Nodes do not fail or depart during the experiment.

Unlike in the previous experiment, this time a single instance of RAMBO service is used. Hence, there is no overhead associated with running multiple *Reader-Writer* and *Recon* services. However, RAMBO sees the domain as a single object and it cannot benefit from the mentioned earlier simple communication optimization applied to DO-RAMBO. Meaning, DO-RAMBO is aware of the individual objects that compose the domain and can respond to a read/write request with a message that includes information pertaining to the specific request. Whereas, RAMBO is not aware of the internal structure of the super-object, hence whenever a request is made to access some object within the super-object the resulting messages must include the entire super-object. We expect performance of RAMBO to decrease as the size of the domain increases – larger message size causes increase in message latency and network throughput.

Figure 8 (right) presents the average latency of read/write operations (over all nodes) as the number of objects in the domain increases from 1 to 1000. The chart shows that DO-RAMBO outperforms the single super-object RAMBO. As the number of objects increases so does the size of the messages exchanged by RAMBO, hence degrading operation latency. In comparison, messages in DO-RAMBO (in this experiment) include information for a single object only, hence are of constant size. Therefore, as the size of the domain increases the message latency remains unchanged, hence resulting in roughly constant latency for read and write operations.

Experiment 3: *Impact of reconfigurations.* This last experiment is designed to measure the impact of reconfigurations on the performance of DO-RAMBO and RAMBO systems. The system tested

implements a three object memory system. In the case of DO-RAMBO this means a domain of size three. The RAMBO-based system is a composition of three RAMBO instances, one for each object. More specifically, we measure the impact of reconfiguration on the throughput of each system in terms of the number of read and write operations per second.

For this experiment nine nodes were used. We run one copy of DO-RAMBO per node. One nodes act as a reconfigurer, where new configurations are submitted with varying delays between completion of one reconfiguration request and submission of another to allow us to throttle frequency of reconfiguration. We use two configurations of four nodes, with no members in common, and the reconfigurer alternates between the two. The remaining nodes continuously perform read and write operations, where read and write requests locally alternate, and three nodes access the first object, three nodes access the second object, and two nodes access the third. There are a total of 500 read and write operations initiated at each node, and each data point on the graph in Figure 9 represents an average system throughput that is computed over all operations and all nodes.

To assess the behavior of RAMBO, we used three instances of the implementation, where three copies of RAMBO are run on each node. Again, nine nodes are used and one is chosen as a reconfigurer for each RAMBO service. Configurations used are as before. Total of eight instances are chosen to perform the read/write test using the same setup as in the test of DO-RAMBO. The data points on the graph represent system throughput that is computed using average operation latency (as explained in the DO-RAMBO part of this experiment).

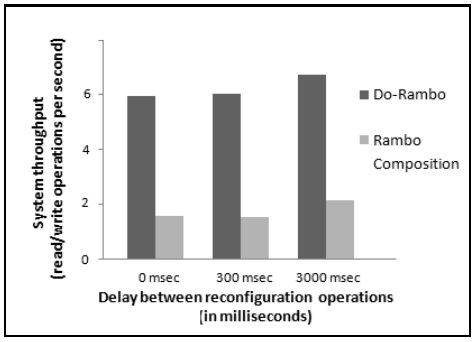

Fig. 9. Throughput during reconfigurations.

As it was the case in Experiment 1, we expect that the overhead caused by running multiple instances of *Reader-Writer* and *Recon* service will result in the composition of RAMBO services to have poor performance. The compelling reason supporting our expectation is that the *Recon* service utilizes, communication expensive, consensus to ensure total ordering of installed configurations. This is regardless of the fact that there is only one reconfigurer present in this experiment, since the implementation allows for any number of configuration proposals to be submitted to the system at the same time. Therefore, we expect the performance of RAMBO composition to degrade as the frequency of reconfiguration increases. The data in Figure 9 indicates that DO-RAMBO outperforms the composition of three RAMBO services in the presence of reconfigurations. As expected, a likely reason for this behavior is that RAMBO running three instances of reconfiguration, requiring consensus, generates a significant number of messages hence leading to increased network latency. With increased latency and volume of messages, the messages used by read and write operations also require more time for delivery and processing, hence negatively impacting the latency of these operations.

VI. DISCUSSION

RAMBO [1] is an atomic memory service for dynamic networks. Several proposals were recently made to make this service more practical [2]–[4], [7]. An implementation of RAMBO is presented in [7]. These successive improvements improved 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 inefficient. In this paper we presented a specification and an efficient implementation of a 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 compared its performance to the performance of a similar implementation of the prior RAMBO service.

One remaining interesting question is whether our approach can be used to implement a snapshot operation for a set of related registers. We intend to pursue research in this direction.

Acknowledgements. The authors thank the anonymous referees for insightful comments that allowed us to improve the presentation of our work. We also acknowledge the question of one referee regarding the possibility of implementing snapshot memory using our approach.

REFERENCES

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

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

[3] ——, "RAMBO: A robust, reconfigurable atomic memory service for dynamic networks," CSAIL,MIT, Tech. Rep., 2008.

[4] C. Georgiou, P. Musial, and A. Shvartsman, "Long-lived RAMBO: Trading knowledge for communication," *Theoretical Computer Science*, vol. 383(1), pp. 59–85, September 2007.

[5] N. Lynch and M. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, 1987, pp. 137–151.

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

[7] P. Musial and A. Shvartsman, "Implementing a reconfigurable atomic memory service for dynamic networks," in *Proceedings of 18th International Parallel and Distributed Symposium*, 2004.

[8] D. Gifford, "Weighted voting for replicated data," in *Proceedings of 7th ACM Symp. on Oper. Sys. Princ.*, 1979, pp. 150–162.

[9] R. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems*, vol. 4(2), pp. 180–209, June 1979.

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

[11] B. Awerbuch and P. Vitanyi, "Atomic shared register access by asynchronous hardware," in *Proceedings of 27th IEEE Symposium on Foundations of Computer Science*, 1986, pp. 233–243.

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

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

[14] N. Lynch and A. Shvartsman, "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," in *Proceedings of 27th International Symposium on Fault-Tolerant Computing*, 1997, pp. 272–281.

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

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

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