Reconfigurable Distributed Storage for Dynamic Networks[⋆]

Gregory Chockler[1], Seth Gilbert[1], Vincent Gramoli[2,3], Peter M Musial[2], and Alexander A Shvartsman[1,2]

[1] CSAIL, Massachusetts Institute of Technology, Cambridge, MA 02139, USA.
grishac@csail.mit.edu, sethg@mit.edu, alex@theory.csail.mit.edu
[2] Dep. of Comp. Sci. and Eng., University of Connecticut, Storrs, CT 06269, USA.
piotr@cse.uconn.edu
[3] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France.
vgramoli@irisa.fr

Abstract. This paper presents a new algorithm, *RDS* (Reconfigurable Distributed Storage), for implementing a reconfigurable distributed shared memory in an asynchronous dynamic network. The algorithm guarantees atomic consistency (linearizability) in all executions in the presence of arbitrary crash failures of processors and message loss and delays. The algorithm incorporates a quorum-based read/write algorithm and an optimized consensus protocol, based on Paxos. RDS achieves the design goals of: (i) allowing read and write operations to complete rapidly, and (ii) providing long-term fault tolerance through reconfiguration, a process that evolves the quorum configurations used by the read and write operations. The new algorithm improves on previously developed alternatives by using a more efficient reconfiguration protocol, thus guaranteeing better fault tolerance and faster recovery from network instability. This paper presents RDS, a formal proof of correctness, conditional performance analysis, and experimental results.

Keywords: Distributed algorithms, reconfiguration, atomic objects, performance.

1 Introduction

Providing consistent and available data storage in a dynamic network is an important basic service for modern distributed applications. To be able to tolerate failures, such services must replicate data, which results in the challenging problem of maintaining consistency despite a continually changing computation and communication medium. The techniques that were previously developed to maintain consistent data in static network are largely inadequate for the dynamic settings of extant and emerging networks.

Recently a new direction was proposed that integrates dynamic reconfiguration within a distributed data storage service. The goal of this research was to enable the storage service to guarantee consistency (safety) in the presence of asynchrony, arbitrary changes in the collection of participating network nodes, and varying connectivity. The original service, called RAMBO (Reconfigurable Atomic Memory for Basic Objects) [1, 2], supports multi-reader/multi-writer atomic objects in dynamic settings. The reconfiguration service is loosely coupled with the read/write service. This allows

[⋆] This work is supported in part by the NSF Grants 0311368 and 0121277.

for the service to separate data access from reconfiguration, during which the previous set of participating nodes can be upgraded to an arbitrary new set of participants. Of note, read and write operations can continue to make progress while the reconfiguration is ongoing. Reconfiguration is a two step process. First, the next configuration is agreed upon by the members of the previous configuration; then obsolete configurations are removed using a separate configuration upgrade process. As a result, multiple configurations can co-exist in the system if the removal of obsolete configurations is slow. This approach leads to an interesting dilemma. (a) On the one hand, decoupling the choice of new configurations from the removal of old configurations allows for better concurrency and simplified operation. Thus each operation requires weaker fault-tolerance assumptions. (b) On the other hand, the delay between the installation of a new configuration and the removal of obsolete configurations is increased. Delaying the removal of obsolete configurations can slow down reconfiguration, lead to multiple extant configurations, and require stronger fault-tolerance assumptions.

Our broader current research direction is to study the trade-off between the simplicity of loosely coupled reconfiguration protocols, as in [1, 2], and the fault tolerance properties that they require. This paper presents a new algorithm that more tightly integrates the two stages of reconfiguration. Our goal is to reduce the cost of reconfiguration, both in terms of latency and the fault-tolerance properties required of the configurations. We bound and reduce the time during which the old configurations need to remain active, without impacting the efficiency of data access operations. Reducing this time can substantially increase the fault-tolerance of the service, despite the more complicated integrated reconfiguration operation.

Contributions. In this paper we present a new distributed algorithm for implementing a read/write distributed shared memory in a dynamic asynchronous network. This algorithm, named *RDS* (Reconfigurable Distributed Storage), is fault-tolerant, using replication to ensure that data can survive node failures, and reconfigurable, tolerating continuous changes in the set of participating nodes. As in the original approach [1], we implement atomic (linearizable) object semantics, where in order to maintain consistency in the presence of small and transient changes, the algorithm uses *configurations* consisting of *quorums* of locations. Read and write operations consist of two phases, each phase accessing the needed read- or write-quorums. In order to tolerate significant changes in the computing medium we implement *reconfiguration* that evolves quorum configurations over time.

In RDS we take a radically different approach to reconfiguration. To speed up reconfiguration and reduce the time during which obsolete configurations must remain accessible, we present an integrated reconfiguration algorithm that overlays the protocol for choosing the next configuration with the protocol for removing obsolete configurations. The protocol for choosing and agreeing on the next configuration is based on an optimized version of Paxos [3–6]. The protocol for removing obsolete configurations is a two-phase protocol, involving quorums of the old and the new configurations.

In summary, RDS improves on the previous solutions [1, 2, 7] by using a more efficient reconfiguration protocol that relaxes some of the fault tolerance assumptions made in prior work and that provides faster recovery following network instability. In this paper we present the new algorithm, a formal proof of correctness, conditional performance results, and highly encouraging experimental results of additional operation latency due to reconfiguration. The highlights of our approach are as follows:

- *Read/write independence:* Read and write operations are independent of ongoing reconfigurations, and can make progress regardless of ongoing reconfiguration or the time it takes for reconfiguration to terminate (e.g., due to the instability of leaders selected by the reconfiguration algorithm). Even if the network is completely asynchronous, as long as reconfigurations are not too frequent (with respect to network latencies), then read and write operations ar able to complete.
- *Fully flexible reconfiguration:* The algorithm imposes no dependencies between the quorum configurations selected for installation.
- *Fast reconfiguration:* The reconfiguration uses a leader-based protocol; when the leader is stable, reconfigurations are very fast: 3 network delays. Since halting consensus requires at least 3 network delays, this is seemingly optimal. Combining quorum reconfiguration with optimized 3-delay "Fast Paxos" requires new techniques since (i) prior attempts to use Paxos for reconfiguration depend on each reconfiguration using the existing quorum system to install the next, while (ii) "Fast Paxos" uses preparatory work from earlier configurations that may be obsolete.
- *Fast read operations:* Read operations require only two message delays when no write operations interfere with it.
- *Fast recovery:* Our solution eliminates the need for recovery following network instability and the associated clean-up of obsolete quorum configurations. Specifically, and unlike the prior RAMBO algorithms [1, 2] that may generate a backlog of old configurations, there is never more than one old configuration at a time.

Our reconfiguration algorithm can be viewed as an example of protocol composition advocated by van der Meyden and Moses [8]. Instead of waiting for the establishment of a new configuration and then running the obsolete configuration removal protocol, we compose (or overlay) the two protocols so that the upgrade to the next configuration takes place as soon as possible.

Background. Several approaches have been used to implement consistent data in (static) distributed systems. Starting with the work of Gifford [9] and Thomas [10], many algorithms have used collections of intersecting sets of objects replicas (such as quorums) to solve the consistency problem. Upfal and Wigderson [11] use majority sets of readers and writers to emulate shared memory. Vitányi and Awerbuch [12] use matrices of registers where the rows and the columns are written and respectively read by specific processors. Attiya, Bar-Noy and Dolev [13] use majorities of processors to implement shared objects in static message passing systems. Extension for limited reconfiguration of quorum systems have also been explored [14, 15].

Virtually synchronous services [16], and group communication services (GCS) in general [17], can also be used to implement consistent data services, e.g., by implementing a global totally ordered broadcast. While the universe of processors in a GCS can evolve, in most implementations, forming a new view takes a substantial time, and client operations are interrupted during view formation. However the dynamic algorithms, such as the algorithm presented in this work and [1, 2, 7], allow reads and writes to make progress during reconfiguration.

Reconfigurable storage algorithms are finding their way into practical implementations [18, 19]. The new algorithm presented here has the potential of making further impact on system development.

Document Structure. Section 2 defines the model of computation. We present the algorithm in Section 3. In Section 4 we present the correctness proofs. In Section 5 we present conditional performance analysis of the algorithm. Section 6 contains experimental results about operation latency. The conclusions are in Section 7.

2 System Model and Definitions

We use a message-passing model with asynchronous processors that have unique identifiers (the set of processor identifiers need not be finite). Processors may crash. Processors communicate via point-to-point asynchronous unreliable channels. In normal operation any processor can send a message to any other processor. In safety (atomicity) proofs we do not make *any* assumptions about the length of time it takes for a message to be delivered.

To analyze the performance of the new algorithm, we make additional assumptions as to the performance of the underlying network. In particular, we assume that eventually (at some unknown point) the network stabilizes, becoming synchronous and delivering messages in bounded (but unknown) time. We also restrict the rate of reconfiguration after stabilization, and limit node failures such that some quorum remains available in an active configuration. (For example, in majority quorums, this means that only a minority of nodes in a configuration fail between reconfigurations.) We present a more detailed explanation in Section 5.

Our algorithm uses *quorum configurations*. A *configuration* c consists of three components: (i) $\mathit{members}(c)$, a finite set of processor ids, (ii) $\mathit{read\text{-}quorums}(c)$, a set of quorums, and (iii) $\mathit{write\text{-}quorums}(c)$, a set of quorums, where each quorum is a subset of $\mathit{members}(c)$. We require that the read quorums and write quorums of a common configuration intersect: formally, for every $R \in \mathit{read\text{-}quorums}(c)$ and $W \in \mathit{write\text{-}quorums}(c)$, the intersection $R \cap W \neq \emptyset$.

3 RDS Algorithm

In this section, we present a description of RDS. An overview of the algorithm appears in Figure 1 and Figure 2 (the algorithm is formally specified in the full paper). We present the algorithm for a single object; atomicity is preserved under composition and the complete shared memory is obtained by composing multiple objects. See [20] for an example of a more streamlined support of multiple objects.

In order to ensure fault tolerance, data is replicated at several nodes in the network. The key challenge, then, is to maintain the consistency among the replicas, even as the underlying set of replicas may be changing. The algorithm uses *quorum configurations* to maintain consistency, and *reconfiguration* to modify the set of replicas. During normal operation, there is a single active configuration; during reconfiguration, when the

$\mathsf{read}()$ or $\mathsf{write}(v)$ operation at node i:

- **RW-Phase-1a:** Node i chooses a unique id, t, and sends a $\langle RW1a, t\rangle$ message to a read quorum of every active configuration. Node i stores the set of active configurations in *op-configs*.
- **RW-Phase-1b:** If node j receives a $\langle RW1a, t\rangle$ message from i, it sends a $\langle RW1b, t, tag, value\rangle$ message back to node i.
- **RW-Phase-2a:** If node i receives a $\langle RW1b, t, tag, value\rangle$ message from j, it updates its tag and value. If it receives $RW1b$ messages from a read quorum of *all* configurations in *op-configs*, then the first phase is complete. If the ongoing operation is a read operation and the tag has already been confirmed, node i returns the current value; otherwise it sends a $\langle RW2a, t, tag', value'\rangle$ message to a write quorum of every active configuration where tag' and $value'$ depend on whether it is a read or a write operation: in the case of a read, they are just equal to the local *tag* and *value*; in the case of a write, they are a newly unique chosen tag, and v, the value to write. Node i resets *op-configs* to the set of active configurations.
- **RW-Phase-2b:** If node j receives a $\langle RW2a, t, tag, value\rangle$ message from i, then it updates its tag and value and sends to i a $\langle RW2b, t, configs\rangle$ message, where *configs* is the set of active configurations.
- **RW-Done:** If node i receives message $\langle RW2b, t, c\rangle$, it adds any new configurations from c to its set of active configurations and to *op-configs*. If it receives a $RW2b$ message from a write quorum of *all* configurations in *op-configs*, then the read or write operation is complete and the tag is marked confirmed. If it is a read operation, node i returns its current value to client.

Fig. 1. The phases of the read and write protocols. Each protocol requires up to two phases.

set of replicas is changing, there may be two active configurations. Throughout the algorithm, each node maintains a set of *active configurations*. During a reconfiguration, a new configuration is added to the set, and at the end of a reconfiguration the old configuration is removed.

Read and Write Operations. Read and write operations proceed by accessing the currently active configurations. Each replica maintains a *tag* and a *value* for the data being replicated. Tag is a counter-id pair used as a write operation version number where its node id serves as a tiebreaker. Each read or write operation potentially requires two phases: **RW-Phase-1** to *query* the replicas, learning the most up-to-date tag and value, and **RW-Phase-2** to *propagate* the tag and value to the replicas. In a *query* phase, the initiator contacts one read quorum from each active configuration, and remembers the largest tag and its associated value. In a *propagate* phase, read and write operations behave differently: a write operation chooses a new tag that is strictly larger than the one discovered in the query phase, and sends the new tag and new value to a write quorum; a read operation sends the tag and value discovered in the query phase to a write quorum.

Sometimes, a read operation can avoid performing the propagation phase, **RW-Phase-2**, if some prior read or write operation has already propagated that particular tag and value. Once a tag and value has been propagated, be it by a read or a write operation, it is marked confirmed. If a read operation discovers that a tag has been confirmed, it can skip the second phase.

One complication arises when during a phase, a new configuration becomes active. In this case, the read or write operation must access the new configuration as well as the

$\mathsf{recon}(c, c')$ at node i: If c is the only configuration in the set of active configurations, then reconfiguration can begin. The request is forwarded to the putative leader, ℓ. If it has already completed Phase 1 for some ballot b, then it can skip Phase 1, and use this ballot in Phase 2. Otherwise, it performs Phase 1.

- **Recon-Phase-1a:** Leader ℓ chooses a unique ballot number b larger than any previously used ballots and sends $\langle Recon1a, b\rangle$ messages to a read quorum of configuration c (the old configuration).
- **Recon-Phase-1b:** When node j receives $\langle Recon1a, b\rangle$ from ℓ, if it has not received any message with a ballot number greater than b, then it replies to ℓ with $\langle Recon1b, b, configs, \langle b', c''\rangle\rangle$ where *configs* is the set of active configurations and b' and c'' represent the largest ballot and configuration that j voted to replace configuration c.
- **Recon-Phase-2a:** If leader ℓ has received a $\langle Recon1b, b, configs, b'', c''\rangle$ message, it updates its set of active configurations; if it receives "Recon1b" messages from a read quorum of configuration c, then it sends a $\langle Recon2a, b, c, v\rangle$ message to a write quorum of configuration c, where: if all the $\langle Recon1b, b, \ldots\rangle$ messages contained empty last two parameters, then v is c'; otherwise, v is the configuration with the largest ballot received in the prepare phase.
- **Recon-Phase-2b:** If a node j receives $\langle Recon2a, b, c, c'\rangle$ from ℓ, and if c is the only active configuration, and if it has not already received any message with a ballot number greater than b, it sends $\langle Recon2b, b, c, c', tag, value\rangle$ to a read quorum and a write quorum of c.
- **Recon-Phase-3a:** If a node j receives $\langle Recon2b, b, c, c', tag, value\rangle$ from a read quorum and a write quorum of c, and if c is the only active configuration, then it updates its tag and value, and adds c' to the set of active configurations *and* to *op-configs*. It then sends a $\langle Recon3a, c, c', tag, value\rangle$ message to a read quorum and a write quorum of configuration c.
- **Recon-Phase-3b:** If a node j receives $\langle Recon3a, c, c', tag, value\rangle$ from a read quorum and a write quorum of configuration c, then it updates its tag and value, and removes configuration c from its active set of configurations (but not from *op-configs*, if it is there).

Fig. 2. The phases of the recon protocol. The protocol requires up to three phases.

old one. In order to accomplish this, read or write operations save the set of currently active configurations, *op-configs*, when a phase begins; a reconfiguration can only add configurations to this set—none are removed during the phase. Even if a reconfiguration finishes with a configuration, the read or write phase must continue to use it.

Reconfiguration. When a client wants to change the set of replicas, it initiates a reconfiguration, specifying a new configuration. The nodes then initiate a consensus protocol, ensuring that everyone agrees on the active configuration, and that there is a total ordering on configurations. The resulting protocol is somewhat more complicated than typical consensus, however, since at the same time, the reconfiguration operation propagates information from the old configuration to the new configuration.

The reconfiguration protocol uses an optimized variant of Paxos [3]. The reconfiguration request is forwarded to a leader, which coordinates the reconfiguration, consisting of three phases: a *prepare* phase, **Recon-Phase-1**, in which a ballot is made ready, a *propose* phase, **Recon-Phase-2**, in which the new configuration is proposed, and a *propagate* phase, **Recon-Phase-3**, in which the results are distributed.

The prepare phase accesses a read quorum of the old configuration, thus learning about any earlier ballots. When the leader concludes the prepare phase, it chooses a configuration to propose: if no configurations have been proposed to replace the current old configuration, the leader can propose its own preferred configuration; otherwise, the leader must choose the previously proposed configuration with the largest ballot. The propose phase then begins, accessing both a read and a write quorum of the old configuration. This serves two purposes: it requires that the nodes in the old configuration vote on the new configuration, and it collects information on the tag and value from the old configuration. Finally, the propagate phase accesses a read and a write quorum from the old configuration; this ensures that enough nodes are aware of the new configuration to ensure that any concurrent reconfiguration requests obtain the desired result.

There are two optimizations included in the protocol. First, if a node has already prepared a ballot as part of a prior reconfiguration, it can continue to use the same ballot for the new reconfiguration, without redoing the prepare phase. This means that if the same node initiates multiple reconfigurations, only the first reconfiguration has to perform the prepare phase. Second, the propose phase can terminate when *any* node, even if it is not the leader, discovers that an appropriate set of quorums has voted for the new configuration. If all the nodes in a quorum send their responses to the propose phase to all the nodes in the old configuration, then all the replicas can terminate the propose phase at the same time, immediately sending out propagate messages. Again, when any node receives a propagate response from enough nodes, it can terminate the propagate phase. This saves the reconfiguration one message delay. Together, these optimizations mean that when the same node is performing repeated reconfigurations, it only requires three message delays: the leader sending the propose message to the old configuration, the nodes in the old configuration sending the responses to the nodes in the old configuration, and the nodes in the old configuration sending a propagate message to the initiator, which can then terminate the reconfiguration.

4 Proof of Correctness (Atomic Consistency)

We now outline the safety proof of RDS, i.e., we show that the read and write operations are atomic (linearizable). We depend on two lemmas commonly used to show linearizability: Lemmas 13.10 and 13.16 in [21]. We use the tags of the operations to induce a partial ordering on operations that allows us to prove the key property necessary to guarantee atomicity: if π_1 is an operation that completes before π_2 begins, then the tag of π_1 is no larger than the tag of π_2; if π_2 is a write operation, the inequality is strict.

Ordering Configurations. Before we can reason about the consistency of read and write operations, we must show that nodes agree on the active configurations. For a reconfiguration replacing configuration c, we say that reconfiguration $\langle c, c' \rangle$ is *well-defined* if no node replaces configuration c with any configuration except c'. This is, essentially, showing that the consensus protocol successfully achieves agreement. The proof is an extension of the proof in [3] which shows that Paxos guarantees agreement, modified to incorporate optimizations in our algorithm and reconfiguration (for lack of space we omit the proof).

Theorem 1. *For all executions, there exists a sequence of configurations, $c_1, c_2, \ldots$, such that reconfiguration $\langle c_i, c_{i+1} \rangle$ is well-defined for all i.*

Ordering Operations. We now proceed to show that tags induce a valid ordering on the operations. If both operations "use" the same configuration, then this property is easy to see: operation π_1 propagates its tag to a write quorum, and π_2 discovers the tag when reading from a read quorum. The difficult case occurs when π_1 and π_2 use differing configurations. In this case, the reconfigurations propagate the tag from one configuration to the next.

We refer to the smallest tag at a node that replaces configuration c_ℓ with configuration $c_{\ell+1}$ as the "tag for configuration $c_{\ell+1}$." We can then easily conclude from this definition, along with a simple induction argument, that:

Invariant 2 *If some node i has configuration $c_\ell + 1$ in its set of active configurations, then its tag is at least as large as the tag for configuration $c_{\ell+1}$.*

This invariant allows us to conclude two facts about how information is propagated by reconfiguration operations: the tag of each configuration is no larger than the tag of the following configuration, and the tag of a read/write operation is no larger than the tag of a configuration in its set of active configurations. The next lemma requires showing how read and write operations propagate information *to* a reconfiguration operation:

Lemma 1. *If c_ℓ is the largest configuration in i's op-config set of operational configurations when **RW-Phase-2** completes, then the tag of the operation is no larger than the tag of configuration $c_{\ell+1}$.*

Proof. During the **RW-Phase-2**, the tag of the read or write operation is sent to a write quorum of the configuration c_ℓ. This quorum must intersect the read quorum during the **Recon-Phase-2** propagation phase of the reconfiguration that installs $c_{\ell+1}$. Let i' be a node in the intersection of the two quorums. If i' received the reconfiguration message prior to the read/write message, then node i would learn about configuration $c_{\ell+1}$. However we assumed that c_ℓ was the largest configuration in *op-config* at i at the end of the phase. Therefore we can conclude that the read/write message to i preceded the reconfiguration message, ensuring that the tag was transfered as required. □

Theorem 3. *For any execution, α, it is possible to determine a linearization of the operations.*

Proof. As discussed previously, we need to show that if operation π_1 precedes operation π_2, then the tag of π_1 is no larger than the tag of π_2, and if π_1 is a write operation, then the inequality is strict.

There are three cases to consider. First, assume π_1 and π_2 use the same configuration. Then the write quorum accessed during the propagate phase of π_1 intersects the read quorum accessed during the query phase of π_2, ensuring that the tag is propagated.

Second, assume that the *smallest* configuration accessed by π_1 in the propagate phase is larger than the *largest* configuration accessed by π_2 in the query phase. This case cannot occur. Let c_ℓ be the largest configuration accessed by π_2. Prior to π_1, some configuration installing configuration $c_{\ell+1}$ must occur. During the final phase **Recon-Phase-2** of the reconfiguration, a read quorum of configuration c_ℓ is notified of the new configuration. Therefore, during the query phase of π_2, the new configuration for $c_{\ell+1}$ would be discovered, contradicting our assumption.

Third, assume that the *largest* configuration c_ℓ accessed by π_1 in the propagate phase **RW-Phase-2** is smaller than the *smallest* configuration $c_{\ell'}$ accessed by π_2 in the query phase **RW-Phase-1**. Then, Lemma 1 shows that the tag of π_1 is no larger than the tag of c_ℓ; Invariant 2 shows that the tag of c_ℓ is no larger than the tag of $c_{\ell'}$ and that the tag of $c_{\ell'}$ is no larger than the tag of π_2. Together, these show the required relationship of the tags.

If π_1 skips the second phase, **RW-Phase-2**, then an earlier read or write must have performed a **RW-Phase-2** for the same tag, and the proof follows as before. $\square$

5 Conditional Performance Analysis

Here we examine the performance of RDS, focusing on the efficiency of reconfiguration and how the algorithm responds to instability in the network. To ensure that the algorithm makes progress in an otherwise asynchronous system, we make a series of assumptions about the network delays, the connectivity, and the failure patterns. In particular, we assume that, eventually, the network stabilizes and delivers messages with a delay of d. The main results in this section are as follows. (i) we show that the algorithm "stabilizes" within $e+2d$ time after the network stabilizes, where e is the time required for new nodes to fully join the system and notify old nodes about their existence. (By contrast, the original RAMBO algorithm [1] might take arbitrarily long to stabilize under these conditions.) (ii) we show that after the algorithm stabilizes, reconfiguration completes in $5d$ time; if a single node performs repeated reconfigurations, then after the first, each subsequent reconfiguration completes in $3d$ time. (iii) we show that after the algorithm stabilizes, reads and writes complete in $8d$ time, reads complete in $4d$ time if there is no interference from ongoing writes, and in $2d$ if no reconfiguration is pending.

Assumptions. Our goal is to model a system that becomes stable at some (unknown) point during the execution. Formally, let α be a (timed) execution and α' a finite prefix of α during which the network may be unreliable and unstable. After α' the network is reliable and delivers messages in a timely fashion.

We refer to $\ell time(\alpha')$ as the time of the last event of α'. In particular, we assume that following $\ell time(\alpha')$: (i) all local clocks progress at the same rate, (ii) messages are not lost and are received in at most d time, where d is a constant unknown to the algorithm, (iii) nodes respond to protocol messages as soon as they receive them and they broadcast messages every d time to all service participants, (iv) all enabled actions are processed with zero time passing on the local clock.

Generally, in quorum-based algorithms, the operations are guaranteed to terminate provided that at least one quorum does not fail. In constrast, for a reconfigurable quorum system we assume that at least one quorum does not fail prior to a successful reconfiguration replacing it. For example, in the case of majority quorums, this means that only a minority of nodes fail in between reconfigurations. Formally, we refer to this as *configuration-viability*: at least one read quorum and one write quorum from each installed configuration survive $4d$ after (i) the network stabilizes and (ii) a following successful reconfiguration operation.

We place some easily satisfied restrictions on reconfiguration. First, we assume that each node in a new configuration has completed the joining protocol at least time e prior to the configuration being proposed, for a fixed constant e. We call this *recon-readiness*. Second, we assume that after stabilization, reconfigurations are not too frequent: $5d$-*recon-spacing* implies that recons are at least $5d$ apart.

Also, after stabilization, we assume that nodes, once they have joined, learn about each other quickly, within time e. We refer to this as *join-connectivity*.

Finally, we assume that a leader election service chooses a single leader at time $\ell time(\alpha') + e$ and that it remains alive until the next leader is chosen and for a sufficiently long time for a reconfiguration to complete. For example, a leader may be chosen among the members of a configuration based on the value of an identifier.

Bounding Reconfiguration Delays. We now show that reconfiguration attempts complete within at most five message delays after the system stabilizes. Let ℓ be the node identified as the leader when the reconfiguration begins.

The following lemma describes a preliminary delay in reconfiguration when a non-leader node forwards the reconfiguration request to the leader.

Lemma 2. *Let the first* $\mathsf{recon}(c,c')$ *event at some active node* i*, where* $i \neq \ell$*, occur at time* t *and let* t' *be* $\max(\ell time(\alpha'),t)+e$*. Then, the leader* ℓ *starts the reconfiguration process at the latest at time* $t'+2d$*.*

Proof (sketch). When the $\mathsf{recon}(c,c')$ occurs at time t, one of two things happen: either the reconfiguration fails immediately, if c is not the current, unique, active configuration, or the recon request is forwarded to the leader. Observe that *join-connectivity* ensures that i knows the identity of the leader at time t', so no later than time $t'+d$, i sends a message to ℓ that includes reconfiguration request information. By time $t'+2d$ the leader receives message from i and starts the reconfiguration process. □

The next lemma implies that after some time following reconfiguration request, there is a communication round where all messages include the same ballot.

Lemma 3. *After time* $\ell time(\alpha')+e+2d$*,* ℓ *knows about the largest ballot in the system.*

Proof (sketch). Let b be the largest ballot in the system at time $\ell time(\alpha')+e+2d$, we show that ℓ knows it. We know that after $\ell time(\alpha')$, only ℓ can create a new ballot. Therefore ballot b must have been created before $\ell time(\alpha')$. Since ℓ is the leader at time $\ell time(\alpha')+e$, we know that ℓ has joined before time $\ell time(\alpha')$.

If ballot b still exists after $\ell time(\alpha')$ (the case we are interested in), then there are two possible scenarios. Either ballot b is conveyed by an in transit message or it exists an active node i aware of it at time $\ell time(\alpha')+e$. In the former case, gossip policy implies that the in transit message is received at time t, such that $\ell time(\alpha')+e<t<\ell time(\alpha')+e+d$. However, it might happen that ℓ does not receive it, if the sender ignored its identity at the time the send event occurred. Thus, at this time one of the receiver sends a message containing b to ℓ. Its receipt occurs before time $\ell time(\alpha')+e+2d$ and ℓ learns about b. In the latter case, by join-connectivity assumption at time $\ell time(\alpha')+e$, i knows about ℓ. Gossip policy implies i sends a message to ℓ before $\ell time(\alpha')+e+d$ and this message is received by ℓ before $\ell time(\alpha')+e+2d$, informing it of ballot b. □

Next theorem says that any reconfiguration completes in at most $5d$ time, following the system stabilization. The proof is straightforward from the code and is omitted for lack of space. In Theorem 5 we show that when the leader node has successfully completed the previous reconfiguration request then it is possible for the subsequent reconfiguration to complete in at most $3d$.

Theorem 4. *Assume that ℓ starts the reconfiguration process initiated by* $\mathsf{recon}(c,c')$ *at time $t \geq \ell time(\alpha') + e + 2d$. Then the corresponding reconfiguration completes no later than $t + 5d$.*

Theorem 5. *Let ℓ be the leader node that successfully conducted the reconfiguration process from c to c'. Assume that ℓ starts a new reconfiguration process from c' to c'' at time $t \geq \ell time(\alpha') + e + 2d$. Then the corresponding reconfiguration from c' to c'' completes at the latest at time $t + 3d$.*

Proof (sketch). By *configuration-viability*, at least one read and one write quorums of c' are active. By Lemma 3, ℓ knows the largest ballot in the system at the beginning of the new reconfiguration. This means that ℓ may keep its ballot and start from **Recon-Phase-2a** (since it has previously executed **Recon-Phase-1b**). Hence only a single message exchange in **Recon-Phase-2a**/**Recon-Phase-2b** and a single broadcast following **Recon-Phase-3a** take place. Therefore, the last phase of Paxos occurs at time $t + 3d$.

Bounding Read-Write Delays. In this section we present bounds on the duration of read/write operations under assumptions stated in the previous section. Recall from Section 3 that both the read and the write operations are conducted in two phases, first the query phase and second the propagate phase. We begin by first showing that each phase requires at least $4d$ time. However, if the operation is a read operation and no reconfiguration and no write propagation phase is concurrent, then it is possible for this operation to terminate in only $2d$ – see proof of Lemma 4. The final result is a general bound of $8d$ on the duration of any read/write operation.

Lemma 4. *Consider a single phase of a read or a write operation initiated at node i at time t, where i is a node that joined the system at time* $\max(t - e - 2d, \ell time(\alpha'))$. *Then this phase completes at the latest at time* $\max(t, \ell time(\alpha') + e + 2d) + 4d$.

Proof. Let c_k be the largest configuration in any active node's *op-configs* set, at time $t - 2d$. By the *configuration-viability* assumption, at least one read and at least one write quorum of c_k are active for the interval of $4d$ after c_{k+1} is installed. By the *join-connectivity* and the fact that i has joined at time $\max(t - e - 2d, \ell time(\alpha'))$, i is aware of all active members of c_k by the time $\max(t - 2d, \ell time(\alpha') + e)$.

Next, by the timing of messages we know that within d time a message is sent from each active members of c_k to i. Hence, at time $\max(t, \ell time(\alpha') + e + 2d)$ node i becomes aware of c_k, i.e. $c_k \in$ *op-configs*.

At d time later, messages from phase **RW-Phase-1a** or **RW-Phase-2a** are received and **RW-Phase-1b** or **RW-Phase-2b** starts. Consequently, no later than $\max(t, \ell time(\alpha') + e + 2d) + 2d$, the second message of **RW-Phase-1** or **RW-Phase-2** is received.

Now observe that configuration might occur in parallel, therefore it is possible that a new configuration is added to the *op-configs* set during **RW-Phase-1** or **RW-Phase-2**.

Discovery of new configurations results in the phase being restarted, hence completing at time $\max(t, \ell time(\alpha') + e + 2d) + 4d$. By *recon-spacing* assumption no more than one configuration is discovered before the phase completes. □

Theorem 6. *Consider a read operation that starts at node i at time t:*

1. *If no write propagation is pending at any node and no reconfiguration is ongoing, then it completes at time* $\max(t, \ell time(\alpha') + e + 2d) + 2d$.
2. *If no write propagation is pending, then it completes at time* $\max(t, \ell time(\alpha') + e + 2d) + 8d$.

Consider a write operation that starts at node i at time t. Then it completes at time $\max(t, \ell time(\alpha') + e + 2d) + 8d$.

Proof. At the end of the **RW-Phase-1**, if the operation is a write, then a new non confirmed tag is set. If the operation is a read, the tag is the highest received one. This tag was maintained by a member of the read queried quorum, and it is confirmed only if the phase that propagated it to this member has completed. From this point, if the tag is not confirmed, then in any operation the fix-point of propagation phase **RW-Phase-2** has to be reached. But, if the tag is already confirmed then the read operation can terminate directly at the end of the first phase. By Lemma 4, this occurs at the latest at time $\max(t, \ell time(\alpha') + e + 2d) + 4d$; or at time $\max(t, \ell time(\alpha') + e + 2d) + 2d$ if no reconfiguration is concurrent. Likewise by Lemma 4, the **RW-Phase-2** fix-point is reached in at most $4d$ time. That is, any operation terminates by confirming its tag no later than $\max(t, \ell time(\alpha') + e + 2d) + 8d$. □

6 Experimental Results

We implemented the new algorithm based on the existing RAMBO codebase [7] on a network of workstations. The primary goal of our experiments was to gauge the cost introduced by reconfiguration. When reconfiguration is unnecessary, there are simpler and more efficient algorithms to implement a replicated DSM. Our goal is to achieve performance similar to the simpler algorithms while using reconfiguration to tolerate dynamic changes.

To this end, we designed three series of experiments where the performance of RDS is compared against the performance of an atomic memory service which has no reconfiguration capability — essentially the algorithm of Attiya, Bar Noy, and Dolev [13] (the "ABD protocol"). In this section we briefly describe these implementations and present our initial experimental results. The results primarily illustrate the impact of reconfiguration on the performance of read and write operations.

For the implementation we manually translated the IOA specification (from the appendix) into Java code. To mitigate the introduction of errors during translation, the implementers followed a set of precise rules to guide the derivation of Java code [22]. The target platform is a cluster of eleven machines running Linux. The machines are various Pentium processors up to 900 MHz interconnected via a 100 Mbps Ethernet switch.

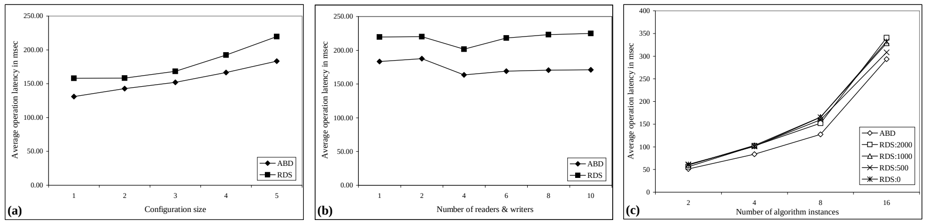

Fig. 3. Average operation latency: (a) as size of configurations changes, (b) as number of nodes performing read/write operations changes, and (c) as the reconfiguration and the number of participants changes.

Each instance of the algorithm uses a single socket to receive messages over TCP/IP, and maintains a list of open, outgoing connections to the other participants of the service. The nondeterminism of the I/O Automata model is resolved by scheduling locally controlled actions in a round-robin fashion. The ABD and RDS algorithm share parts of the code unrelated to reconfiguration, in particular that related to joining the system and accessing quorums. As a result, performance differences directly indicate the costs of reconfiguration. While these experiments are effective at demonstrating comparative costs, actual latencies most likely have little reflection on the operation costs in a fully-optimized implementation.

Experiment (a). In the first experiment, we examine how the RDS algorithm responds to different size configurations (and hence different levels of fault-tolerance). We measure the average operation latency while varying the size of the configurations. Results are depicted in Figure 3(a). In all experiments, we use configurations with majority quorums. We designate a single machine to continuously perform read and write operations and compute average operation latency for different size configurations, ranging from 1 to 5. In the tests involving the RDS algorithm, we chose a separate machine to continuously perform reconfiguration of the system – when one reconfiguration request successfully terminates another is immediately submitted.

Experiment (b). In the second set of experiments, we test how the RDS algorithm responds to varying load. Figure 3(b) presents results of the second experiment, where we compute the average operation latency for a fixed-size configuration of five members, varying the number of nodes performing read/write operations changes from 1 to 10. Again, in the experiments involving RDS algorithm a single machine is designated to reconfigure the system. Since we only have eleven machines to our disposal, nodes that are members of configurations also perform read/write operations.

Experiment (c). In the last experiment we test the effects of reconfiguration frequency. Two nodes continuously perform read and write operations, and the experiments were run varying the number of instances of the algorithm. Results of this test are depicted in Figure 3(c). For each of the sample points on the x-axis, the size of configuration used is half of the algorithm instances. As in the previous experiments, a single node is dedicated to reconfigure the system. However, here we insert a delay between the successful termination of a reconfiguration request and the submission of another. The delays used are 0, 500, 1000, and 2000 milliseconds. Since we only have eleven machines to our disposal, in the experiment involving 16 algorithm instances, some of the machines run two instances of the algorithm.

Interpretation. We begin with the obvious. In all three series of experiments, the latency of read/write operations for RDS is competitive with that of the simpler ABD algorithm. Also, the frequency of reconfiguration has little effect on the operation latency. These observations lead us to conclude that the increased cost of reconfiguration is only modest.

This is consistent with the theoretical operation of the algorithm. It is only when a reconfiguration exactly intersects an operation in a particularly bad way that operations are delayed. This is unlikely to occur, and hence most read/write operations suffer only a modest delay.

Also, note that the messages that are generated during reconfiguration, and read and write operations, include replica information as well as the reconfiguration information. Since the actions are scheduled using a round-robin method, it is likely that in some instances a single communication phase might contribute to the termination of both the read/write and the reconfiguration operation. Hence, we suspect that the dual functionality of messages helps to keep the system latency low.

A final observation is that the latency does grow with the size of the configuration and the number of participating nodes. Both of these require increased communication, and result in larger delays in the underlying network when many nodes try simultaneously to broadcast data to all others. Some of this increase can be mitigated by using an improved multicast implementation; some can be mitigated by choosing quorums optimized specifically for read or write operations.

7 Conclusion

We have presented RDS, a new distributed algorithm for implementing a reconfigurable consistent shared memory in dynamic, asynchronous networks. Prior solutions (e.g., [1, 2]) used a separate new configuration selection service that did not incorporate the removal of obsolete configurations. This resulted in longer delays between the time of new-configuration installation and old configuration removal, hence requiring configurations to remain viable for longer periods of time and decreasing algorithm's resilience to failures. In this work we capitalized on the fact that RAMBO and Paxos solve two different problems using a similar mechanism, namely round-trip communication phases involving sets of quorums. This observation led to the development of RDS that allows rapid reconfiguration and removal of obsolete configurations, hence reducing the window of vulnerability. Finally, our experiments show that reconfiguration is inexpensive, since performance of our algorithm closely mimics that of an algorithm that has no reconfiguration functionality. However, our experiments are limited to a small number of machines and a controlled lab setting. Therefore, as future work we would like to extend the experimental study to a wide area network where many machines participate thereby allowing us to capture a more realistic behavior of this algorithm for arbitrary configuration sizes and network delays.

References

1. Lynch, N., Shvartsman, A.: RAMBO: A reconfigurable atomic memory service for dynamic networks. In: Proc. of 16th Int-l Symposium on Distributed Computing. (2002) 173–190
2. Gilbert, S., Lynch, N., Shvartsman, A.: RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In: Proc. of International Conference on Dependable Systems and Networks. (2003) 259–268
3. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16(2)** (1998) 133–169
4. Lamport, L.: Paxos made simple. ACM SIGACT News (Distributed Computing Column) **32**(4) (2001) 18–25
5. Lampson, B.W.: How to build a highly available system using consensus. In: WDAG '96: Proceedings of the 10th International Workshop on Distributed Algorithms, London, UK, Springer-Verlag (1996) 1–17
6. Boichat, R., Dutta, P., Frolund, S., Guerraoui, R.: Reconstructing paxos. SIGACT News **34**(2) (2003) 42–57
7. Georgiou, C., Musial, P., Shvartsman, A.: Long-lived RAMBO: Trading knowledge for communication. In: Proc. of 11'th Colloquium on Structural Information and Communication Complexity, Springer (2004) 185–196
8. van der Meyden, R., Moses, Y.: Top-down considerations on distributed systems. In: 12th Int. Symp. on Distributed Computing, DISC'98. (1998) 16–19
9. Gifford, D.K.: Weighted voting for replicated data. In: Proceedings of the seventh ACM symposium on Operating systems principles, ACM Press (1979) 150–162
10. Thomas, R.H.: A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. Database Syst. **4**(2) (1979) 180–209
11. Upfal, E., Wigderson, A.: How to share memory in a distributed system. Journal of the ACM **34(1)** (1987) 116–127
12. Awerbuch, B., Vitanyi, P.: Atomic shared register access by asynchronous hardware. In: Proc. of 27th IEEE Symposium on Foundations of Computer Science. (1986) 233–243
13. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1) (1995) 124–142
14. Englert, B., Shvartsman, A.: Graceful quorum reconfiguration in a robust emulation of shared memory. In: Proc. of Int-l Conference on Distributed Computer Systems. (2000) 454–463
15. Lynch, N., Shvartsman, A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: Proc. of 27th Int-l Symp. on Fault-Tolerant Comp. (1997) 272–281
16. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. In: Proc. of the 11th ACM Symposium on Operating systems principles, ACM Press (1987) 123–138
17. : Special issue on group communication services. Communications of the ACM **39(4)** (1996)
18. Albrecht, J., Yasushi, S.: RAMBO for dummies. Technical report, HP Labs (2005)
19. Saito, Y., Frolund, S., Veitch, A.C., Merchant, A., Spence, S.: Fab: building distributed enterprise disk arrays from commodity components. In: ASPLOS04. (2004) 48–58
20. Georgiou, C., Musial, P., Shvartsman, A.A.: Developing a consistent domain-oriented distributed object service. In: Proceedings of the 4th IEEE International Symposium on Network Computing and Applications, NCA 2005, Cambridge, MA, USA (2005)
21. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers (1996)
22. Musial, P., Shvartsman, A.: Implementing a reconfigurable atomic memory service for dynamic networks. In: Proc. of 18'th International Parallel and Distributed Symposium — FTPDS WS. (2004) 208b