J. Parallel Distrib. Comput. ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Reconfigurable distributed storage for dynamic networks☆

Gregory Chockler[a], Seth Gilbert[b], Vincent Gramoli[b,c,*], Peter M. Musial[d], Alex A. Shvartsman[d,e]

[a] IBM Haifa Labs, Israel
[b] EPFL LPD, Switzerland
[c] University of Neuchâtel, Switzerland
[d] Department of Comp. Sci. and Eng., University of Connecticut, United States
[e] MIT CSAIL, United States

ARTICLE INFO

Article history:
Received 11 December 2007
Received in revised form
21 May 2008
Accepted 20 July 2008
Available online xxxx

Keywords:
Distributed algorithms
Reconfiguration
Atomic objects
Performance

ABSTRACT

This paper presents a new algorithm 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 the processing nodes, message delays, and message loss. The algorithm incorporates a classic quorum-based algorithm for read/write operations, and an optimized consensus protocol, based on Fast Paxos for reconfiguration, and 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 resulting algorithm tolerates dynamism. We formally prove our algorithm to be correct, we present its performance and compare it to existing reconfigurable memories, and we evaluate experimentally the cost of its reconfiguration mechanism.

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 or regenerate data fragments, 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 networks are 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) [21, 11], supports multi-reader/multi-writer atomic objects in dynamic settings. The reconfiguration service is loosely coupled with the read/write service. This allows 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. The delayed removal of obsolete configurations can slow down reconfiguration, lead to multiple extant configurations, and require stronger fault-tolerance assumptions.

The contribution of this work is the specification of a new distributed memory service that tightly integrates the two stages of reconfiguration. Our approach translates into a reduced reconfiguration cost in terms of latency and a relaxation of fault-tolerance requirements on the installed configurations. Moreover, we provide a bound on the time during which each configuration

☆ The conference version of this paper has previously appeared in the proceedings of the 9th International Conference on Principles of Distributed Systems and parts of this work have recently appeared in a thesis [V. Gramoli, Distributed shared memory for large-scale dynamic systems, Ph.D. in Computer Science, INRIA - Université de Rennes 1, November, 2007]. This work is supported in part by the NSF Grants 0311368 and 0121277.

* Corresponding address: EPFL-IC-LPD, Station 14, CH-1015 Lausanne, Switzerland.
E-mail address: vincent.gramoli@epfl.ch (V. Gramoli).

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.07.007

needs to remain active, without impacting the efficiency of the data access operations. The developments presented here are an example of a trade-off between the simplicity of a loosely coupled reconfiguration protocols, as in [21,11] and the fault-tolerance properties that tightly coupled reconfiguration protocols, like the current work, achieve.

1.1. Contributions

In this paper we present a new distributed algorithm, named *Reconfigurable Distributed Storage* (RDS). As the RAMBO algorithms [21,11], RDS implements atomic (linearizable) object semantics, where consistency of data is maintained via use of *configurations* consisting of *quorums* of network locations. Depending on the properties of the quorums, configurations are capable of sustaining small and transient changes and remain fully usable at the same time. Read and write operations consist of two phases, where each accesses 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 from RAMBO and RAMBO II. 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 Fast Paxos [5,18], an optimized version of Paxos [16,17,19]. The protocol for removing obsolete configurations is a two-phase protocol, involving quorums of the old and the new configurations.

In summary, we present a new algorithm, RDS, that implements a survivable atomic memory service. We formally show that the new algorithm correctly implements atomic objects in all executions involving asynchrony, processor stop-failures, and message loss. We present the time complexity of the algorithm when message delays become bounded. More precisely, our upper-bound on operation latency requires that at most one reconfiguration success occurs every 5 message delays, and our upper-bound on reconfiguration latency requires that a leader is eventually elected and at least one read-quorum and one write-quorum remain active during 4 message delays. Furthermore, we compare the latencies obtained and show that RDS supersedes other existing reconfigurable memories. Finally, we present the 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 the reconfiguration process, and can terminate regardless of a success or a failure of the ongoing reconfiguration. However, network instability can postpone termination of the read and write operations.
- *Fully flexible reconfiguration:* The algorithm imposes no dependencies between the quorum configurations selected for installation.
- *Fast reconfiguration:* The reconfiguration uses a leader-based consensus protocol, similar to Fast Paxos [5,18]; when the leader is stable, reconfigurations are very fast: three network delays. Since halting consensus requires at least three network delays, reconfiguration does not add any overhead and thus reaches time optimality.
- *Fast read operations:* Read operations require only two message delays when no write operations interfere with it. Consequently, their time complexity is optimal [6].
- *No recovery need:* Our solution does not need to recover after network instability by cleaning up obsolete quorum configurations. Specifically, unlike the prior RAMBO algorithms [21,11] that may generate an arbitrarily long backlog of old configurations, there is never more than one old configuration present in the system at a time, diminishing message complexity accordingly. More importantly, RDS tolerates the failures of all old configurations but the last one.

Our reconfiguration algorithm can be viewed as an example of protocol composition advocated by van der Meyden and Moses [29]. 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.

1.2. Background

Several approaches have been used to implement consistent data in (static) distributed systems. Starting with the work of Gifford [10] and Thomas [27], many algorithms have used collections of intersecting sets of objects replicas (such as quorums) to solve the consistency problem. Upfal and Wigderson [28] use majority sets of readers and writers to emulate shared memory. Vitányi and Awerbuch [3] use matrices of registers where the rows and the columns are written and respectively read by specific nodes. Attiya, Bar-Noy and Dolev [2] use majorities of nodes to implement shared objects in static message passing systems. Extensions for limited reconfiguration of quorum systems have also been explored [7,22] and the recent timed quorum systems [13,15] provide only probabilistic consistency.

Virtually synchronous services [4], and group communication services (GCS) in general [26], can also be used to implement consistent data services, e.g., by implementing a global totally ordered broadcast. While the universe of nodes 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 [21,11,8], allow reads and writes to make progress during reconfiguration and can benefit from grouping multiple objects into domains as described in [9].

RDS improves on these latter solutions [21,11,8] by using a more efficient reconfiguration protocol that makes it more fault tolerant. Finally, reconfigurable storage algorithms are finding their way into practical implementations [1,25]. The new algorithm presented here has the potential of making further impact on system development.

1.3. Document structure

Section 2 defines the model of computation. Section 3 presents some key ideas to obtain an efficient read/write memory for dynamic settings. We present the algorithm in Section 4. In Section 5 we present the correctness proofs. In Section 6 we present conditional performance analysis of the algorithm. Section 7 compares explicitely the complexity of RDS to the complexity of the RAMBO algorithms. Section 8 contains experimental results about operation latency. The conclusions are in Section 9.

2. System model and definitions

Here, we present the system model and give the prerequisite definitions.

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

2.1. Model

We use a message-passing model with asynchronous processors (also called nodes), that have unique identifiers (the set of node identifiers need not be finite). Nodes may crash (stop-fail). Nodes communicate via point-to-point asynchronous unreliable channels. More precisely, messages can be lost, duplicated, and reordered, but new messages can not be created by the link. In normal operation, any node can send a message to any other node. 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 about the performance of the underlying network. In particular, we assume the presence of a leader election service that stabilizes when failures stop and message delays are bounded. (This leader must be a node that has already joined the system, but does not necessarily need to be part of any configuration.) This service can be implemented deterministically, for example nodes periodically send the smallest node identifier they have received so far to other nodes: the nodes that has never received a smaller identifier than their own can decide to be leader; after some time there will be a single leader. In addition, we assume that eventually (at some unknown point) the network stabilizes, becoming synchronous and delivering messages in bounded (but unknown) time. We also assume that the rate of reconfiguration after stabilization is not too high, 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 6.

2.2. Data types

The set of all node identifiers is denoted as $I \subset \mathbb{N}$. This is a set of network locations where the RDS service can be executed.

The RDS algorithm is specified for a single object. Let X be the set of all data objects, and RDS_x, for $x \in X$, denotes an automaton that implements atomic object x. A complete memory system is created by composing the individual RDS automata. The composition of the RDS automata implements an atomic memory, since atomicity is preserved under composition. From this point on, we fix one particular object $x \in X$ and omit the implicit subscript x. We refer to V as the set of all possible values for object x. With each object x we associate a set T of tags, where each tag is a pair of counter and node identifier $-T \subset \mathbb{N} \times I$.

A *configuration* $c \in C$ consists of three components: (i) *members*(c), a finite set of node ids, (ii) *read-quorums*(c), a set of quorums, and (iii) *write-quorums*(c), a set of quorums, where each quorum is a subset of *members*(c). That is, C is the set of all tuples representing a different configuration c. We require that the read quorums and write quorums of a common configuration intersect: formally, for every $R \in$ *read-quorums*(c) and $W \in$ *write-quorums*(c), the intersection $R \cap W \neq \emptyset$. Neither two read quorums nor two write quorums need to intersect. Note that a node participating in the service does not have to belong to any configuration.

The following are the additional data types and functions that help to describe the way nodes handle and aggregate configuration information. For this purpose, we use the not-yet-created ($\perp$) and removed ($\pm$) symbols. and we partially order the elements of $C \cup \{\perp, \pm\}$ such that for any $c \in C$, $\perp < c < \pm$. The data types and functions follow:

- *CMap*, the set of *configuration maps*, defined as the set of mappings from integer indices $\mathbb{N}$ to $C \cup \{\perp, \pm\}$.
- *update*, a binary function on $C \cup \{\perp, \pm\}$, defined by $update(c, c') = \max(c, c')$ if c and c' are comparable (in the partial ordering of $C \cup \{\perp, \pm\}$), $update(c, c') = c$ otherwise.
- *extend*, a binary function on $C \cup \{\perp, \pm\}$, defined by $extend(c, c') = c'$ if $c = \perp$ and $c' \in C$, and $extend(c, c') = c$ otherwise.
- *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* if and only if $truncate(cm) = cm$.

The *update* and *extend* operators are extended element-wise to binary operations on *CMap*.

3. Overview of the main ideas

In this section, we present an overview of the main ideas that underlie the RDS algorithm. In Section 4, we present the algorithm in more detail. Throughout this section, we discuss the implementation of a single memory location x; each of the protocols presented supports read and write operations on x.

We begin in Section 3.1 by reviewing a simple algorithm for implementing a read/write shared memory in a *static* system, i.e., one in which there is no reconfiguration or change in membership. Then, in Section 3.2, we review a *reconfigurable* atomic memory, that consists of two decoupled components: a read/write component (similar to that described in Section 3.1), and a reconfiguration component, based on Paxos [16,17,19]. Finally, in Section 3.3, we describe briefly how the RDS protocol improves and merges these two components, resulting in a more efficient integrated protocol.

3.1. Static read/write memory

In this section, we review a well-known protocol for implementing read/write memory in a static distributed system. This protocol (also known as *ABD*) was originally presented by Attiya, Bar-Noy, and Dolev [2]. (For the purposes of presentation, we adapt it to the terminology used in this paper.)

The ABD protocol relies on a single configuration, that is, a single set of members, read-quorums, and write-quorums. (It does not support any form of reconfiguration.) Each member of the configuration maintains a replica of memory location x, as well as a tag that contains some meta-data about the most recent write operation. Each tag is a pair consisting of a sequence number and a process identifier.

Each read and write operation consists of two phases: (1) a *query* phase, in which the initiator collects information from a read-quorum, and (2) a *propagate* phase, in which information is sent to a write-quorum.

Consider, for example, a write operation initiated by node i that attempts to write value v to location x. First, the initiator i contacts a read-quorum, collecting the set of tags and values returned by each quorum member. The initiator then selects the tag with the largest sequence number, say, s, and creates a new tag $\langle s + 1, i \rangle$. The initiator then sends the new value v and the new tag $\langle s + 1, i \rangle$ to a write-quorum.

A read operation proceeds in a similar manner. The initiator contacts a read-quorum, collecting the set of tags and values returned by each quorum member. It then selects the value v associated with the largest tag t (where tags are considered in lexicographic order). Before returning the value v, it sends the value v and the tag t to a write-quorum.

The key observation is as follows: consider some operation π_2 that begins after an earlier operation π_1 completes; then the write-quorum contacted by π_2 in the propagate phase intersects with

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

the read-quorum contacted by π_1 in the query phase, and hence the second operation discovers a tag at least as large as the first operation. If π_2 is a read operation, we can then conclude that it returns a value at least as recent as the first operation.

3.2. Dynamic read/write memory

The RAMBO algorithms [21,11] introduce the possibility of *reconfiguration*, that is, choosing a new configuration with a new set of members, read-quorums, and write-quorums. RAMBO consists of two main components: (1) a Read–Write component that extends the ABD protocol, supporting read and write operations; and (2) a Reconfiguration component that relies on Paxos [16,17,19], a consensus protocol, to agree on new configurations. These two components are decoupled, and operate (almost) independently.

3.2.1. The read–write component

The Read–Write component of RAMBO is designed to operated in the presence of multiple configurations. Initially, there is only one configuration. During the execution, the Reconfiguration component may produce additional new configurations. Thus, at any given point, there may be more than one active configuration. At the same time, a *garbage-collection* mechanism proceeds to remove old configurations. If there is a sufficiently long period of time with no further reconfigurations, eventually there will again only be one active configuration.

Read and write operations proceed as in the ABD protocol, in that each operation consists of two phases, a query phase and a propagation phase. Each query phase accesses one (or more) read-quorums, while each write operation accesses one (or more) write-quorums. Unlike ABD, however, each phase may need to access quorums from more than one configuration. In fact, each phase accesses one quorum from *each* active configuration.

The garbage-collection operation proceeds much like the read and write operations. It first performs a query phase, collecting tag and value information from the configuration to be removed, that is, from a read-quorum and a write-quorum of the old configuration. It then propagates that information to the new configuration, i.e., to a write-quorum of the new configuration. At this point, it is safe to remove the old configuration.

3.2.2. The reconfiguration component

The Reconfiguration component is designed to produce new configurations. Specifically, it receives, as input, proposals for new configurations, and produces, as output, a sequence of configurations, with the guarantee that each node in the system will learn an identical sequence of configurations. In fact, the heart of the Reconfiguration component is a consensus protocol, in which all the nodes attempt to agree on the sequence of configurations.

In more detail, the Reconfiguration component consists of a sequence of instances of consensus, $P_1, P_2, \ldots$. Each node presents as input to instance P_k a proposal for the kth configuration c_k. Instance P_k then uses the quorum-system from configuration c_{k-1} to agree on the new configuration c_k, which is then output by the Reconfiguration component.

For the purpose of this paper, we consider the case where each consensus instance P_k is instantiated using the Paxos agreement protocol [16,17,19].

In brief, Paxos works as follows. (1) *Preliminaries:* First, a leader is elected, and all the proposals are sent to the leader. (2) *Prepare phase:* Next, the leader proceeds to choose a ballot number b (larger than any prior ballot number known to the leader) and to send this ballot-number to a read-quorum; this is referred to as the prepare phase. Each replica that receives a prepare message responds only if the ballot number b is in fact larger than any previously received ballot number. In that case, it responds by sending back any proposals that it has previously voted on. The leader then chooses a proposal from those returned by the write-quorum; specifically, it chooses the one with the highest ballot number. If there is no such proposal that has already been voted on, then it uses its own proposal. (3) *Propose phase:* The leader then sends a message to a write-quorum including the chosen proposal and the ballot number. Each replica that receives such a proposal votes for that proposal if it has still seen no ballot number larger than b. If the leader receives votes from a write-quorum, then it concludes that its proposal has been accepted and sends a message to everyone indicating the decision.

The key observation that implies the correctness of Paxos is as follows: notice that if a leader eventually decides some value, then there is some write-quorum that has voted for it; consider a subsequent leader that may try to render a different decision; during the prepare phase it will access a read-quorum, and necessarily learn about the proposal that has already been voted on. Thus every later proposal will be identical to the already decided proposal, ensuring that there is at most one decision. See [16,17,19] for more details.

3.3. RDS overview

The key insight in this paper is that both the Read–Write component and the Paxos component of RAMBO operate in the same manner, and hence they can be combined. Thus, as in both ABD and RAMBO, each member of an active configuration stores a replica of location x, along with a tag consisting of a sequence number s and a node identifier. Similarly as before, read and write operations rely on a query phase and a propagation phase, each of which accesses appropriate quorums from all active configurations, but in RDS some operations consist only of a query phase.

Unlike RAMBO algorithms, the reconfiguration process does two steps simultaneously: it both decides on the new configuration, and it removes the old configuration. Reconfiguration from old configuration c to new configuration c' consists of the following steps:

Preliminaries: First, the request is forwarded to a possible leader ℓ. If the leader has already completed Phase 1 for some ballot b, then it can skip Phase 1, and use this ballot in Phase 2. Otherwise, the leader performs Phase 1.

Phase 1: Leader ℓ chooses a unique ballot number b larger than any previously used ballot and sends $\langle Recon1a, b\rangle$ messages to a read quorum of configuration c (the old configuration). 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 $\langle Recon1b, b, configs, b'', c''\rangle$ where *configs* is the set of active configurations and b'' and c'' represent the largest ballot and configuration which j has voted should replace configuration c.

Phase 2: 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 contain empty signifiers for the last two parameters, then v is c'; otherwise, v is the configuration with the largest ballot received in the prepare phase. 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, where *value* and *tag* correspond to the current object value and its version that j has locally.

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

Input:	**Output:**	**Internal:**
$join(W)_i$	$join\text{-}ack_i$	$query\text{-}fix_i$
$read_i$	$read\text{-}ack(v)_i$	$prop\text{-}fix_i$
$write(v)_i$	$write\text{-}ack_i$	$prepare(b)_i$
$recon(c, c')_i$	$recon\text{-}ack(r)_i$	$prepare\text{-}done(b)_i$
$recv(m)_i$	$send(m)_i$	$propose(b)_i$
$fail_i$		$propose\text{-}done(b)_i$
$leader(r)_i$		$recon\text{-}done(\ell)_i$

Fig. 1. Signature.

Phase 3: 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 configuration c' to the set of active configurations. It then sends a $\langle Recon3a, c, c', tag, value\rangle$ message to a read quorum and a write quorum of configuration c. 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.

4. RDS algorithm

In this section, we present the RDS service and its specification. The RDS algorithm is formally stated using the Input/Output Automata notation [20]. 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 [9] 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 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 set of replicas is changing, there may be two active configurations. Throughout the algorithm, each node maintains a set of *active configurations*. A new configuration is added to the set during a reconfiguration, and the old one is removed at the end of a reconfiguration.

4.1. Signature

The external specification of the algorithm appears in Fig. 1. Before issuing any operations, a client instructs the node to join the system, providing the algorithm with a set of "seed" nodes already in the system. When the algorithm succeeds in contacting nodes already in the system, it returns a join-ack. A client can then choose to initiate a read or write operation, which result, respectively, in read-ack and write-ack responses. A client can initiate a reconfiguration, recon, resulting in a recon-ack. The network sends and recvs messages, and the node may be caused to fail. Finally, a leader election service may occasionally notify a node as to whether it is currently the leader.

4.2. State

The state of the algorithm is described in Fig. 2. The $value \in V$ of node i indicates the value of the object from the standpoint of i. A $tag \in T$ is maintained by each node as a unique pair of *counter* and *id*. The *counter* denotes the version of the value of the object from a local point-of-view, while the *id* is the node identifier and serves as a tie-breaker, when two nodes have the same counter for two different values. The value and the tag are simultaneously sent and updated when a larger tag is discovered, or when a write operation occurs.

The *status* of node i expresses the current state of i. A node may participate fully in the algorithm only if its status is active. The set of identifiers of nodes known to i to have joined the service is maintained locally in a set called *world*. Each processor maintains a list of configurations in a configuration map. A configuration map is denoted $cmap \in CMap$, a mapping from integer indices to $C \cup \{\bot, \pm\}$, and initially maps every integer, except 0, to $\bot$. The index 0 is mapped to the default configuration c_0 that is used at the beginning of the algorithm. This default configuration can be arbitrarily set by the designer of the application depending on its needs: e.g., since the system is reconfigurable, the default configuration can be chosen as a single node known to be reliable a sufficiently long period of time for the system to bootstrap. The configuration map tracks which configurations are active, which have not yet been created, indicated by $\bot$, and which have already been removed, indicated by $\pm$. The total ordering on configurations determined by the reconfiguration ensures that all nodes agree on which configuration is stored in each position in *cmap*. We define $c(k)$ to be the configuration associated with index k.

Read and write operations are divided into phases; in each phase a node exchanges information with all the replicas in some set of quorums. Each phase is initiated by some node that we refer to as the *phase initiator*. When a new phase starts, the *pnum*1 field records the corresponding phase number, allowing the client to determine which responses correspond to its phase. The *pnum* 2 field maps an identifier j to an integer $pnum2(j)_i$ indicating that i has heard about the $pnum2(j)_i^{th}$ phase of node j. The three records *op*, *pxs*, and *ballot* store the information about read/write operations, reconfiguration, and ballots used in reconfiguration, respectively. We describe their subfields in the following:

- The record *op* is used to store information about the current phase of an ongoing read or write operation. The $op.cmap \in CMap$ subfield records the configuration map associated with a read/write operation. This consists of the node's *cmap* when a phase begins. It is augmented by any new configuration discovered during the phase in the case of a read or write operation. A phase completes when the initiator has exchanged information with quorums from every valid configuration in *op.cmap*. The *op.pnum* subfield records the read or write phase number when the phase begins, allowing the initiator to determine which responses correspond to the phase. The *op.acc* subfield records which nodes from which quorums have responded during the current phase.
- The record *pxs* stores information about the paxos subprotocol. It is used as soon as a reconfiguration request has been received. The *pxs.pnum* subfield records the reconfiguration phase number, the *pxs.phase* indicates if the current phase is idle, prepare, propose, or propagate. The *pxs.conf-index* subfield is the index of *cmap* for the last installed configuration, while the *pxs.old-conf* subfield is the last installed configuration. Therefore, *pxs.conf-index* + 1 represents the index of *cmap* where the new configuration, denoted by the subfield *pxs.conf*,

1 *value* $\in V$, initially 0
2 *tag* $\in T$, a tag containing
3 *counter* $\in \mathbb{N}$, *initially* 0
4 *id* $\in I$, *initially* i
5 *status* $\in$ {idle,joining,active,failed}, initially idle
6 *world*, a finite subset of I, initially $\emptyset$
7 *cmap* $\in$ *CMap*, initially c_0 at index 0 and $\perp$ elsewhere
8 *pnum1* $\in \mathbb{N}$, initially 0
9 *pnum2*, a mapping from $I \to \mathbb{N}$, initially mapping all to 0
10 *isLeader* $\in$ {true,false}, initially false
11 *confirmed*, a set of tags, initially $\emptyset$
12 *failed* $\in$ {true,false}, initially false
13
14 *op*, a record with fields:
15 *type* $\in$ {read,write}, initially $\perp$
16 *phase* $\in$ {idle,query,prop,done} initially idle
17 *pnum* $\in \mathbb{N}$, initially 0
18 *cmap* $\in$ *CMap* initially c_0 at index 0 and $\perp$ elsewhere
19 *acc*, a finite subset of I, initially $\emptyset$
20 *tag* $\in T$, initially $\langle 0, i\rangle$
21 *value* $\in V$, initially 0
22 *pxs*, a record with fields:
23 *pnum* $\in \mathbb{N}$, initially 0
24 *phase* $\in$ {idle,prepare,propose,propagate}, initially idle
25 *conf-index* $\in \mathbb{N}$, initially 0
26 *old-conf* $\in C$, initially $\perp$
27 *conf* $\in C$, initially $\perp$
28 *acc*, a finite subset of I, initially $\emptyset$
29 *prepared-id* $\in I$, the id of the lastly prepared ballot
30 *ballot*, a record with fields:
31 *id* $\in T$, a tag initially $\langle 0, i\rangle$
32 *conf-index* $\in \mathbb{N}$, initially 0
33 *conf* $\in C$, initially $\perp$
34 *voted-ballots*, a set of ballots, initially $\emptyset$.

Fig. 2. State.

will be installed (in case reconfiguration succeeds). The *pxs.acc* subfield records which nodes from which quorums have responded during the current phase.
- Record *ballot* stores the information about the current ballot. This is used once the reconfiguration is initiated. The *ballot.id* subfield records the unique ballot identifier. The *ballot.conf-index* and the *ballot.conf* subfields record *pxs.conf-index* and *pxs.conf*, respectively, when the reconfiguration is initiated.

Finally, the *voted-ballot* set records the set of ballots that have been voted by the participants of a read quorum of the last installed configuration. In the remaining, a state field indexed by *i* indicates a field of the state of node *i*, e.g. tag_i refers to field *tag* of node *i*.

4.3. Read and write operations

The pseudocode for read and write operations appears in Figs. 3 and 4. Read and write operations proceed by accessing quorums of the currently active configurations. Each replica maintains a *tag* and a *value* for the data being replicated. Each read or write operation potentially requires two phases: one to *query* the replicas, learning the most up-to-date tag and value, and a second to *propagate* the tag and value to the replicas. First, the *query* phase starts when a read (Fig. 3, Line 1) or a write (Fig. 3, Line 11) event occurs and ends when a query-fix event occurs (Fig. 3, Line 22). In a *query* phase, the initiator contacts one read quorum from each active configuration, and remembers the largest tag and its associated value by possibly updating its own tag-value pair, as detailed in Section 4.4. Second, the *propagate* phase starts when the aforementioned query-fix event occurs and ends when a prop-fix (Fig. 3, Line 42) event occurs. In a *propagate* phase, read operations and write operations behave differently: a write operation chooses a new tag (Fig. 3, Line 35) 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, 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* (Fig. 3, Line 51). If a read operation discovers that a tag has been confirmed, it can skip the second phase (Fig. 3, Lines 62–70).

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 old one. In order to accomplish this, read or write operations save the set of currently active configurations, *op.cmap*, when a phase begins (Fig. 3, Lines 8, 18, 40); 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.

4.4. Communication and independent transitions

In this section, we describe the transitions that propagate information between processes. Those appear in Fig. 4. Information is propagated in the background via point-to-point channels that are accessed using send and recv actions. In addition, we present the join and join-ack actions which describe the way a node joins the system. The join input sets the current node into the joining status and indicates a set of nodes denoted *W* that it can contact to start being active. Finally, a leader election service informs a node that it is currently the leader, through a leader action, and the fail action models a disconnection.

The most tricky transitions are the communication transitions. This is due to the piggybacking of information in messages: each message conveys not only information related to the read and write operations (e.g. *tag*, *value*, *cmap*, *confirmed*) but also information related to the reconfiguration process (e.g. *ballot*, *pxs*, *voted-ballot*).

Moreover, all messages contain fields common to operations and reconfiguration: the set of nodes ids *world* the sender node knows of, and the current configuration map *cmap*. When node *i* receives a message, provided *i* is not failed or idle, it sets its status to active—completing the join protocol, if it has not already done so. It also updates its information with the message content: *i* starts participating in a new reconfiguration if the ballot received is larger than its ballot, *i* updates some of its *pxs* subfield (Lines 60–64) if *i* discovers that a pending consensus focuses on a larger indexed configuration than the one it is aware of (Line 58). That is, during a stale reconfiguration *i* might catch up with the actual reconfiguration, while aborting the stale one. The receiver also progresses in the reconfiguration (adding the sender id to its *pxs.acc* subfield, Lines 68, 71,74) if the sender uses the same ballot (Line 70), and responds to the right message of *i* (Line 67). Observe that if *i* discovers another consensus instance aiming at installing a configuration at a larger index, or if *i* discovers a larger ballot than its, then *i* sets its *pxs.phase* to idle. Thus, *i* stops participating in the reconfiguration.

In the meantime, *i* updates fields related to the read/write operations and either continues the phase of the current operation,

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

```
1  Input read_i
2  Effect:
3    if ¬ failed and status = active then
4      pnum1 ← pnum1 + 1
5      op.pnum ← pnum1
6      op.type ← read
7      op.phase ← query
8      op.cmap ← cmap
9      op.acc ← ∅
10
11 Input write(v)_i
12 Effect:
13   if ¬ failed and status = active then
14     pnum1 ← pnum1 + 1
15     op.pnum ← pnum1
16     op.type ← write
17     op.phase ← query
18     op.cmap ← cmap
19     op.acc ← ∅
20     op.value ← v
21
22 Internal query-fix_i
23 Precondition:
24   ¬ failed and status = active
25   op.type ∈ {read,write}
26   op.phase = query
27   ∀k ∈ ℕ, c ∈ C : op.cmap(k) = c
28     ⟹ (∃ R ∈ read-quorums(c) : R ⊆ op.acc)
29 Effect:
30   if op.type = read then
31     op.value ← value
32     op.tag ← tag
33   else
34     value ← op.value
35     tag ← ⟨tag.counter + 1, i⟩
36     op.tag ← tag
37   pnum1 ← pnum1 + 1
38   op.pnum ← pnum1
39   op.phase ← prop
40   op.cmap ← cmap
41   op.acc ← ∅
42 Internal prop-fix_i
43 Precondition:
44   ¬ failed and status = active
45   op.type ∈ {read,write}
46   op.phase = prop
47   ∀k ∈ ℕ, c ∈ C : op.cmap(k) = c
48     ⟹ (∃ W ∈ write-quorums(c) : W ⊆ op.acc)
49 Effect:
50   op.phase ← done
51   confirmed ← confirmed ∪ {op.tag}
52
53 Output read-ack(v)_i
54 Precondition:
55   ¬ failed and status = active
56   op.type = read
57   op.phase = done
58   v = op.value
59 Effect:
60   op.phase = idle
61
62 Output read-ack(v)_i
63 Precondition:
64   ¬ failed and status = active
65   op.type = read
66   op.phase = prop
67   op.tag ∈ confirmed
68   v = op.value
69 Effect:
70   op.phase = idle
71
72 Output write-ack_i
73 Precondition:
74   ¬ failed and status = active
75   op.type = write
76   op.phase = done
77 Effect:
78   op.phase = idle
```

Fig. 3. Read/write transitions.

or restarts it depending on the current phase and the incoming phase number (Lines 47–53). Node *i* compares the incoming tag *t* to its own *tag*. If *t* is strictly greater, it represents a more recent version of the object; in this case, *i* sets its *tag* to *t* and its value to the incoming value *v*. Node *i* updates its configuration map *cmap* with the incoming *cm*, using the update operator defined in Section 2. Furthermore, node *i* updates its *pnum*2(*j*) component for the sender *j* to reflect new information about the phase number of the sender, which appears in the *pns* component of the message. If node *i* is currently conducting a phase of a read or write operation, it verifies that the incoming message is "recent", in the sense that the sender *j* sent it after *j* received a message from *i* that was sent after *i* began the current phase. Node *i* uses the phase number to perform this check: if the incoming phase number *pnr* is at least as large as the current operation phase number (*op.pnum*), then process *i* knows that the message is recent.

If *i* is currently in a query or propagate phase and the message effectively corresponds to a fresh response from the sender (Line 47) then *i* extends its *op.cmap* record used for its current read and write operations with the *cmap* received from the sender. Next, if there is no gap in the sequence of configurations of the extended *op.cmap*, meaning that *op.cmap* ∈ *Truncated*, then node *i* takes notice of the response of *j* (Lines 49 and 50). In contrast, if there is a gap in the sequence of configuration of the extended *op.cmap*, then *i* infers that it was running a phase using an out-of-date configuration and restarts the current phase by emptying its field *op.acc* and updating its *op.cmap* field (Lines 51–53).

4.5. Reconfiguration

The pseudocode for reconfiguration appears in Figs. 4–8. 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.

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

Input join$(W)_i$
Effect:
  $status \leftarrow$ joining
  $world \leftarrow world \cup W$

Output join-ack$_i$
Precondition:
  $status =$ active
  $joined =$ **false**
Effect:
  $joined \leftarrow$ **true**

Input leader$(b)_i$
Effect:
  $isLeader \leftarrow b$

Input fail$_i$
Effect:
  $failed \leftarrow$ **true**

Output send$(W,v,t,cnf,cm,pns,pnr,b,p,vb)_{i,j}$
Precondition:
  $\neg\, failed$ **and** $status \neq$ idle
  $j \in world$
  $W = world$
  $v = value$
  $t = tag$
  $cnf = confirmed$
  $cm = cmap$
  $pns = pnum1$
  $pnr = pnum2(j)$
  $b = ballot$
  $p = pxs$
  $vb = voted\text{-}ballots$
Effect:
  None.

Input recv$(W,v,t,cnf,cm,pns,pnr,b,p,vb)_{j,i}$
Effect:
  if $\neg\, failed$ **and** $status \neq$ idle **then**
    $status \leftarrow$ active
    $world \leftarrow world \cup W$
    $confirmed \leftarrow confirmed \cup cnf$
    if $t > tag$ **then**
      $\langle value, tag\rangle \leftarrow \langle v, t\rangle$
    $cmap \leftarrow$ update$(cmap, cm)$
    $pnum2(j) \leftarrow \max(pnum2(j), pns)$
    if $op.phase \in$ {query,prop} **and** $pnr \geq op.pnum$ **then**
      $op.cmap \leftarrow$ extend$(op.cmap, cm)$
      if $op.cmap \in Truncated$ **then**
        $op.acc \leftarrow op.acc \cup \{j\}$
      else
        $op.acc \leftarrow \emptyset$
        $op.cmap \leftarrow cmap$
    if $b.id > ballot.id$ **then**
      $ballot \leftarrow b$
      $pxs.phase \leftarrow$ idle
      $pxs.acc \leftarrow \emptyset$
    if $p.conf\text{-}index > pxs.conf\text{-}index$ **then**
      if $recon\text{-}in\text{-}progress =$ **false then**
        $pxs.conf\text{-}index \leftarrow p.conf\text{-}index$
        $pxs.conf \leftarrow p.conf$
        $pxs.old\text{-}conf \leftarrow p.old\text{-}conf$
        $pxs.phase \leftarrow$ idle
        $pxs.acc \leftarrow \emptyset$
    $voted\text{-}ballots \leftarrow voted\text{-}ballots \cup vb$
    if $pxs.phase =$ prepare **then**
      if $pnr \geq pxs.pnum$ **then**
        $pxs.acc \leftarrow pxs.acc \cup \{j\}$
    else if $pxs.phase =$ propose **then**
      if $ballot \in vb$ **and** $ballot = b$ **then**
        $pxs.acc \leftarrow pxs.acc \cup \{j\}$
    else if $pxs.phase =$ propagate **then**
      if $cm(ballot.conf\text{-}index) = ballot.conf$ **then**
        $pxs.acc \leftarrow pxs.acc \cup \{j\}$

Fig. 4. Send/receive/other transitions.

Input recon$(c, c')_i$
Effect:
  if $\neg\, failed$ **and** $status =$ active **then**
    let $k = \max(\ell : cmap(\ell) \in C)$
    $pxs.conf\text{-}index \leftarrow k{+}1$
    $pxs.old\text{-}conf \leftarrow c$
    $pxs.conf \leftarrow c'$
    $pxs.phase \leftarrow$ idle
    $pxs.acc \leftarrow \emptyset$
    $recon\text{-}in\text{-}progress \leftarrow$ **true**

Internal init$_i(c)$
Precondition:
  $\neg\, failed$ **and** $status =$ active
  $c = pxs.conf \neq \perp$
  $k = pxs.conf\text{-}index \neq \perp$
  $cmap(k) = \perp$
  $cmap(k\text{-}1) = pxs.old\text{-}conf \neq \perp$
  if $k > 1$ **then** $cmap(k\text{-}2) = \pm$
  $isLeader =$ true
Effect:
  $pxs.phase \leftarrow$ idle
  $pxs.acc \leftarrow \emptyset$
  $ballot.conf \leftarrow c$
  $ballot.conf\text{-}index \leftarrow k$

Output recon-ack$(r)_i$
Precondition:
  $\neg\, failed$ **and** $status =$ active
  $recon\text{-}in\text{-}progress =$ **true**
  let $k = pxs.conf\text{-}index$
  $cmap(k) \in C$ **or**
    $cmap(k\text{-}2) \neq \pm$ **or**
    $cmap(k\text{-}1) \neq pxs.old\text{-}conf$
  if $cmap(k) = pxs.conf$ **then**
    $r =$ ok
  else $r =$ failed
Effect:
  $pxs.conf \leftarrow \perp$
  $pxs.conf\text{-}index \leftarrow \perp$
  $recon\text{-}in\text{-}progress \leftarrow$ **false**

Fig. 5. Initiate reconfiguration.

The reconfiguration protocol uses an optimized variant of Paxos [16,18]. The reconfiguration initialization is presented in Fig. 5. The reconfiguration is requested at some node through the recon action. If the requested node is not the leader the request is forwarded to the leader via the generic information exchange. Then, the leader starts the reconfiguration by executing

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

```
1  Internal prepare(b)_i
2  Precondition:
3     ¬ failed and status = active
4     isLeader = true
5     b = ballot
6     pxs.phase = idle
7  Effect:
8     pnum1 ← pnum1 + 1
9     pxs.pnum ← pnum1
10    pxs.acc ← ∅
11    ballot.id ← ⟨ballot.id.counter + 1, i⟩
12    pxs.phase ← prepare

13 Internal prepare-done(b)_i
14 Precondition:
15    ¬ failed and status = active
16    isLeader = true
17    b = ballot
18    pxs.phase = prepare
19    let k = ballot.conf-index
20    let c = cmap(k -1)
21    c ∈ C
22    ∃ R ∈ read-quorums(c) : R ⊆ pxs.acc
23 Effect:
24    pxs.prepared-id ← ballot.id
25    pxs.acc ← ∅
26    pxs.phase ← idle
```

Fig. 6. Prepare.

```
1  Internal init-propose(k)_i
2  Precondition:
3     ¬ failed and status = active
4     isLeader = true
5     ballot.conf-index = k ≠ ⊥
6     ballot.id = pxs.prepared-id
7     pxs.conf-index = ballot.conf-index
8     pxs.conf = ballot.conf
9  Effect:
10    pxs.phase ← idle
11    let S = {b ∈ voted-ballots : b.conf-index = k}
12    if S ≠ ∅ then
13       let b' = b'' : b''.id = argmax_{b∈S}(b.id)
14       ballot.conf ← b'.conf
15    voted-ballots ← voted-ballots ∪ {ballot}

16 Internal propose(k)_i
17 Precondition:
18    ¬ failed and status = active
19    ballot.conf-index = k ≠ ⊥
20    ballot ∈ voted-ballots
21 Effect:
22    pxs.phase ← propose
23    pnum1 ← pnum1 + 1
24    pxs.pnum ← pnum1
25    pxs.acc ← ∅

26 Internal propose-done(k)_i
27 Precondition:
28    ¬ failed and status = active
29    pxs.phase = propose
30    let k = ballot.conf-index
31    let c = cmap(k -1)
32    c ∈ C
33    ∃ R_1 ∈ read-quorums(c) :
34       R_1 ⊆ pxs.acc
35    ∃ W_1 ∈ write-quorums(c) :
36       W_1 ⊆ pxs.acc
37 Effect:
38    pxs.phase ← idle
39    cmap(k) ← ballot.conf
40    pxs.acc ← ∅
```

Fig. 7. Propose.

```
1  Internal propagate(k)_i
2  Precondition:
3     ¬ failed and status = active
4     k = ballot.conf-index
5     cmap(k) ∈ C
6  Effect:
7     pxs.phase ← propagate
8     pnum1 ← pnum1 + 1
9     pxs.pnum ← pnum1
10    pxs.acc ← ∅

11 Internal propagate-done(k)_i
12 Precondition:
13    ¬ failed and status = active
14    pxs.phase = propagate
15    k = ballot.conf-index
16    let c = cmap(k-1)
17    let c' = cmap(k)
18    c ∈ C
19    c' ∈ C
20    ∃ W_2 ∈ write-quorums(c) : W_2 ⊆ pxs.acc
21    ∃ R_2 ∈ read-quorums(c) : R_2 ⊆ pxs.acc
22 Effect:
23    cmap(k-1) ← ±
24    pxs.phase ← idle
25    pxs.acc ← ∅
```

Fig. 8. Propagate.

an init event, and the reconfiguration completes by a recon-ack event. More precisely, the recon(c, c') event is executed at some node i starting the reconfiguration aiming to replace configuration c by c'. To this end, this event records the reconfiguration information in the *pxs* field. That is, node i records the c and c' in *pxs.old-conf* and *pxs.conf*, respectively. Node i selects the index of its *cmap* that immediately succeeds the index of the latest installed configuration and records it in *pxs.conf-index* as a possible index for c' (Fig. 5, Line 4). Finally, i starts participating in the reconfiguration by reinitializing its *pxs.acc* field. The leader ℓ sets its reconfiguration information either during a recon event or when it receives this information from another node, as described in Section 4.4. The leader executes an init event and starts a new consensus instance to decide upon the k^{th} configuration, only if the *pxs* field is correctly set (e.g. *pxs.old-conf* must be equal to $cmap(k-1)$). If so, the *pxs.acc* is emptied, the configuration of this

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

consensus instance is recorded as the ballot configuration with k, its index.

The leader coordinates the reconfiguration, which consists of three phases: a *prepare* phase in which a ballot is made ready (Fig. 6), a *propose* phase (Fig. 7), in which the new configuration is proposed, and a *propagate* phase (Fig. 8), in which the results are distributed. The prepare phase, appearing in Fig. 6, sets a new ballot identifier larger than any previously seen ballot identifier, accesses a read quorum of the old configuration (Fig. 6, Line 22), thus learning about any earlier ballots, and associates the largest encountered ballot to this consensus instance. But, if a larger ballot is encountered, then *pxs.phase* becomes idle (Fig. 4, Line 56). When the leader concludes the prepare phase, it chooses a configuration to propose through an init-propose event: 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 (Fig. 6, Line 13). The propose phase, appearing in Fig. 7, then begins by a propose event, accessing both a read and a write quorum of the old configuration (Fig. 7, Lines 33–36). 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, appearing in Fig. 8, begins by a propagate event and accesses a read and a write quorum from the old configuration (Fig. 8, Lines 20–21); 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.

4.6. Good executions

We consider *good* executions of RDS, whose traces satisfy a set of environment assumptions. Those environment assumptions are the simple following *well-formedness* conditions:

Well-formedness for RDS:

- For every x and i:
  - No $\mathsf{join}(*)_i$, read_i, $\mathsf{write}(*)_i$ or $\mathsf{recon}(*,*)_i$ event is preceded by a fail_i event.
  - At most one $\mathsf{join}(*)_i$ event occurs.
  - Any read_i, $\mathsf{write}(*)_i$ or $\mathsf{recon}(*,*)_i$ is preceded by a join-ack $(\mathsf{rds})_i$ event.
  - Any read_i, $\mathsf{write}(*)_i$ or $\mathsf{recon}(*,*)_i$ is preceded by a -ack event for any preceding event of any of these kind.
- For every c, at most one $\mathsf{recon}(*,c)_i$ event occurs. Uniqueness of configuration identifier is achievable using local process identifier and sequence numbers.
- For every c, c', and i, if a $\mathsf{recon}(c,c')_i$ event occurs, then it is preceded by:
  - A $\mathsf{recon\text{-}ack}(c)_i$ event, and
  - A $\mathsf{join\text{-}ack}_j$ event for every $j \in members(c')$.

5. Proof of correctness (atomic consistency)

In this section, we show that the algorithm is correct. That is, we show that the read and write operations are linearizable. We depend on two lemmas commonly used to show linearizability: Lemmas 13.10 and 13.16 in [20]. This requires that there exists a partial ordering on all completed operations satisfying certain properties.[1]

Theorem 5.1. *Let S be an algorithm for read/write shared memory. Assume that for every execution, α, in which every operation completes, there exists a partial ordering, $\prec$, on all the operations in α with the following properties:*

(i) *all write operations are totally ordered, and every read operation is ordered with respect to all the writes,*
(ii) *the partial order is consistent with the external order of invocations and responses, that is, there do not exist read or write operations π_1 and π_2 such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$, and*
(iii) *every read operation that is ordered after any writes returns the value of the last write preceding it in the partial order; any read operation ordered before all writes returns v_0.*

Then S guarantees that operations are linearizable.

First, fix α such that every operation initiated by any node i completes, i.e., for each operation read_i and write_i event in α there is a corresponding $\mathsf{read\text{-}ack}_i$ and $\mathsf{write\text{-}ack}_i$ event, respectively, later in α. For each operation π in α at node i, we define the query-fix event (resp. prop-fix event) for π as the last query-fix event (resp. prop-fix event) that occurs during operation π and $tag(\pi)$ as the tag of i right after the query-fix event for π occurs. If query-fix for π never occurs, then $tag(\pi)$ is undefined. Moreover, we define a partial order, $\prec$, in terms of the tags: the write operations are totally ordered in terms of their (unique) tags, and each read operation is ordered immediately following the write operation identified by the same tag. This ordering immediately satisfies conditions (i) and (iii). The main purpose of this section, then, is to show that this order satisfies condition (ii).

5.1. Ordering configurations

Before we can reason about the consistency of the operations, however, we must show that nodes agree on the active configurations. Observe that there is a single default configuration c_0 in the *cmap* field of every node of the system when the algorithm starts, as indicated at Line 7 of Fig. 2. For index ℓ, we say that the configuration of index ℓ is *well-defined* if there exists a configuration, *config*, such that for all nodes i, at all points in α, $cmap(\ell)_i$ is either undefined ($\perp$), removed ($\pm$), or equal to *config*. In particular, no other configuration is ever installed in slot ℓ of the *cmap*. We first show, inductively, that configuration ℓ is well-defined for all ℓ. The following proof, at its heart, is an extension of the proof in [16]

[1] In [20], a fourth property is included, assuming that each operation is preceded by only finitely many other operation. This is unnecessary, as it is implied by the other properties.

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

showing that Paxos ensures agreement. It has been modified to fit the presented pseudocode and to be compatible with the rest of the algorithm; it has been extended to handle changing configurations. (The original proof in [16] assumes a single quorum system/configuration, and shows that the participants can agree on a sequence of values.)

Theorem 5.2. *For all executions, for all ℓ, for any $i, j \in I$, if $cmap(\ell)_i, cmap(\ell)_j \in C$ then $cmap(\ell)_i = cmap(\ell)_j$ at any point in α.*

Proof. First, initially $cmap(0)_i = cmap(0)_j = c_0$ for any $i, j \in I$, by definition. We proceed by induction: assume that for all $\ell' < \ell$, $cmap(\ell')_i = cmap(\ell')_j$ (so that we can omit the index i and j and denote this by $cmap(\ell')$). We show that $cmap(\ell)_i = cmap(\ell)_j$.

Assume, by contradiction, that there exist two propose-done(ℓ) events, ρ_1 and ρ_2 at nodes i and j, respectively, that install two different configurations in slot ℓ of i and j's *cmap* in Fig. 7, Line 39. Let $b = ballot_i$ immediately after ρ_1 occurs and $b' = ballot_j$ immediately after ρ_2 occurs. The ballot when the two operations complete must refer to different configurations: $b.conf \neq b'.conf$. Without loss of generality, assume that $b.id < b'.id$. (Ballot identifiers are uniquely associated with configurations, so the two ballots cannot have the same identifier.)

At some point, a prepare(b') action must have occurred at some node—we say in this case that ballot b' has been prepared. First, consider the case where b' was prepared as part of a recon operation installing configuration ℓ. Let R be a read-quorum of configuration $cmap(\ell - 1)$ accessed by the prepare-done of ballot b', and let W_1 be a write-quorum of $cmap(\ell - 1)$ accessed by the propose-done associated with b. Since $cmap(\ell-1)_i = cmap(\ell-1)_j$ for any $i, j \in I$ by the inductive hypothesis, there is some node $i' \in R \cap W_1$. There are two sub-cases to consider: i' processed either the prepare first or the propose first. If i' processed the prepare first, then the propose would have been aware of ballot b', and hence the ballot identifier at the end of the proposal could have been no smaller than $b'.id$, contradicting the assumption that $b.id < b'.id$. Otherwise, if i' processed the propose first, then ballot b ends up in $voted\text{-}ballots_{i'}$, and eventually in $voted\text{-}ballots_j$. This ensures that j proposes the same configuration as i, again contradicting our assumption that ρ_1 and ρ_2 result in differing configurations for ℓ.

Consider the case, then, where b' was prepared as part of a recon operation installing a configuration $\ell' < \ell$. In this case, we can show that $b.id \geq b'.id$, contradicting our assumption. In particular, some recon for ℓ' must terminate prior to the ρ_1 and ρ_2 beginning reconfiguration for ℓ. By examining the quorum intersections, we can show that the identifier associated with ballot b' must have been passed to the propose for this recon for ℓ', and from there to the propose of a recon for $\ell' + 1$, and so on, until it reaches the propose for ρ_1, leading to the contradiction.

We can therefore conclude that if two recon s complete for configuration ℓ, they must both install the same configuration, and hence $cmap(\ell - 1)_i = cmap(\ell - 1)_j$ for any $i, j \in I$. □

5.2. Ordering operations

We now proceed to show that tags induce a valid ordering on the operations, that is, if operation π_1 completes before π_2 begins, then $tag(\pi_1) \leq tag(\pi_2)$, and if π_2 is a write operation then the inequality is strict. We first focus on the case where π_1 is a two-phase operation; that is, π_1 is not a read operation that short-circuits the second phase due to a tag being previously confirmed.

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.

In order to formalize this, we define the $tag(\ell)$, for reconfiguration ℓ, as the smallest tag found at any node i immediately after a propose-done(ℓ)$_i$ event occurs. If no propose-done(ℓ) event occurs in reconfiguration ℓ, then $tag(\ell)$ is undefined. We first notice that any node that has received information on configuration $c(\ell)$ has a tag at least as large as $tag(\ell)$:

Invariant 5.3. *If $cmap(\ell)_i \in C \cup \{\pm\}$ (i.e., node i has information on configuration $c(\ell)$), then $tag_i \geq tag(\ell)$.*

Proof. The proof is done by induction on events in α. The base case is immediate since a propose-done(ℓ) must have occurred by definition of *cmap*. Assume that prior to some point in α, the invariant holds. There are two ways that $cmap(\ell)_i$ is set $\neq \perp$: either as a result of a propose-done event, in which case the invariant follows by definition, or by receiving a message from another node, j, in which case j must have previously been in a state where $cmap(\ell)_j \in C \cup \{\pm\}$, and by the inductive hypothesis $tag_j \geq tag(\ell)$. Since i received a message from j, the result follows from Lines 43 and 44 of Fig. 4. □

This invariant allows us to conclude two facts about how information is propagated by reconfiguration operations: first, each reconfiguration has at least as large a tag as the prior reconfiguration, and second, an operation has at least as large a tag as the previous reconfiguration.

Corollary 5.4. *For all $\ell > 0$ such that $tag(\ell)$ is defined, $tag(\ell) \leq tag(\ell + 1)$.*

Proof. A recon$_i$ event where k is set to $\ell + 1$ can occur only after i has received information about configuration ℓ, i.e., only if $cmap(\ell)_i \in C$ due to the precondition at Fig. 5, Line 4. Thus Invariant 5.3 implies that $tag_i \geq tag(\ell)$ when the recon$_i$ occurs. Any node that receives a message relating to the reconfiguration also receives the tag, implying that any node j that performs a propose-done($\ell + 1$) also has a tag at least that large. □

Corollary 5.5. *Let π be a read or write operation at node i in α and assume that $cmap(\ell) \in C$ immediately prior to any* query-fix *event for π. Then $tag(\ell) \leq tag(\pi)$, and if π is a write operation then $tag(\ell) < tag(\pi)$.*

Proof. Invariant 5.3 implies that by the time the query-fix event occurs, $tag_i \geq tag(\ell)$. In the case of a read, the corollary follows immediately. In the case of a write operation, notice that the query-fix event increments the tag. □

We next need to consider the relationship between a read or write operation and the following reconfiguration. The next lemma shows that a read or write operation correctly propagates its tag to the reconfiguration operation.

Lemma 5.6. *Let π be a read or write operation at node i, and let ℓ be the largest entry in $cmap_i$ not equal to $\perp$ at the time immediately preceding the* query-fix *event for π. Then $tag(\pi) \leq tag(\ell + 1)$.*

Proof. Consider the prop-fix$_i$ event for π and the propose-done $(\ell + 1)_j$ event at node j. Note that we have not yet shown anything about the ordering of these two events. Let W be a write quorum associated with the prop-fix and let R be a read quorum associated with the propose-done($\ell + 1$)—i.e., such that $W \subseteq op.acc$ and $R \subseteq op.acc$ immediately prior to the prop-fix event. Let $i' \in R \cap W$. This follows from Theorem 5.2, in that both refer to quorums of the same configuration, and the assumption that every read quorum intersects every write quorum in a configuration. First, we show that i' must receive the message from i associated with π before

sending the message to j associated with the reconfiguration. Otherwise, node i' would have sent a message to node i including information about configuration $\ell + 1$. However, by assumption, configuration ℓ is the largest configuration known by i. Since i' receives the message from i before sending the message to j, node i' includes the tag from π in the message to j, leading to the desired result. □

We can now show that for any execution, α, it is possible to determine a linearization of the operations. As discussed previously, we need to show that if operation π_1 precedes operation π_2, then $tag(\pi_1) \leq tag(\pi_2)$; if π_1 is a write operation, then $tag(\pi_1) < tag(\pi_2)$.

Theorem 5.7. *If operation π_1 completes before operation π_2 begins, then*

- *$tag(\pi_1) \leq tag(\pi_2)$ in any case and*
- *$tag(\pi_1) < tag(\pi_2)$ if π_1 is a write operation.*

Proof. On the one hand, assume that π_1 is not a one phase read operation but a two phase operation. Let i be the node initiating operation π_1 while j is the node initiating π_2. There are three cases to consider.

(i) First, assume there exists k such that $op.cmap(k)_i = op.cmap(k)_j \in C$ meaning that π_1 and π_2 use a common configuration. With no loss of generality, let c denote this common configuration. Then the write quorum(s) of c accessed in action prop-fix$_i$ for π_1 (write quorum(s) $W \subseteq op.acc_i$) intersects the read quorum(s) accessed in action query-fix$_j$ for π_2 (read quorum(s) $R \subseteq op.acc_j$) ensuring that tag_j right after the query-fix$_j$ for operation π_2 is larger than tag_i right after the query-fix$_i$ for operation π_1. By definition of $tag(\pi_1)$ and $tag(\pi_2)$, the result follows.

(ii) Second, assume that the smallest k such that $op.cmap(k)_i \in C$ when prop-fix$_i$ for π_1 occurs (i.e., k is the smallest index of configuration accessed during π_1), is larger than the largest ℓ such that $op.cmap(\ell)_j \in C$ when query-fix$_i$ for π_2 occurs (i.e., ℓ is the largest index of configuration accessed during π_2). This case cannot occur. Prior to π_1, some reconfiguration installing configuration $\ell + 1$ must occur. During the final phase of the reconfiguration, a read quorum of configuration ℓ is notified of the new configuration. Therefore, during the query phase of π_2, the new configuration for $\ell + 1$ would be discovered, contradicting our assumption.

(iii) Third, assume that the largest k such that $op.cmap(k)_i \in C$ is accessed by π_1 during prop-fix$_i$, is smaller than the smallest ℓ such that $op.cmap(\ell)_j \in C$ is accessed by π_2 during query-fix$_j$. Then, Lemma 5.6 shows that $tag(\pi_1) \leq tag(\ell)$; Corollary 5.4 shows that $tag(\ell) \leq tag(\ell')$; finally, Corollary 5.5 shows that $tag(\ell') \leq tag(\pi_2)$ and if π_2 is a write operation then the inequality is strict. Together, these show the required relationship of the tags.

On the other hand, consider the case where π_1 is a one-phase read operation. A one-phase read operation occurs at node i only if the $op.tag_i$ belongs to $confirmed_i$. In order for the tag to be confirmed, there must exist some prior two-phase operation, π', that put the tag in the *confirmed* set. This operation must have completed prior to π_1, and hence prior to π_2 beginning. Since π' is a two-phase operation, we have already shown that $tag(\pi') \leq tag(\pi_2)$. Moreover, it is clear that $tag(\pi') = tag(\pi_1)$, implying the desired result. □

6. 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 [21] might take arbitrarily long to stabilize under these conditions.) (ii) We show that after the algorithm stabilizes, every 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.

6.1. 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 stable 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 participants;
(iv) All other enabled actions are processed with zero time passing on the local clock.

Generally, in quorum-based algorithms, operations are guaranteed to terminate provided that at least one quorum does not fail. In contrast, 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, i.e., $\ell time(\alpha')$ (ii) a reconfiguration operation.

We place some easily satisfied restrictions on reconfiguration. First, we assume that each node in a new configuration has completed the join 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: *recon-spacing* saying that for any k, the propose-done(k) events and the propose-done($k+1$) 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 ℓ among the joined nodes at time $\ell time(\alpha') + e$ and that ℓ remains alive forever. For example, a leader may be chosen among the members of a configuration based on the value of an identifier, however, the leader does not need to belong to any configuration.

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

6.2. 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 6.1. *Let γ be the first recon$_i$ for all $i \in I$, let t be the time γ occurs, and let $t' = \max(\ell time(\alpha'), t) + e$. Then, the leader ℓ starts the reconfiguration process at the latest at time $t' + 2d$.*

Proof. With no loss of generality, let $\gamma = \text{recon}(c, c')_i$. In the following we show that the preconditions of the $\text{init}(c')_\ell$ event are satisfied before time $t' + 2d$. Denote k by $k = \text{argmax}(k' : \forall j \in I, cmap_j(k') \in C)$ at time t. That is $c = c(k)$. First, since no $\text{recon}(c, c')$ occurs prior to time t, c' is not installed yet and $cmap(k+1) = \perp$ at time t.

Second, we show that if $k > 0$ then $cmap(k-1) = \pm$ at any node at time $t + d$. Assume that $k > 0$ and some recon-ack(ok), installing $c(k)$ occurs before time t in the system, and let γ_k be the last of these events. Since a matching $\text{recon}(c(k-1), c)$ precedes γ_k, *recon-readiness* and *join-connectivity* imply that members of $c(k-1)$ and c know each other plus the leader at time t'. That is, less than $2d$ time after (before time $t' + 2d$), a $\text{propagate-done}(k)$ event occurs and $cmap(k-1)$ is set to $\pm$ at any of these nodes.

Next, by examination of the code, just after the $\text{recon}(c, c')$ event, $pxs.conf\text{-}index_i = k + 1$, $pxs.conf_i = c' \neq \perp$ and $pxs.old\text{-}conf_i = c \neq \perp$. Prior to time $t' + d$, a $\text{send}(*, cm, *, p, *)_{i,\ell}$ event occurs with $cm = cmap_i$ and $p = pxs_i$. That is, before $t' + 2d$, the corresponding $\text{recv}(*, cm, *, p, *)_{i,\ell}$ event occurs. Therefore, the $p.conf\text{-}index = k + 1$ subfield received is larger than ℓ's $pxs.conf\text{-}index_\ell$, and subfields $pxs.conf_\ell$ and $pxs.old\text{-}conf_\ell$ are set to the received ones, c' and c, respectively.

Consequently, d time after the $\text{recon}(c, c')_i$ event occurs with $k = pxs.conf\text{-}index_\ell - 1$, preconditions of event $init(c')_\ell$ are satisfied and therefore this event occurs at the latest at time $t' + 2d$. □

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

Lemma 6.2. *After time $\ell time(\alpha') + e + 2d$, ℓ knows always about the largest ballot in the system.*

Proof. 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 $\ell time(\alpha')$ or ℓ is aware of b at the time it creates it. Since ℓ is the leader at time $\ell time(\alpha') + e$, we know that ℓ has started joining 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, assumption (ii) 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. Assumption (iii) 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 algorithm stabilization. In Theorem 6.4 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 6.3. *Assume that ℓ starts the reconfiguration process, initiated by $\text{recon}(c, c')$, at time t. Then the corresponding reconfiguration completes no later than $\max(t, \ell time(\alpha') + e + 2d) + 5d$.*

Proof. First-of-all, observe by assumption (iv) that any internal enabled action is executed with no time passing. As a result, if at time $\ell time(\alpha') + e + 2d$, ℓ knows that a reconfiguration should have been executed $pxs.conf\text{-}index_\ell \neq \perp$ (Fig. 5, Line 15) but the reconfiguration is not complete yet $cmap(pxs.conf\text{-}index)_\ell = \perp$ (Fig. 5, Line 16), then the reconfiguration restarts immediately. In case, ℓ the reconfiguration request is received at time $t' > \ell time(\alpha') + e + 2d$, the reconfiguration starts immediately. Let $t'' = \max(t', \ell time(\alpha') + e + 2d)$.

Next, we subsequently show a bound on each of the three phases of the reconfiguration started at time t''. Observe that if an $\text{init}(c)_\ell$ event occurs at time t', then a prepare_ℓ occurs too. By Lemma 6.2 and since $t'' \geq \ell time(\alpha') + e + 2d$, $ballot_\ell$ augmented by this event is at this time, the strictly highest one. By *join-connectivity* and *recon-readiness*, messages are sent from ℓ to every member of configuration $c(k-1)$ where $k = \text{argmax}(k' : cmap(k')_\ell \in C)$. Therefore they update their *ballot* before time $t'' + d$, and ℓ receives their answer no later than time $t'' + 2d$. Because of the prepare-done_ℓ occurring, the prepare phase completes in $2d$.

For the propose phase, observe that $\text{init-propose}(k)_\ell$ and $\text{propose}(k)_\ell$ occur successively with no time passing. Next, all members of configuration $c(k-1)$ receive a message from i, update their *voted-ballot* field, execute their $\text{propose}(k)$ event and send in turn a message no later than time $t'' + 3d$. Consequently the participation of the members of $c(k-1)$ completes the propose phase before time $t + 4d$.

Since $cmap(k) = pxs.conf$ at time $t'' + 4d$ at all members of $c(k-1)$ and $c(k)$, $\text{recon-ack}(ok)$ occurs without any time passing. Notice that at time $t + 4d$, all members of configuration $c(k-1)$ and $c(k)$ have set their $cmap(k)$ to *ballot.conf* by the $\text{propose-done}(k)$ effect. Thus, $\text{propagate}(k)$ occurs at all these nodes at time $t'' + 4d$ or earlier and no more than d time later, they all have exchanged messages of the propagate phase. That is, the propagate phase completes in one message delay and the whole reconfiguration ends no later than time $t'' + 5d$. □

Theorem 6.4. *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. This proof shows that the prepare phase of the reconfiguration can be skipped under the present conditions. Let $\gamma_{k''}$ and $\gamma_{k'}$ be the reconfigurations that aim at installing configuration c'' and configuration c', respectively. After $\gamma_{k'}$, $ballot.id_\ell = pxs.prepared\text{-}id_\ell$, since by Lemma 6.2 $ballot_\ell$ remains unchanged. That is, after the $\text{init}(c'')_\ell$ event the $\text{init-propose}(k'')_\ell$ occurs without any time passing. From this point on, the propose phase and the propagation phase are executed like mentioned in proof of Theorem 6.3. Since the propose phase is done in $2d$ and the propagation phase requires d time, $\gamma_{k''}$ completes successfully by time $t + 3d$. □

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

Algorithm	Min. operation latency	Max. operation latency	Max. configuration installation delay	Max. configuration removal delay
RAMBO	4δ	$8d+\epsilon$	$10d+\epsilon$	$4(s-1)d+\epsilon$
RAMBO II	4δ	$8d+\epsilon$	$10d+\epsilon$	$4d+\epsilon$
RDS	2δ	$8d+\epsilon$	$5d+\epsilon$	0

Fig. 9. Time complexity of RAMBO, RAMBO II, and RDS. Letter δ refers to a lower-bound on message delay, letter d refers to an upper-bound on message delay, s refers to the number of active configurations, and $\epsilon = O(1)$ is a constant independent from message delay.

6.3. Bounding read–write delays

In this section, we present bounds on the duration of read/write operations under the assumptions stated in the previous section. Recall from Section 4 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 most $4d$ 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 6.5. The final result is a general bound of $8d$ on the duration of any read/write operation.

Lemma 6.5. *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 no later than at time* $\max(t - e - 2d, \ell time(\alpha'))$ *and denote t' by* $\max(t, \ell time(\alpha') + e + 2d)$. *Then this phase completes at the latest at time $t' + 4d$.*

Proof. Let $k = \text{argmax}\{\ell : \forall j \in I, cmap(\ell)_j \in C\}$ at time $t' - d$. First we show that any of these j knows about configuration $c(k)$ at time t'. Because of *configuration-viability*, members of $c(k)$ are active during the reconfiguration. Moreover because of *join-connectivity* and since join-ack$_j$ occurs prior to time $t' - e - d$, we know that j is connected to members of $c(k)$ at time $t' - d$. Because of assumption (ii), d time later j receives a message from members of $c(k)$. That is at time t', j knows about configuration $c(k)$.

For the phase to complete, node i sends a message to all the nodes in its $world_i$ set (the set of nodes i knows of). Next, node i has to wait until the accurate response of some members of the active configurations. Hence each phase needs at least two message delays to complete.

From now on, assume that some recon-ack occurs setting $cmap(k+1)$ to an element of C after time $t' - d$ and prior to time $t' + 2d$. That is j might learn about this new configuration $c(k+1)$ and the phase might be delayed an additional $2d$ time since j has now to contact a quorum of configuration $c(k+1)$.

Since a recon-ack event occurs after time t, *recon-spacing* ensures that no further recon-ack occurs before time $t'+5d$, and the phase completes at most at time $t' + 4d$. Especially, the phase can complete in only $2d$ if no recon-ack event occurs after time $t' - d$ and before $t' + 2d$. □

Theorem 6.6. *Consider a read operation that starts at node i at time t and denote t' by* $\max(t, \ell time(\alpha') + e + 2d)$:

(i) *If no write propagation is pending at any node and no reconfiguration is ongoing, then it completes at the latest at time $t' + 2d$.*
(ii) *If no write propagation is pending, then it completes no later than time $t' + 8d$.*

Consider a write operation that starts at node i at time t. Then it completes at the latest at time $t' + 8d$.

Proof. When a read$_i$ or write$_i$ event occurs at time t', the phase is set to query. From now on, by Lemma 6.5, we know that the query fix-point is reached and the current $phase_i$ becomes prop no later than time $t' + 4d$. If the operation is a write, then a new tag_i is set that does not belong to the exchanged *confirmed* tags set yet. If the operation is a read, the tag_i is the highest received one. This tag was maintained by a member of the read quorum queried, and it was confirmed only if the phase that propagated it to this member has completed.

From this point on, if the tag appears not to be confirmed to i, then in any operation the propagation phase fix-point has to be reached. But, if the tag is already confirmed and i learns it (either by receiving a confirmed set containing it or by having propagated it itself) then the read operation can terminate directly by executing a read-ack$_i$ event without any time passing, after a single phase. By Lemma 6.5, this occurs prior to time $t' + 4d$; and at time $t' + 2d$ if no reconfiguration is concurrent.

Likewise by Lemma 6.5, the propagation phase fix-point is reached in at most $4d$ time. That is, any operation terminates at the latest at time $t' + 8d$. □

7. Complexity improvements

Here we explain how RDS improves on RAMBO [21] and RAMBO II, [11]. The time complexity is given as a function of the message delays. RAMBO and RAMBO II use an external consensus algorithm to install new configurations, and a separated mechanism to remove old configurations. As previously said, coupling the installation of new configurations with the removal of old configurations makes the RDS reconfiguration mechanism more efficient. We denote by s the number of active configurations and by $\epsilon = O(1)$ a constant independent from message delay.

7.1. Configuration installation

RAMBO and RAMBO II time complexities have previously been measured after system stabilization where message delay is upper bounded by d [23,21,12,11]. These results are compared in Fig. 9. As far as we know, when the system stabilizes installing a new configuration may take $10d + \epsilon$, not only in RAMBO but also in RAMBO II, since both algorithms use the same installation mechanism. In contrast, we know by Theorems 6.3 and 6.4 that the installation of a new configuration is upper bounded by $5d + \epsilon$ and can even complete in $3d + \epsilon$ in RDS. Hence, RDS speeds up configuration installation by at least a factor of 2.

7.2. Configuration removal

An even more significant improvement relies on the time needed to remove old configurations from the list of active configurations. This represents a critical period of time, during which the system reliability depends on the non-faultiness of all old configurations. The configuration removal of RAMBO, called garbage collection, removes each old configuration successively

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

in $4d + \epsilon$ leading to $4(s-1)d + \epsilon$ time to remove $s-1$ old configurations. The configuration removal mechanism of RAMBO II, called configuration upgrade, removes all $s-1$ old configurations in a row in $4d+\epsilon$ time. Conversely, RDS does not need any additional configuration removal process since the configuration removal is already integrated in the installation mechanism. That is, no old configurations can make RDS fail: its reliability relies only on the one or two current configurations at any time.

7.3. Operations

Furthermore, it has been shown that operations complete within $8d + \epsilon$ in RAMBO and RAMBO II, however, it is easy to see that they require at least 4δ to complete, where δ is a lower bound on the message delay, since each operation consists in two successive message exchanges with quorums. Finally, although the time needed for writing is the same in RAMBO, RAMBO II, and RDS, in some cases the read operations of RDS are twice faster than the read operations of RAMBO and RAMBO II (cf. Theorem 6.6). Thus, the best read operation time complexity that RDS achieves is optimal [6].

7.4. Communication complexity

Finally, we can not measure the detailed message complexity, since the amount of messages depends on the number of active configurations and the number of members by quorum. Nevertheless, since RDS limits the number of active configurations s to 2 while neither RAMBO nor RAMBO II bound explicitly the number s of active configurations, seemingly RDS presents lower message complexity. Finally, some improvements on the message complexity of RAMBO appeared in [14] and rely on the manner nodes gossip among each other, but these improvements require a strong additional assumption and adapting these improvements in RDS remains an open question.

8. Experimental results

In this section we attempt to understand the cost of reconfiguration by comparing RDS to a non-reconfigurable distributed shared memory. There is an inherent trade-off between reliability – here, a result of quorums and reconfiguration – and performance. These results illustrate this trade-off.

We implemented the new algorithm based on the existing RAMBO codebase [8] 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 simple and efficient algorithms to implement a replicated distributed shared memory. Our goal is to achieve performance similar to the simple 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 [2] (the "ABD protocol"). In this section we 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 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 [24]. 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.

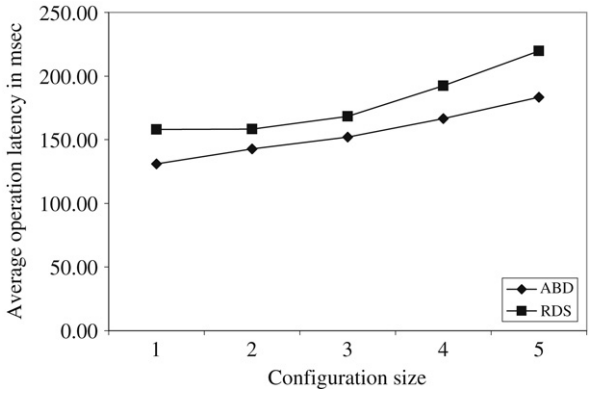

Fig. 10. Average operation latency as size of quorums changes.

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. Each point on the graphs represents an average of ten scenario runs. One hundred read and write operations each (implemented as reads and writes of a Java Integer) are performed independently and the latency is an average of time intervals from operation invocation to corresponding acknowledgment.

8.1. Quorum size

In the first experiment, we examine how the RDS algorithm responds to different size quorums (and hence different levels of fault-tolerance). We measure the average operation latency while varying the size of the quorums. Results are depicted in Fig. 10.

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 configuration sizes, ranging from 1 to 5. The ratio of read operations to write operations is set to 1. 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.

For ABD, there is no reconfiguration.

8.2. Load

In the second set of experiments, we test how the RDS algorithm responds to varying load. Fig. 11 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 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. The local minimum at four reader/writers can be explained by increased messaging activity that is associated with quorum communication.

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

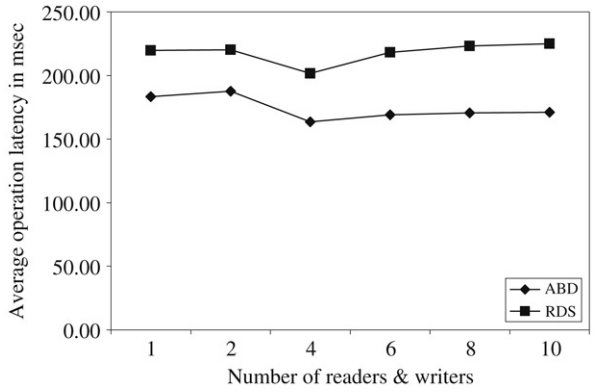

Fig. 11. Average operation latency as number of nodes performing read/write operations changes.

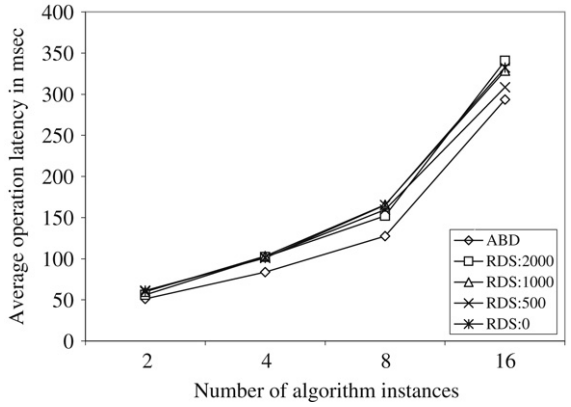

Fig. 12. Average operation latency as the reconfiguration and the number of participants changes.

8.3. Reconfiguration

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 Fig. 12. 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 ms. 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.

8.4. 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 less robust 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. An interesting open question might be adapting these techniques to probabilistic quorum systems that use less communication [15].

9. Conclusion

We have presented RDS, a new distributed algorithm for implementing a reconfigurable shared memory in dynamic, asynchronous networks.

Prior solutions (e.g., [21,11]) 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.

Acknowledgments

We are grateful to Nancy Lynch for discussions at the early stage of this work. We would also like to thank the anonymous reviewers for their careful reading and helpful comments.

References

[1] J. Albrecht, S. Yasushi, RAMBO for dummies, Tech. Rep., HP Labs, 2005.
[2] H. Attiya, A. Bar-Noy, D. Dolev, Sharing memory robustly in message-passing systems, Journal of the ACM 42 (1) (1995) 124–142.
[3] B. Awerbuch, P. Vitanyi, Atomic shared register access by asynchronous hardware, in: Proceedings of 27th IEEE Symposium on Foundations of Computer Science, 1986, pp. 233–243.

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007

[4] K. Birman, T. Joseph, Exploiting virtual synchrony in distributed systems, in: Proceedings of the 11th ACM Symposium on Operating Systems Principles, ACM Press, 1987, pp. 123–138.
[5] R. Boichat, P. Dutta, S. Frolund, R. Guerraoui, Reconstructing paxos, SIGACT News 34 (2) (2003) 42–57.
[6] P. Dutta, R. Guerraoui, R.R. Levy, A. Chakraborty, How fast can a distributed atomic read be? in: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, ACM Press, New York, NY, USA, 2004, pp. 236–245.
[7] B. Englert, 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.
[8] C. Georgiou, P. Musial, A. Shvartsman, Long-lived RAMBO: Trading knowledge for communication, in: Proceedings of 11th Colloquium on Structural Information and Communication Complexity, Springer, 2004, pp. 185–196.
[9] C. Georgiou, P. Musial, A.A. Shvartsman, Developing a consistent domain-oriented distributed object service, in: Proceedings of the 4th IEEE International Symposium on Network Computing and Applications, Cambridge, MA, USA, 2005, pp. 149–158.
[10] D.K. Gifford, Weighted voting for replicated data, in: Proceedings of the Seventh ACM Symposium on Operating Systems Principles, ACM Press, 1979, pp. 150–162.
[11] S. Gilbert, N. Lynch, 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.
[12] S. Gilbert, N. Lynch, A. Shvartsman, RAMBO II: Implementing atomic memory in dynamic networks, using an aggressive reconfiguration strategy, Tech. Rep., LCS, MIT, 2003.
[13] V. Gramoli, Distributed shared memory for large-scale dynamic systems, Ph.D. in Computer Science, INRIA — Université de Rennes 1, November 2007.
[14] V. Gramoli, P.M. Musial, A.A. Shvartsman, Operation liveness and gossip management in a dynamic distributed atomic data service, in: Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems, 2005, pp. 206–211.
[15] V. Gramoli, M. Raynal, Timed quorum systems for large-scale and dynamic environments, in: Proceedings of the 11th International Conference on Principles of Distributed Systems, in: LNCS, vol. 4878, Springer-Verlag, 2007, pp. 429–442.
[16] L. Lamport, The part-time parliament, ACM Transactions on Computer Systems 16 (2) (1998) 133–169.
[17] L. Lamport, Paxos made simple, ACM SIGACT News (Distributed Computing Column) 32 (4) (2001) 18–25.
[18] L. Lamport, Fast paxos, Distributed Computing 19 (2) (2006) 79–103.
[19] B.W. Lampson, How to build a highly available system using consensus, in: Proceedings of the 10th International Workshop on Distributed Algorithms, Springer-Verlag, London, UK, 1996, pp. 1–17.
[20] N. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, 1996.
[21] N. Lynch, A. Shvartsman, RAMBO: A reconfigurable atomic memory service for dynamic networks, in: Proceedings of 16th International Symposium on Distributed Computing, 2002, pp. 173–190.
[22] N. Lynch, A. Shvartsman, Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts, in: Proceedings of 27th International Symposium on Fault-Tolerant Comp., 1997, pp. 272–281.
[23] N. Lynch, A. Shvartsman, RAMBO: A reconfigurable atomic memory service for dynamic networks, Tech. Rep., LCS, MIT, 2002.
[24] P. Musial, A. Shvartsman, Implementing a reconfigurable atomic memory service for dynamic networks, in: Proceedings of 18th International Parallel and Distributed Symposium — FTPDS WS, 2004, p. 208b.
[25] Y. Saito, S. Frolund, A.C. Veitch, A. Merchant, S. Spence, Fab: building distributed enterprise disk arrays from commodity components, in: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, 2004, pp. 48–58.
[26] Special issue on group communication services, Communications of the ACM 39(4).
[27] R.H. Thomas, A majority consensus approach to concurrency control for multiple copy databases, ACM Transactions on Database Systems 4 (2) (1979) 180–209.
[28] E. Upfal, A. Wigderson, How to share memory in a distributed system, Journal of the ACM 34 (1) (1987) 116–127.
[29] R. van der Meyden, Y. Moses, Top-down considerations on distributed systems, in: Proceedings of the 12th International Symposium on Distributed Computing, in: LNCS, vol. 1499, Springer, 1998, pp. 16–19.

Gregory Chockler is a research staff member at IBM Research where he is affiliated with the distributed Middleware group at Haifa Research Lab. He received the B.Sc., M.Sc., and Ph.D. degrees in computer science from the Hebrew University of Jerusalem in 1993, 1997, and 2003 respectively. He spent the year of 2002 with IBM Research before joining, in 2003, the Nancy Lynch's group at CSAIL/MIT as a postdoctoral associate. He returned to IBM in 2005. Dr. Chockler's research interests span all areas of distributed computing, including both theory and practice. His most significant past work was in the area of group communication, reliable distributed storage, and theory of fault tolerant computing in wireless ad hoc networks. His current interests are centered around building highly scalable distributed systems to empower future generations of the enterprise data centers. He regularly publishes and serves on conference organizing committees in these fields, including flagship distributed computing conferences of ACM and IEEE. He has delivered numerous scientific lectures at scientific symposia, leading universities, and industrial research institutes, including several invited keynotes and tutorials. At IBM, he is a co-chair of the Distributed and Fault-Tolerant Computing professional interest community.

Seth Gilbert is currently a postdoc in the Distributed Programming Laboratory (LPD) at EPFL in Switzerland. His research focuses primarily on the challenges associated with highly dynamic distributed systems, particularly wireless ad hoc networks. Prior to EPFL, Seth received his Ph.D. from MIT in the Theory of Distributed Systems (TDS) group under Nancy Lynch. Previously, he worked at Microsoft, developing new tools to simplify the production of large-scale software. He graduated from Yale University with a degree in Electrical Engineering and Math.

Vincent Gramoli is a postdoc at EPFL LPD and University of Neuchâtel in Switzerland working on software transactional memory. He received an MS degree in computer science from Université Paris 7 and an MS degree in distributed systems from Université Paris 11. In 2004, he worked as a visiting research assistant at the University of Connecticut and visited the TDS group at MIT, focusing on distributed algorithms for dynamic environments. His Ph.D., obtained from Université de Rennes 1 and INRIA in 2007 presents distributed shared memories for large-scale dynamic systems. He also worked recently as a visiting scientist in the Distributed Systems group at Cornell University.

Peter M. Musial received Ph.D. degree from the University of Connecticut in 2007, with Prof. Alexander A. Shvartsman as the adviser. The topic of his doctoral dissertation is specification, refinement, and implementation of an atomic memory service for dynamic environments. He also worked as a research associate/developer for VeroModo, Inc. A company developing computer-aided tools for specification and analysis of complex distributed systems. In 2007, he joined the Naval Postgraduate School as a postdoctoral fellow though National Research Council program, with Prof. Luqi as the mentor. His research there concentrates on the development of methodologies that guide software development activates of complex systems, where the methodologies are based on analysis of requirements documentation and project risk assessment.

Alexander Shvartsman is a Professor of Computer Science and Engineering at the University of Connecticut. He received his Ph.D. in Computer Science from Brown University in 1992. Prior to embarking on the academic career, he worked for over 10 years at AT&T Bell Labs and Digital Equipment Corporation. His research in distributed computing has been funded by several NSF grants, including the NSF Career Award. Shvartsman is an author of over 100 papers, two books, and several book chapters. He chaired and he served on many program committees of the top conferences in distributed computing, and he is a Vigneron d'Honneur of Jurade de Saint-Emilion.

Please cite this article in press as: G. Chockler, et al., Reconfigurable distributed storage for dynamic networks, J. Parallel Distrib. Comput. (2008), doi:10.1016/j.jpdc.2008.07.007