J. Parallel Distrib. Comput. 69 (2009) 100–116

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 26 July 2008

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.

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 $\textit{update}(c, c') = \max(c, c')$ if c and c' are comparable (in the partial ordering of $C \cup \{\perp, \pm\}$), $\textit{update}(c, c') = c$ otherwise.
- *extend*, a binary function on $C \cup \{\perp, \pm\}$, defined by $\textit{extend}(c, c') = c'$ if $c = \perp$ and $c' \in C$, and $\textit{extend}(c, c') = c$ otherwise.
- *truncate*, a unary function on *CMap*, defined by $\textit{truncate}(cm)(k) = \perp$ if there exists $\ell \leq k$, such that $cm(\ell) = \perp$, $\textit{truncate}(cm)(k) = cm(k)$ otherwise. This truncates configuration map cm by removing all the configuration identifiers that follow a $\perp$.
- *Truncated*, the subset of *CMap* such that $cm \in \textit{Truncated}$ if and only if $\textit{truncate}(cm) = cm$.

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

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.

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, $pxs.conf\text{-}index + 1$ 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,

```
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.

Fig. 4. Send/receive/other transitions.

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