Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts

Nancy Lynch Alex Shvartsman

Massachusetts Institute of Technology, Laboratory for Computer Science,
545 Technology Square, NE43-365, Cambridge, MA 02139, USA.

December 2, 1996

Abstract

This paper presents *robust* emulation of *multi-writer/multi-reader* registers in message-passing systems using *dynamic quorum configurations*. In addition to processor and link failures, this emulation tolerates changes in quorum configurations, i.e., on-line replacements of one quorum system consisting of read and write quorums with another such system. This work extends the results of Attiya, Bar-Noy and Dolev [1] who showed how to emulate single-writer/multi-reader registers robustly in message-passing systems using majorities.

The emulation in this paper is specified using a modular two-layer architecture. The lower layer uses unreliable broadcast to disseminate a request from the higher layer to a set of processors, and then to collect responses from a subset of the processors. The subset can be specified by a predicate or by using a quorum system. The lower layer then computes a function on the collected responses and returns the result to the higher layer. The broadcast can take advantage of hardware-assisted broadcast as we do *not* assume that the broadcast is reliable or that it has FIFO, causal or atomic properties. The higher layer algorithm emulates robust multi-writer/multi-reader registers where *quorum configurations* are used to ensure that the registers are atomic.

A unique feature of the read/write service is that it implements *dynamically changing quorum configurations*. The service includes two interfaces, a *functional* interface for reads and writes, and a *management* interface for reconfiguration. The processor designated as the reconfigurer executes requests that replace the current quorum configuration with the new configuration. The combination of the higher and lower layers allows essentially unlimited concurrency and does not involve locks. Waiting can occur only (a) due to processor or link failures that disconnect at least one processor in each read and write quorum of the specified configurations, or (b) when frequent reconfigurations interfere with reads/writes and cause them to contribute to reconfigurations. However, as soon as reconfigurations stop, and as long as for each lower level request specifying a set of read or write quorums there exists a single quorum of active and connected processors, then reads and writes complete without waiting. All of this is transparent to the clients of the service.

The algorithms are specified here in terms of I/O automata [8, 9], and their correctness is proven using invariants and partial-order-based methods. It is shown that the algorithm is correct, and that it implements atomic replicated read/write objects.

Keywords : Distributed algorithms, fault-tolerance, atomic registers, message-passing, quorums.

Author e-mail : lynch@theory.lcs.mit.edu (primary contact), alex@theory.lcs.mit.edu.

Submission category : Regular paper.

Approximate word count : 9,500 (not counting optional appendices).

This material has been cleared through author affiliations.

1 Introduction

The two major multiprocessor computation paradigms are the shared-memory paradigm and the message-passing paradigm. Developing efficient algorithms that can tolerate component failures and timing delays for these models has been a goal for algorithm designers for a long time. It has been observed that in many cases it is easier to develop algorithms for the shared-memory model than for the message-passing model. Consequently, in such cases there is value in developing an algorithm first for the shared-memory model and then automatically converting it to run in the message-passing model. Among the important results in this area are the algorithms of Attiya, Bar-Noy and Dolev [1] who showed that it is possible to emulate shared memory robustly in message-passing systems. Their very interesting, fully asynchronous algorithm implements atomic single-writer/multi-reader registers in unreliable, asynchronous networks. Our work is inspired by and builds on their results.

In more detail, [1] shows that any wait-free algorithm for the shared-memory model that uses atomic single-writer/multi-reader registers can be emulated in the message-passing model where processors or links are subject to crash failures. The authors of [1] give a basic algorithm for complete networks using unbounded timestamps, a version for arbitrary network topologies, and they also modify their algorithms so that it uses only bounded timestamps. These algorithms are based on processor majorities and thus are able to tolerate scenarios where any minority of processors are disabled or are unable to communicate. The algorithms [1] are constructed with the help of a communicate procedure that uses half-duplex, ping-pong, point-to-point links to broadcast messages and to collect responses from any majority of processors. The basic algorithmic techniques are very efficient and they render the algorithm suitable for an effective implementation.

Using majorities is a special case of quorum systems [6]. A simple *quorum system* (also called *coterie*) is a collection of sets such that any two sets, called *quorums*, intersect [5]. A more refined approach divides the quorum system into a collection of read quorums and a collection of write quorums such that any read quorum intersects any write quorum. Such systems have been used to implement distributed mutual exclusion [5] and data replication protocols [4, 7]. Quorums can be used with replicated data in transaction-style synchronization that *limits* concurrency (cf. [2]), whereas our goal and the goal of [1] is to reduce restrictions on asynchrony and concurrency.

In this paper we present a service that emulates shared memory registers using broadcasts and dynamically changing quorum configurations. Our algorithms extend the unbounded-timestamp single-writer solution of Attiya, Bar-Noy and Dolev [1] in four ways:

1. Our construction emulates *multi-writer/multi-reader* registers.
2. We replace the majority-based approach of [1] with a *quorum-based* approach – this is done in a way that *does not* involve synchronization and that preserves the asynchrony and non-determinism found in the original solution [1].
3. We augment the multi-writer/multi-reader service with a *management interface* used to *reconfigure* the quorum system on-the-fly without changing the functional interface of the service and without suspending any reads/writes in progress or disabling new requests.
4. Our algorithm is defined in a modular way using a two-layer architecture; the lower level specifies a new general-purpose primitive that formalizes abstract acknowledged-broadcast computation.

Of the four extensions, the most technically challenging part of our work is the dynamic quorum system reconfigurations. We next cover in more detail the specific extentions and innovations. We split the complexity of the overall solution by specifying it in a modular fashion as a composition of two layers.

- The *lower layer* implements a computation primitive that we call Γ. The primitive uses *quorum-acknowledged broadcasts* and *condenser functions* to perform computations requested by the higher layer. The broadcast used by the lower layer can take advantage of hardware-assisted broadcast as we do *not* assume that the broadcast is reliable or that it has FIFO, causal or atomic properties.

  We specify two versions of the primitive, the simpler primitive $\Gamma(C)$ uses a globally-known static quorum configuration C, while Γ allows for the quorum configurations to be changed. Each primitive admits straightforward implementation using message-passing. Our use of the Γ primitive illustrates how computing with the Γ primitive can be an effective tool in developing distributed algorithms.

- The *higher layer* algorithm emulates robust multi-writer/multi-reader registers where quorum systems are used to ensure that the registers are atomic. This layer extends the single writer protocol [1] to a multi-writer protocol. We use quorum systems in a way that ensures the atomicity of the multi-writer/multi-reader registers without resorting to locking or mutual exclusion.

  A unique feature of this layer is that we deal with *dynamically changing quorum configurations*. In a static quorum system the same configuration is used regardless of changing load balancing or availability concerns. Our service exports two interfaces, a *functional* interface offering the read/write service, and a *management* interface for reconfiguration. The management interface designates one processor as the reconfigurer. This processor executes requests that replace the current quorum configuration with a new configuration using Γ. This does not involve any synchronization, but some read and write requests concurrent with a reconfiguration may need to perform steps that contribute to the reconfiguration. This is done transparently to the clients of the service.

The solution implemented by the composition of the two layers reflects practical system concerns dealing with *communication efficiency*, with *fault-tolerance* and with *system management* (i.e., with supervision and control of the system so that it fulfills the requirements of its users, cf. [16]).

Our service can be implemented by using point-to-point messages or by taking advantage of broadcast. In the network settings where processors closely cooperate, it is increasingly important to assume the availability of efficient broadcast or multicast. This assumption is reasonable for LAN-based environments and for emerging high-speed WANs. The availability of hardware-assisted broadcast [14, 3] makes the cost of using broadcast similar to the cost of sending a point-to-point message. Note that our algorithms do not require such broadcast to have atomic, FIFO, or causal properties.

Our robust emulation can tolerate a broad range of patterns of processor and link failures. The service is guaranteed to continue operation provided that the processor performing a service request is able to communicate with processors constituting some read quorum and some write quorum during a certain time interval. The duration of this interval must be sufficient to allow the completion of the individual invocations of Γ using the configurations containing these quorums. The actual quorums need not be the same for all invocations.

When a quorum system needs to be reconfigured, this is done using the management interface of our service without suspending or interrupting the read/write service provided to its clients via the functional interface. The successful deployment and use of complex distributed applications often depends on our ability to manage the application as a distributed resource on the basis of current and historical observations [15]. A resource manager can monitor the environment for changing performance requirements and availability conditions and, in our case, evolve the quorum system using the management interface of the service.

In achieving the above, we formally specify and analyze the algorithms for the multi-writer/multi-reader service. Our algorithms are specified in terms of I/O automata [8, 9]. We use *invariants* and *partial-order* based methods to prove that our algorithms are correct, and that it implements atomic replicated read/write objects. The main proof introduces a new *"Fill"* notion used to predict the acknowledgment vector of Γ invocations. This is an effective tool in reducing the complexity and size of proofs.

The *correctness* analysis assumes no bounds on message delivery times. We carry out conditional *performance* analysis by assuming that point-to-point messages are delivered and locally processed in bounded time d (unknown to the processors), or not delivered at all. In the absence of reconfigurations, and assuming that messages to and from a set of processors constituting at least one read and one write quorum are delivered, reads and writes take no more than time $4d$ using the current quorum system. Each reconfiguration takes no more than time $6d$, whether or not there are any concurrent writes or reads. When reconfigurations are encountered, the response time for writes and reads grows incrementally. In general, using a quorum system that is k versions older than the current system increases time by at most $2dk$, thus reads and writes take at most time $4d + 2dk$.

In our implementation, we assume the availability of unbounded counters, whereas Attiya, Bar-Noy and Dolev [1] also provide an implementation using bounded counters. We discuss this assumption at the end of the paper.

The rest of this paper is as follows. In Sec. 2 we define models and conventions. In Sec. 3 we present $\Gamma(C)$ and Γ primitives. In Sec. 4 we specify multi-writer/multi-reader service that uses $\Gamma(C)$. In Sec. 5 we give the reconfigurable algorithm using Γ, prove atomicity of the emulated registers, and assess the performance of the service. We conclude with a discussion in Sec. 6. Supporting material and proofs are given in the **optional** appendices.

2 The model of computation and conventions

The message-passing model of computation we use in this work is as follows. There are n processors with unique identifiers in the set *PID*. For simplicity we assume $PID = \{1 \ldots, n\}$, but we do not assume that the identifier set is compact. The processors communicate using *point-to-point* messages at the level of abstraction of the *network layer*, i.e., in normal operations, any two processors can send messages to each other, the delivery is unreliable, but the contents of messages are not corrupted. We assume no bounds on message delivery times – the algorithms must be asynchronous. In the cases where a message is sent to all processors, broadcast can be used. Such broadcast does not need to guarantee any atomic, FIFO, causal or any other such properties.

We use the following failure model:

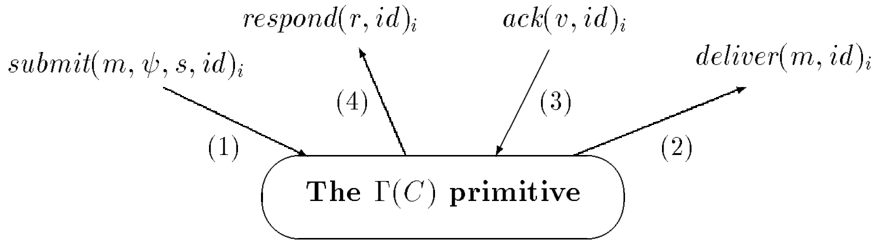

Figure 1: The model of the quorum-acknowledged broadcast primitive $\Gamma(C)$.

- Processors are subject to crash-failures and restarts. Processors do not lose their context due to failures. Such failures can be modelled as (possibly unbounded) delays.
- Link failures may render some of the nodes unreachable some of the time.
- In general we allow an adversary to cause arbitrary patterns of failures. When we assess performance of an algorithm we assume that when a response is expected from a quorum in some configuration, then the processors in at least one such quorum do respond.

In presenting distributed algorithms and showing the algorithms to be correct we make no assumptions about the length of time it takes for a message to be delivered or the amount of time it takes to perform a local computation. These assumptions will be made only for the purpose of evaluating the performance of the algorithms.

For the rest of the paper we define the following data types and conventions:

- $PID = \{1, 2, \ldots, n\}$, the set of processor unique identifiers
- $OID = \cup_{i \in PID} OID_i$, operation unique identifiers, where OID_i is the set of identifiers generated by processor i and for $i \neq j$ we have $OID_i \cup OID_j = \emptyset$
- M, the set of messages sent by processors
- A, the set of values sent and returned by processors
- A *condenser* is a function $\phi : (A \cup \{\perp\})^n \rightarrow A$; let Φ be the set of *condenser* functions
- $\mathcal{Q} = 2^{PID}$, the set of quorums
- $\mathcal{C} \subseteq \mathcal{Q}^* \times \mathcal{Q}^*$, the set of quorum configurations, each configuration has selectors *read* and *write*, such that if $C = \langle C.read, C.write \rangle \in \mathcal{C}$, where $C.read = \{R_1, \ldots, R_r\}$ and $C.write = \{W_1, \ldots, W_w\}$, then every R_i has a nonempty intersection with every W_j.

The specifications in this paper are done in terms of I/O automata [8, 9]. When a named value x is used in the code fragment of an action and the name is neither a part of the state, nor appears in the signature, we declare the type of such name using **Hidden**(x) notation.

3 The quorum-acknowledged broadcast primitive

We define two versions of the primitive, $\Gamma(C)$ which uses a fixed quorum configuration $C = \langle C.read, C.write \rangle$, and Γ which allows changing quorum configurations.

3.1 The $\Gamma(C)$ primitive

The model of the primitive is given in Figure 1. The primitive is invoked via the *submit* action (1) that contains the message m, the condenser function ψ and selector s which is either *read* or *write* to indicate whether to use read or write quorums of the globally known configuration C. The message is delivered to a processor via the *deliver* action (2), and the processor acknowledges the message by returning the value v via the *ack* action (3). The invoking processor applies the function ψ to a set of responses corresponding to an appropriate quorum at some point after these acknowledgements become available, and it returns the results to its client (4).

Below we state an abstract specification of $\Gamma(C)$, give an abstract implementation using send/receive channels and assess its performance.

The $\Gamma(C)$ primitive

Data-types:

$m \in M$ | Condenser function: $\psi \in \Phi$
$v \in A$ | Unique identifiers: $id \in OID$

Operation descriptors: $d, desc = \langle msg, con, sel, acc[1..n], dlv, rsp \rangle \in \mathcal{D}$,
where $\mathcal{D} = M \times \Phi \times \{read, write\} \times A^n \times 2^{PID} \times Bool$. The selectors are:

- msg: the message to be broadcast
- con: the condenser function
- sel: quorum type selector, either *read* or *write*
- $acc[1..n]$: array of accumulated acknowledgements, where n is the number of processors
- dlv: a set of member *id*s to whom the message was delivered
- rsp: a boolean indicating whether the submitter had responded to its client

Operations: $op \in \mathcal{O}$, where $\mathcal{O} = OID \rightarrow \mathcal{D} \cup \{\perp\}$

Actions of i:

Input: $submit(m, \psi, s, id)_i$, $ack(v, id)_i$

Output: $respond(r, id)_i$, $deliver(m, id)_i$

State:

$op \in \mathcal{O}$, initially empty, i.e., for any id, $op(id)$ is undefined

$C \in \mathcal{C}$, the fixed quorum configuration

Transitions of i:

$submit(m, \psi, s, id)_i$
Eff: $op(id) := \langle m, \psi, s, \perp^n, \emptyset, false \rangle$

$deliver(m, id)_i$
Pre: $op(id).rsp = false$
$i \notin op(id).dlv$
$op(id).msg = m$
Eff: $op(id).dlv := op(id).dlv \cup \{i\}$

$ack(v, id)_i$
Eff: $op(id).acc[i] := v$

$respond(r, id)_i$ Hidden($a[1..n] \in V^n, Q \in \mathcal{Q}$)
Pre: $op(id).rsp = false$
$Q \in C.(op(id).sel)$
$Q \subseteq \{k : op(id).acc[k] \neq \perp\}$
$\forall k \in Q : a[k] = op(id).acc[k]$
$\forall k \notin Q : a[k] = \perp$
Eff: $r := (op(id).con)(a)$
$op[id].rsp := true$

We assume that the clients of $\Gamma(C)$ adhere to the syntax of the specification. Furthermore, each *submit* is made unique by the invocation identifier *id*, and that any *ack* is issued only in response to a *deliver* and only once. The clients of the primitive have OID_i as a state component and they structure their output *submit* actions so that its precondition includes the conjunct "$id \in OID_i$" and its effect includes "$OID_i := OID_i - \{id\}$".

An *execution* α of an I/O automaton A is a finite or infinite sequence of alternating states and actions of A starting with the initial state. The *trace* of α, denoted by $trace(\alpha)$, is the subsequence of α consisting of all the external actions. Let α be an execution of $\Gamma(C)$ together with clients as above.

For a configuration C we have a lemma that follows from the properties of quorums:

Lemma 3.1 Suppose α is an execution of $\Gamma(C)$ together with its clients and $respond(\ldots, id_1)_i$ and $respond(\ldots, id_2)_j$ are two actions in α with $id_1 \neq id_2$. Suppose that α includes $submit(\ldots, write, id_1)_i$ and $submit(\ldots, read, id_2)_j$. Then there is an index k such that both $ack(\ldots, id_1)_k$ and $ack(\ldots, id_2)_k$ occur in α.

In Appendix A we present a straightforward implementation of the $\Gamma(C)$ primitive that we call $\Delta(C)$. The implementation uses send/receive point-to-point channels. Each channel is modelled having $send(m)_{i,j}$ and $recv(m)_{j,i}$ actions, and $channel_{i,j}$ state variables for $i, j \in PID$. Such channels have very simple specifications (cf. [8]) which are omitted here.

Lemma 3.2 The composition of $\Delta(C)$ and the channel automata implements $\Gamma(C)$.

The performance analysis is as follows:

Theorem 3.3 Suppose in any execution of $\Delta(C)$ (a) there is a fixed upper bound on local step time during which a processor reads all received messages, performs local computation, and sends any necessary replies, (b) for any delivered message, it is delivered after at most a known fixed delay, and (c) there exists a set of processors $Q \in C.s$ for s specified in any *submit* action such that they receive the request and their acknowledgements are delivered to the invoker of the *submit*, then it takes $O(1)$ time between the *submit* transition and the matching *respond* transition, and there are $\Theta(n)$ messages sent as the result of the *submit*.

3.2 The Γ primitive

The Γ primitive is an extension of the $\Gamma(C)$ primitive which does not rely on the fixed globally-known quorum configuration C. The single difference in the interfaces of the two primitives is that the *submit* action of $\Gamma(C)$ has the argument $s \in \{read, write\}$ indicating whether to use $C.read$ or $C.write$ quorums, while Γ has the argument $q \in \mathcal{Q}^*$ in which the client specifies the set of quorums to use.

The state of Γ does not include C, and definition of the operation descriptors is changed so that *sel* selector is replaced with $qrm \in \mathcal{Q}^*$. The qrm component is initialized to q in the effect of the *submit* action. The only remaining change is in the *respond* action, where in the precondition the conjunct "$Q \in C.(op(id).sel)$" is replaced with the conjunct "$Q \in op(id).qrm$".

The implementation $\Delta(C)$ can be similarly extended to produce the implementation Δ for the Γ primitive. It is not difficult to see that Lemma 3.1 and Lemma 3.3 given in the previous subsection equally apply to Γ and Δ.

4 Fixed quorums algorithm using $\Gamma(C)$

In this section we specify an algorithm for atomic multi-writer/multi-reader registers using a fixed quorum configuration and the $\Gamma(C)$ primitive. The algorithm specifies the higher layer and $\Gamma(C)$ the lower layer of the robust register emulation. We give a proof sketch of the algorithm correctness and of its performance analysis. The presentation illustrates the main algorithmic ideas and proof techniques used in the next section in the more complicated algorithm using dynamic quorum configurations.

4.1 Fixed quorums algorithm specification

In the approach of Attiya, Bar-Noy and Dolev [1], each copy of the register is stored together with a *label* used to order the writes and to determine the result of which write is returned by reads. In their single-writer approach the monotonically increasing label is maintained by the writer and is associated with the register. When the writer assigns a new value to the register along with the next higher label, it informs a majority of processors of the new value and label. Readers perform their operation by reading a majority of values and associated labels, selecting the value with the maximum label, and then informing a majority of processors of the value and the labels adopted by the write before returning the chosen value ot the client.

We generalize this approach by using a quorum configuration instead of majorities. Our solution is a pleasingly uniform algorithm for the readers and the (now multiple) writers. We replace the labels of [1] with the *tags* generated by the writers. The tags are pairs consisting of the sequence number *seq* and the processor identifier *pid*, and the tags are ordered lexicographically. Thus each register is represented locally at each processor by its value *val* and its tag *tag*. To simplify the presentation, we state the solution for one emulated register. Other than the interface, the only difference between the readers and the writers is that the writers assign new tags by incrementing the maximum tag found, while readers simply use the maximum tags.

The formal specification is given below. At a high level, the writer (reader) accepts a client *write* (*read*) request, invokes the $\Gamma(C)$ primitive by using the *submit* action to query all processors in a read quorum for their tags. When this *query* phase completes with the *respond* action, the writer lexicographically increments the maximum tag returned and then invokes the $\Gamma(C)$ primitive to propagate the new tag *prop-tag* and the new value *prop-val* to all processors in a write quorum. The reader simply propagates the maximum tag.

Each processor has two queues. The *request-q* maintains client read requests in the form $\langle$ *"read"*, $c\rangle$ and write requests in the form $\langle$ *"write"*, $v, c\rangle$, where c is the client identifier. Only the request at the head of the queue is processed at any given time. The second queue *ack-q* is used for acknowledgments to be sent out subsequently to the specific *deliver* transitions.

Reader/writer specification with fixed quorums

Data-types:

$\mathcal{T} = \mathcal{N} \times PID$, the tags of read and write operations with selectors *seq* and *pid*

$q \in \{read, write\}$, selector for the quorum configuration

L, client unique identifiers

(Other data-types as in Γ definition)

State: (for each processor $p \in PID$)

$tag \in \mathcal{T}$, initially $tag = \langle seq, pid\rangle = \langle 0, 0\rangle$

$val \in A$, initially $val = v_0 \in A$

prop-tag $\in \mathcal{T}$, tag used in propagating results, initially *prop-tag* $= \langle 0, 0\rangle$
prop-val $\in A$, initially undefined
status $\in$ { *query-ready, query-active, prop-ready, prop-active, prop-done* }, initially *idle*
request-q, a sequence of $(\{\text{"}read\text{"}\} \times L) \cup (\{\text{"}write\text{"}\} \times A \times L)$, queue of requests, initially empty
ack-q, a sequence of $M \times ID$, initially empty

Condenser functions:
$\sigma \equiv \lambda\,(a).(\langle a[k].val, a[k].tag\rangle : \forall j : a[k].tag \geq a[j].tag)$: maximum tag computation.

Actions:

Input:
$write(v)_{c,p}$
$read_{c,p}$
$respond(\langle \text{"query-ack"}, v, t\rangle, id)_p$
$respond(\langle \text{"prop-ack"}\rangle, id)_p$
$deliver(\langle \text{"query"}\rangle, id)_p$
$deliver(\langle \text{"propagate"}, v, t\rangle, id)_p$

Output:
$read\text{-}confirm(v)_{c,p}$
$write\text{-}confirm_{c,p}$
$submit(\langle \text{"query"}\rangle, (\text{"query-ack"}, \sigma), q, id)_p$
$submit(\langle \text{"propagate"}, v, t\rangle, (\lambda\,(a).(\text{"prop-ack"})), q, id)_p$
$ack(m, id)_p$

Transitions:

$write(v)_{c,p}$
Eff: append $\langle \text{"write"}, v, c\rangle$ to *request-q*

$read_{c,p}$
Eff: append $\langle \text{"read"}, c\rangle$ to *request-q*

$submit(\langle \text{"query"}\rangle, (\text{"query-ack"}, \sigma), q, id)_p$
Pre: *status* = *query-ready*
request-q $\neq \emptyset$
$q = read$
Eff: *status* := *query-active*

$respond(\langle \text{"query-ack"}, v, t\rangle, id)_p$ Hidden$(u \in A)$
Eff: **if** head(*request-q*) $= \langle \text{"write"}, u, c\rangle$ **then**
prop-val := u;
prop-tag := $\langle t.seq + 1, p\rangle$
else
prop-val := v;
prop-tag := t
status := *prop-ready*

$submit(\langle \text{"propagate"}, v, t\rangle, (\lambda\,(a).(\text{"prop-ack"})), q, id)_p$
Pre: *status* = *prop-ready*
$q = write$
v = *prop-val*
t = *prop-tag*
Eff: *status* := *prop-active*

$respond(\langle \text{"prop-ack"}\rangle, id)_p$
Eff: *status* := *prop-done*

$read\text{-}confirm(v)_{c,p}$
Pre: v = *prop-val*
status = *prop-done*
head(*request-q*) $= \langle \text{"read"}, c\rangle$
Eff: *request-q* := tail(*request-q*)
status := *query-ready*

$write\text{-}confirm_{c,p}$ Hidden(v)
Pre: *status* = *prop-done*
head(*request-q*) $= \langle \text{"write"}, v, c\rangle$
Eff: *request-q* := tail(*request-q*)
status := *query-ready*

$deliver(\langle \text{"query"}\rangle, id)_p$
Eff: append $\langle\langle \text{"query-ack"}, val, tag\rangle, id\rangle$ to *ack-q*

$deliver(\langle \text{"propagate"}, v, t\rangle, id)_p$
Eff: **if** $t >_{lex} tag$ **then**
$val := v$; $tag := t$
append $\langle\langle \text{"prop-ack"}\rangle, id\rangle$ to *ack-q*

$ack(m, id)_p$
Pre: head(*ack-q*) $= \langle m, id\rangle$
Eff: *ack-q* := tail(*ack-q*)

We now define conventions that in the rest of the paper are used to identify client-level read and write operations. We use variable π (appropriately subscripted when necessary) to uniquely identify the client-level operations.

Client-level read and write operations contain the *query* and *propagation* phases in each of which the $\Gamma(C)$ primitive is invoked once for the case of the fixed quorum configuration. The first phase uses read quorums $C.read$ and the second uses the write quorums $C.write$.

Definition 4.1 The phases of the read or write operation π are defined as follows:

1. The operation π is in its *query* phase after the transition of the *submit* of *"query"* and prior to the *submit* of *"propagate"*.
2. The operation π is in its *propagate* phase after the transition of the *submit* of *"propagate"* and prior to the response to its client. $\square$

In a given execution α we say that a read (write) operation π *propagates* a tag if the tag is used in the *submit* action in the propagation phase of π. We denote by $\tau_\alpha(\pi)$ the tag propagated by operation π. Where α is clear from the context we omit it and use $\tau(\pi)$.

The invocation event of a client-level read (write) operation is its corresponding *read* (*write*) action. The response event of the read (write) operation is its corresponding *read-confirm* (*write-confirm*) action.

Suppose for some execution α the actions of an operation π include the actions of the $\Gamma(C)$ primitive for some id, starting with the *submit* action and including the *respond* action. The unique identifier id also uniquely identifies the client-level operation π. When the propagation tag $\tau_\alpha(\pi)$ is defined for an operation π in the *respond* action uniquely identified by id_1, or when $\tau_\alpha(\pi)$ is propagated by the $\Gamma(C)$ primitive using unique identifier id_2, then we also let $\tau(id_1)$ or $\tau(id_2)$ stand for $\tau_\alpha(\pi)$.

For the client-level operations in an execution we define relation *CP*, the *client-preceding* order as follows:

Definition 4.2 If in an execution α, any Γ invoked in the operation π_1 completes before any Γ is invoked in the operation π_2 (i.e., π_1 completes before π_2 starts), then $\langle \pi_1, \pi_2 \rangle \in CP$. Where convenient, we use the notation $\pi_1 \prec_{cp} \pi_2$ to indicate the same. $\square$

The *CP* relation can be dynamically maintained as a history variable – for the purpose of the proofs only. This is done by maintaining a set *completed* of operations that is initially empty, and by adding an operation to it at the point of its completion, i.e., setting *completed* to *completed* $\cup\{\pi\}$, where π is the operation just being completed. We dynamically construct *CP* by setting *CP*, upon the start of a new operation π, to *CP* $\cup\{(\pi', \pi) : \pi' \in$ *completed*$\}$. Note that these derived variables are otherwise not used in any way by the algorithm.

4.2 Proof of correctness

For any execution we are interested in showing the atomicity of the read and write operations. We show atomicity of the implementation by using the following lemma of [8]:

Lemma 4.1 [8] Let β be a (finite or infinite) sequence of actions of a read/write object external interface. Suppose that β is well-formed for each $i \in PID$, and contains no incomplete operations. Let Π be the set of all operations in β. Suppose that $\prec$ is an irreflexive partial ordering of all the operations in Π, satisfying the following properties:

1. For any operation $\pi \in \Pi$, there are only finitely many operations ϕ such that $\phi \prec \pi$.
2. If the response event for π precedes the invocation event for ϕ in β, then it cannot be the case that $\phi \prec \pi$.
3. If π is a *write* operation and ϕ is any operation in Π, then either $\pi \prec \phi$ or $\phi \prec \pi$.
4. The value returned by each *read* operation is the value written by the last preceding *write* operation according to $\prec$ (or v_0, the initial value, if there is no such *write*).

Then β satisfies the atomicity property. $\square$

This lemma lists four conditions involving a partial order on operations in β. If an ordering satisfying these four conditions exists, it is guaranteeing that there is some way to insert serialization points satisfying the atomicity property. Condition 1 rules out orderings in which infinitely many operations precede some particular other operation. Condition 2 says that the $\prec$ ordering must be consistent with the order of invocations and responses by the clients. Condition 3 says that $\prec$ totally orders the *write* operations and orders all the *read* operations with respect to the *write* operations. Condition 4 says that the responses to *read*s are consistent with $\prec$.

We now present the proof of atomicity of the registers implemented by the composition of the fixed quorum algorithm and $\Gamma(C)$. We proceed with preliminary lemmas that lead to the main result (a selection of proofs is in the optional Appendix B).

It is easy to see that since tags are only changed in the effects of *deliver* actions where tags are lexically increased, we have:

Lemma 4.2 Tags maintained by each processor are monotonically nondecreasing.

Each read and write operations include exactly two sequential invocations of the $\Gamma(C)$ primitive. The first invocation uses read quorums and the second uses the write quorums.

Lemma 4.3 If for an operation π, t is the tag returned by the *query* phase of the algorithm and $\tau(\pi)$ is the tag used in the *propagation* phase, then (i) if π is a read then $t = \tau(\pi)$, and (ii) if π is a write then $t < \tau(\pi)$.

Now the main supporting lemma:

Lemma 4.4 If in an execution α, $\pi_1 \prec_{cp} \pi_2$, then (i) if the operation π_2 is a read, then $\tau_\alpha(\pi_1) \leq_{lex} \tau_\alpha(\pi_2)$, and (ii) if the operation π_2 is a write, then $\tau_\alpha(\pi_1) <_{lex} \tau_\alpha(\pi_2)$.

We now define the partial order needed to apply Lemma 4.1 in the main theorem for fixed configurations as follows: Let β be any sequence of read and write operations Π containing no incomplete operations. We define the (irreflexive) partial order $PO = \langle \Pi, \prec \rangle$ on the operations by letting: $\pi_1 \prec \pi_2$ for $\pi_1, \pi_2 \in \Pi$, if (a) $\tau(\pi_1) <_{lex} \tau(\pi_2)$, or (b) π_1 is a write and π_2 is a read such that $\tau(\pi_1) =_{lex} \tau(\pi_2)$.

The following theorem is shown with the help of Lemma 4.1:

Theorem 4.5 β satisfies the atomicity property.

4.3 Conditional performance analysis

To assess the performance of the atomic multi-writer/multi-reader service, we assume that for any invocation of $\Gamma(C)$ the invoker does not fail, and that it receives responses from at least one quorum of processors in C. We also assume that d is an upper bound for the longest message delivery delay (when message is indeed delivered), plus local processing of the message, and sending any replies. In addition, it is assumed that processors that have enabled transitions continue taking steps. With these assumptions, and using Theorem 3.3 (recall its assumptions) we show the following:

Theorem 4.6 Any read or write operation takes time $4d$ and at most $8n$ messages.

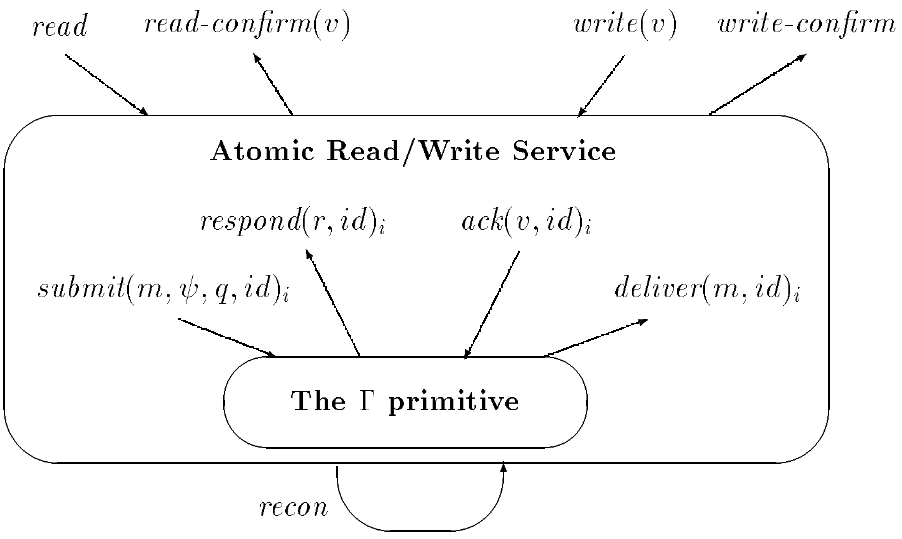

Figure 2: Two-layer modular view of the read/write service with reconfigurable quorums.

5 Reconfigurable quorums algorithm using Γ

In this section we specify the multi-writer/multi-reader algorithm with quorum reconfigurations using the Γ primitive. We then prove that the algorithm correctly implements atomic multi-reader/multi-writer registers and assess its conditional performance. A high-level modular representation of the service is given in Figure 2.

5.1 Reconfigurable quorums algorithm specification

To extend the fixed quorum algorithm to reconfigurable quorums, we need to solve the problems of (1) informing active processors of new configurations, (2) ensuring that it is safe to stop using an older configuration in favor of the new one, and (3) ensuring that any processors that attempt to use any of the obsolete configurations are able to obtain the current configuration.

Achieving this is technically challenging for several reasons. We do not assume availability of reliable broadcast or channels, thus not all processors may learn of the existence of a new configuration. Furthermore, since we allow processor restarts, and restarted processors may have their configurations *arbitrarily* out of date. We need a distributed solution which does not rely on the availability of the reconfigurer to dispense current quorum configurations to processors with obsolete configurations. The solution also has to allow concurrency in the presence of reconfigurations without resorting to locking or mutual exclusion.

We give specification in two parts. In the first part we give common data-types and the transitions of reader/writer. In the second part we define the transitions of the reconfigurer.

Read and write operations consist of two phases in which the Γ primitive is invoked at least once. The first phase, *query*, uses read quorums and the second, *propagation*, the write quorums. Definitions 4.1 and 4.2 are the same for client operations with reconfigurations. The fact the queries and propagations may involve more than one invocation of Γ has no impact on the meaning of these definitions.

As in the case with fixed configurations, $\tau(\pi)$ denotes the tag propagated by operation π.

Similarly, the main difference between reads and writes is that in the case of reads, the value with the associated maximum tag are propagated, and in case of writes, the new value and the lexicographically incremented tag are propagated.

A reader/writer maintains configuration index pair cix and configuration pair Cfg. These are such that $cix.act$ is the index of the active current configuration, $cix.bid$ is the index of the proposed configuration, and $Cfg.Act$ is the active current configuration and $Cfg.Bid$ is the proposed configuration. When $cix.act = cix.bid$, it means that $Cfg.Act = Cfg.Bid$, and that the proposed configuration is accepted as active. Note that the configuration index pairs can be compared lexicographically.

The *query* and *propagation* phases of readers/writers are similar to the phases of the fixed quorum algorithm, but include a possible iteration. Each phase invokes Γ until the response returns an index pair that contains no higher configuration index than the index of the invoker. If a higher active index is detected, it is adopted and the primitive is invoked using the configuration with the higher index. When a reader/writer uses the current configuration, the processing is essentially identical to the fixed configuration case. Perhaps surprisingly, the configurations used in the *query* and *propagation* phases need not be the same!

Reconfigurable quorums algorithm

Data-types:

The set of configuration indices: $\mathcal{I}^2$, with selectors act, the active configuration number, and bid, the proposed configuration number

Configuration indices: $x, z, cix =\in \mathcal{I}^2$

The set of configuration pairs: $\mathcal{C}^2$, with selectors Act, the active configuration, and Bid, the proposed configuration

Configuration pairs: $X, Z \in \mathcal{C}^2$

The values returned in the acknowledgements of the query phase (and accumulated in $op(id)acc[1..n]$ by Γ) are of the type $M \times A \times \mathcal{T} \times \mathcal{I}^2 \times \mathcal{Q}^2$. The selectors for each component is as follows:

$msg \in M$, the message type of *"query-ack"*
$val \in A$, the data object value
$tag \in T$, the tag of the object
$cix \in \mathcal{I}^2$, the configuration index pair
$cfg \in \mathcal{C}^2$, the quorum configuration pair

Condenser functions:

$\sigma \equiv \lambda\,(a).(\langle a[k].val, a[k].tag\rangle : \forall j : a[k].tag \geq a[j].tag)$: maximum tag (same as for fixed configurations)

$\xi \equiv \lambda\,(a).(\langle a[k].cix, a[k].cfg\rangle : \forall j : a[k].cix \geq a[j].cix)$: maximum configuration index and associated configuration

State of the reader/writer: (for each processor $p \in PID$)

The state components are the same as for the fixed quorums algorithm, but with the following additions:

$cix \in \mathcal{I}^2$, the configuration index pair, initially $\langle 0, 0\rangle$

$Cfg \in \mathcal{C}^2$, the configuration pair, initially $\langle C_0, C_0\rangle$, for some C_0.

$next\text{-}config : \mathcal{C}$, a generator of configurations.

Actions of the reader/writer:

(The actions *write*, *read*, *write-confirm* and *read-confirm* are identical to their counterparts in the fixed quorums specification and we do not repeat them here.)

Inputs:
$respond(\langle \text{“}query\text{-}ack\text{”},\ v, t, z, Z\ \rangle,\ id)_p$
$respond(\langle \text{“}prop\text{-}ack\text{”},\ z, Z\ \rangle,\ id)_p$
$deliver(\langle \text{“}query\text{”}\ \rangle,\ id)_p$
$deliver(\langle \text{“}propagate\text{”},\ v, t\ \rangle,\ id)_p$
$deliver(\langle \text{“}recon\text{-}done\text{”},\ z, Z\ \rangle,\ id)_p$

Outputs:
$submit(\langle \text{“}query\text{”}\ \rangle,\ (\text{“}query\text{-}ack\text{”}, \sigma, \xi),\ q, id)_p$
$submit(\langle \text{“}propagate\text{”},\ v, t\ \rangle,\ (\lambda\,(a).(\text{“}prop\text{-}ack\text{”}),\ \xi),\ q, id)_p$
$ack(m, id)_p$

Transitions of the reader/writer:

$submit(\langle \text{“}query\text{”}\ \rangle,\ (\text{“}query\text{-}ack\text{”},\ \sigma, \xi),\ q, id)_p$
Pre: $status = query\text{-}ready$
$request\text{-}q \neq \emptyset$
$q = Cfg.Act.read$
Eff: $status := query\text{-}active$
$cix\text{-}used := cix.act$
$cfg\text{-}used := Cfg.Act$

$respond(\langle \text{“}query\text{-}ack\text{”},\ v, t, z, Z\ \rangle,\ id)_p$
Eff: **if** $cix\text{-}used \geq z.bid$ **then**
  if head($request\text{-}q$)=$\langle \text{“}write\text{”}, u, c\ \rangle$ **then**
    $prop\text{-}val := u;$
    $prop\text{-}tag := \langle t.seq + 1, p\rangle$
  else
    $prop\text{-}val := v;$
    $prop\text{-}tag := t$
  $status := prop\text{-}ready$
else
  if $z > cix$ **then**
    $cix := z; \quad Cfg := Z$
  $status := query\text{-}ready$

$deliver(\langle \text{“}query\text{”}\ \rangle,\ id)_p$
Eff: append $\langle\langle \text{“}query\text{-}ack\text{”},\ val, tag, cix,\ Cfg\,\rangle,\ id\,\rangle$ to $ack\text{-}q$

$deliver(\langle \text{“}propagate\text{”},\ v, t\ \rangle,\ id)_p$
Eff: **if** $t >_{lex} tag$ **then** $val := v; tag := t$
append $\langle\langle \text{“}prop\text{-}ack\text{”},\ cix,\ Cfg\,\rangle,\ id\,\rangle$ to $ack\text{-}q$

$deliver(\langle \text{“}query\text{-}install\text{”},\ z, Z\ \rangle,\ id)_p$
Eff: append $\langle\langle val, tag, cix,\ Cfg\rangle,\ id\,\rangle$ to $ack\text{-}q$
if $z > cix$ **then** $cix := z; Cfg := Z$

$submit(\langle \text{“}propagate\text{”},\ v, t\ \rangle,\ (\lambda\,(a).(\text{“}prop\text{-}ack\text{”}),\ \xi),\ q, id)_p$
Pre: $status = prop\text{-}ready$
$q = Cfg.Act.write$
$v = prop\text{-}val$
$t = prop\text{-}tag$
Eff: $status := prop\text{-}active$
$cix\text{-}used := cix.act$
$cfg\text{-}used := Cfg.Act$

$respond(\langle \text{“}prop\text{-}ack\text{”},\ z, Z\ \rangle,\ id)_p$
Eff: **if** $cix\text{-}used \geq z.bid$ **then**
  $status := prop\text{-}done$
else
  if $z > cix$ **then**
    $cix := z; \quad Cfg := Z$
  $status := prop\text{-}ready$

$deliver(\langle \text{“}recon\text{-}done\text{”},\ z,\ Z\,\rangle,\ id)_p$
Eff: **if** $z > cix$ **then** $cix := z; Cfg := Z$
append $\langle\langle \text{“}prop\text{-}ack\text{”}\ \rangle,\ id\,\rangle$ to $ack\text{-}q$

$ack(m, id)_p$
Pre: head($ack\text{-}q$) $= \langle m,\ id\,\rangle$
Eff: $ack\text{-}q :=$ tail($ack\text{-}q$)

The reconfigurer has three phases. Each phase consist of a single invocation of Γ. In the *query-install* phase it informs a read quorum and a write quorum in current configuration of the new configuration and it obtains the register value with the maximum tag it found. In the *propagate* phase it propagates this tag and value to a quorum in the new configuration. In the *recon-idle* phase it announces the reconfiguration complete.

Definition 5.1 The phases of the reconfigurer are defined as follows:

1. The reconfigurer is in its *query-install* phase after the transition of the *submit* of *"query-install"* and prior to the *submit* of *"propagate"*.
2. The reconfigurer is in its *propagate* phase after the transition of the *submit* of *"propagate"* and prior to the *submit* of *"recon-done"*.
3. The reconfigurer is in its *recon-idle* phase after the transition of the *submit* of *"recon-done"* and prior to the *submit* of *"query-install"*. The reconfigurer is also in its *"recon-idle"* phase prior to the *submit* of the very first *"query-install"*. □

The reconfigurer r maintains the current quorum configuration sequence number $cix.act_r$ and the current configuration $Cfg.Act_r$. In any global state, the *current* configuration index is defined to be $cix.act_r$. For any processor p, its configuration is *current*, if $cix.act_p = cix.act_r$.

State of the reconfigurer r:

The state components are the same as for the reader/writer above, except that the *request-q* component is *deleted*

Actions of the reconfigurer:

(The *deliver* and *ack* actions are identical to the actions of readers/writers)

Inputs:
$recon_r$
$respond(\langle \text{"install-ack"}, v, t, z, Z \rangle, id)_r$
$respond(\langle \text{"prop-ack"}, z, Z \rangle, id)_r$
$respond(\langle \text{"recon-ack"} \rangle, id)_r$

Outputs:
$submit(\langle \text{"query-install"}, z, Z \rangle, (\lambda(a).(\text{"install-ack"}), \sigma, \xi), q, id)_r$
$submit(\langle \text{"propagate"}, v, t \rangle, (\lambda(a).(\text{"prop-ack"}), \xi), q, id)_r$
$submit(\langle \text{"recon-done"}, z, Z \rangle, \lambda(a).(\text{"recon-ack"}), q, id)_r$

Transitions of the reconfigurer:

$recon_r$
Pre: $status = idle$
Eff: $Cfg.Bid := next\text{-}config$
$cix.bid := cix.act + 1$
$status := new\text{-}config$

$submit(\langle \text{"query-install"}, z, Z \rangle, (\lambda(a).(\text{"install-ack"}), \sigma, \xi), q, id)_r$
Pre: $status = new\text{-}config$
$z = cix \wedge Z = Cfg$
$q = Cfg.Act.read \bowtie Cfg.Act.write$ (see note)
Eff: $cix := z$; $Cfg := Z$
$status := query\text{-}active$

$respond(\langle \text{"install-ack"}, v, t, z, Z \rangle, id)_r$
Eff: $prop\text{-}val := v$; $prop\text{-}tag := t$
$status := query\text{-}done$

$submit(\langle \text{"propagate"}, v, t \rangle, (\lambda(a).(\text{"prop-ack"}), \xi), q, id)_r$
Pre: $status = query\text{-}done$
$v = prop\text{-}val \wedge t = prop\text{-}tag$
$q = Cfg.Bid.write$
Eff: $status := prop\text{-}active$

$respond(\langle \text{"prop-ack"}, z, Z \rangle, id)_r$
Eff: $status := propagate\text{-}done$

$submit(\langle \text{"recon-done"}, z, Z \rangle, \lambda(a).(\text{"recon-ack"}), q, id)_r$
Pre: $status = propagate\text{-}done$
$z = \langle cix.bid, cix.bid \rangle$
$Z = \langle Cfg.Bid, Cfg.Bid \rangle$
$q = Cfg.Bid.write$
Eff: $cix := z$
$Cfg := Z$

$respond(\langle \text{"recon-ack"} \rangle, id)_r$
Eff: $status := idle$

The *deliver* and *ack* actions are identical to the actions of readers/writers.

Note: For $A, B \in \mathcal{Q}$, we define $A \bowtie B$ as the set $\{a \cup b : a \in A \wedge b \in B\}$.

5.2 Correctness of the composed automaton

We now consider the composition of the multi-reader/multi-writer automaton, the reconfigurer automaton and the Γ primitive.

In showing the correctness of the composed automaton, we introduce a succinct and effective way of expressing the eventuality of certain outcomes based on the current knowledge. The proof uses a new "*Fill*" notion, which we use to predict the acknowledgment vector for a current invocation. This notion can be used to great advantage in stating our invariants and in reducing the size of their proofs.

Our *Fill* notion produces a "virtual" acknowledgment from each processor based on taking the actual acknowledgment if it is already defined, else a predicted acknowledgment determined as follows. If a *deliver* has occurred at p without the corresponding *ack*, then the queued acknowledgment; if the *deliver* has not occurred, then the acknowledgment that would be produced if the deliver occurred as the next event.

Formally,

Definition 5.2 For the invocation of the Γ primitive with the unique identifier id, let $\mu_p : M \times State \rightarrow M$ be the function computed in the effects of the *deliver* action by processor p to construct the acknowledgment message upon the receipt of a message from the *submit*-er, we define:

$$
\begin{array}{ll}
\mathit{Fill}(p, id) \equiv & \textbf{if } op(id) = \perp \\
& \textbf{then } \perp \\
& \textbf{else if } p \in op(id).acks \\
& \quad \textbf{then } op(id).acc[p] \\
& \quad \textbf{else if } \exists \langle m, id \rangle \in ack\text{-}q_p \\
& \quad\quad \textbf{then } m \\
& \quad\quad \textbf{else } \mu_p(m, state_p)
\end{array}
\quad \Box
$$

We now show the atomicity of the implementation using Lemma 4.1. In the rest of this section we state the most important lemmas and theorems. The detailed proofs are given in Appendix C. The numbering below preserves the numbering given in Appendix C.

The key to the proof is a multi-part invariant, which we present just below. Part **I3** is the most important part; it mirrors Lemma 4.4 of the algorithm with fixed quorum configurations as it relates the tags of operations where one follows another. Parts **I1** and **I2** are auxiliary invariants.

Parts **I1a,b,c** deal with the properties of the tags of completed operations and the state of the reconfiguration. Part **I1a** states that for any completed read or write operation π, if no new quorum system is being processed by the reconfigurer, then there exists a current write quorum such that all processors in it reflect either π or some other operation that supercedes it.

Part **I1b** states that if the reconfigurer invoked Γ to install a new configuration, then no matter what active read quorum it ends up using, it is guaranteed to obtain a tag that is at least as large as the tag of any completed operation. This guarantee is expressed using the *Fill* notation.

Part **I1c** states that if the reconfigurer invoked Γ to propagate the maximum tag it found to a new write quorum, then this tag is as high as the tag of any completed operation and any processors that have acknowledged the propagated tag have updated their own tags.

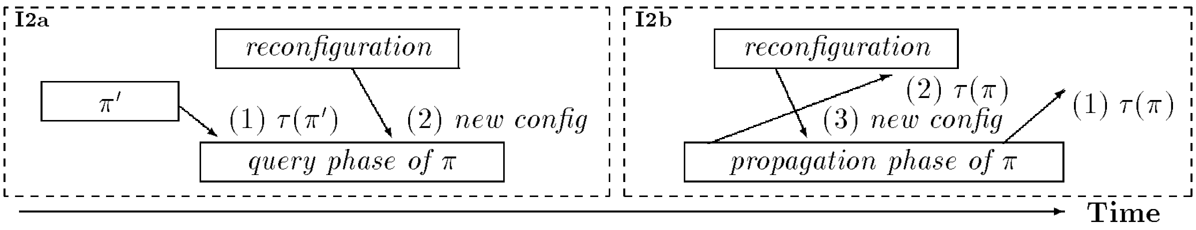

Figure 3: Invariant illustration for parts **I2a** and **I2b**.

Part **I2a** states that for any read or write in its *query* phase, either (1) the tag returned by the *query* is guaranteed to be at least as high as the tag of any completed operation – this is expressed with the help of the *Fill* notation, or (2) the operation detects that its configuration is obsolete – the guarantee of detection is expressed using *Fill*. See Figure 3.

Part **I2b** states that for any read or write operation π in its *propagation* phase, then at least on of the following conditions is guaranteed to hold: (1) its propagation tag is either being propagated using the current configuration, or (2) the tag is already reflected in a write quorum of the new configuration, or (3) π detects that its configuration is obsolete – again this guarantee of detection is expressed using *Fill*. See Figure 3.

Part **I3** is the key part of the invariant. It states that a read completely following another operation has the tag that is at least as large, and that a write has the tag strictly larger than any other operation that precedes it.

Lemma 5.14 In all reachable states:

I1 $\forall \pi \in completed$,
- (a) if the reconfigurer is in its *recon-idle* phase :
  $\exists W \in Cfg.Act.write_r : \forall i \in W : \tau(\pi) \leq tag_i$
- (b) if the reconfigurer is in its *query-install* phase having invoked Γ using identifier oid_r:
  $\forall R \in Cfg.Act.read_r : \tau(\pi) \leq \max_{i \in R}\{Fill(i, oid_r).tag\}$
- (c) if the reconfigurer is in its *propagate* phase having invoked Γ using identifier oid_r and the tag $\tau(recon)$: $(\tau(\pi) \leq \tau(recon)) \wedge (\forall i \in op(oid_r).acks : \tau(recon) \leq tag_i)$

I2 $\forall \pi \notin completed$,
- (a) If $\pi' \prec_{cp} \pi$ and π at processor p is in the *query* phase having invoked Γ using identifier oid, then for any $R \in cfg\text{-}used.read_p$, then
  (1) $\tau(\pi') \leq \max_{i \in R}\{Fill(i, oid).tag\}$, **or**
  (2) $cix\text{-}used_p < \max_{i \in R}\{Fill(i, oid).cix.bid\}$.
- (b) If π is in the *propagation* phase having invoked Γ using identifier oid, then
  (1) $cix\text{-}used_p$ is *current*, **or**
  (2) $\exists W \in Cfg.Act.write_r : \forall i \in W : \tau(\pi) \leq tag_i$, **or**
  (3) $\forall W \in cfg\text{-}used.write_p : cix\text{-}used_p < \max_{i \in W}\{Fill(i, oid).cix.bid\}$.

I3 If $\pi_1 \prec_{cp} \pi_2$ and $\tau(\pi_2)$ is defined, then
- (a) $\tau(\pi_1) \leq \tau(\pi_2)$ when π_2 is a read,
- (b) $\tau(\pi_1) < \tau(\pi_2)$ when π_2 is a write.

The proof of the lemma is by induction on the length of any execution of the composed automata (Appendix C).

Lemma 5.15 In any execution, if $\pi_1 \prec_{c.p.} \pi_2$, then (i) if π_2 is a read operation, then $\tau(\pi_1) \leq_{lex} \tau(\pi_2)$, and (ii) if π_2 is a write operation, then $\tau(\pi_1) <_{lex} \tau(\pi_2)$.

Proof: Using Lemma 5.14(I3) and Lemma 5.2. □

We now prove the atomicity of the register implementation similarly to the proof of the fixed quorums implementation by constructing a partial order and using Lemma 4.1.

Let β be Π containing no incomplete operations. We define the (irreflexive) partial order $PO = \langle \Pi, \prec \rangle$ on the operations by letting: $\pi_1 \prec \pi_2$ for $\pi_1, \pi_2 \in \Pi$, if

(a) $\tau(\pi_1) <_{lex} \tau(\pi_2)$, or

(b) π_1 is a write and π_2 is a read such that $\tau(\pi_1) =_{lex} \tau(\pi_2)$.

Theorem 5.16 Any such sequence of read and write operations β satisfies the atomicity property.

Proof: Follows the proof of Theorem 4.5. □

5.3 Conditional performance analysis

To assess the performance of the atomic multi-writer/multi-reader service, we make the same assumptions as in Section 4.3. With these assumptions, and adapting Theorem 3.3 for reconfigurable quorums, we show the following:

Theorem 5.1 In the absence of reconfigurations, any client-level read or write operation takes (a) time $4d$ if it starts with the current configuration, and (b) time $4d + (\mathit{current} - \mathit{cix\text{-}used}) \cdot 2d$ if it starts in the configuration *cix-used*.

The performance of any reconfiguration does not depend on any concurrent client-level-operations:

Theorem 5.2 Any reconfiguration takes time at most $6d$ and at most $12n$ messages.

6 Discussion

We have presented a robust service that emulates atomic multi-writer/multi-reader register in message passing systems. The service ensures atomicity of the emulated registers by relying on quorum systems in a way that allows great deal of asynchrony, concurrency and fault-tolerance. The service also allows for the quorum systems to be evolved dynamically, for example in response to changing operating conditions. This dynamic changes do not require any synchronization and the performance of the atomic register service is degraded gracefully when reconfigurations are frequent.

On manageability of distributed services: One of the problems often encountered in deploying distributed systems is that they are difficult to manage – many resource come without sufficient management facilities and require either manual intervention or else are equipped with management interfaces that are either inadequate or require out-of-band communication. Although the management interface provided by our service solves a narrowly focused management problem, it gives a good example of clean integration of functional and management aspects of the service. In particular, we require no out-of-band communication or reliance on fixed external quorum systems – the reconfiguration is achieved by using exclusively the native communication primitives and the quorum system that is being changed!

On efficient atomic read/write registers and bounded sequence numbers: Our algorithms assume the availability of unbounded counters used to number register versions and quorum configurations. The single-writer algorithm of Attiya, Bar-Noy and Dolev [1] is refined by the authors to use bounded counters at a modest increase in storage in message sizes. The implementation of [1] relies on a reliable *ping-pong* mechanism. This is done to allow, in a particular section of the protocol, only a single unacknowledged message between any two processors. Furthermore, any link is assumed to be reliable unless it crashes, after which the link remains forever inoperable.

It appears that such reliable *ping-pong* mechanism would assume too much reliability on the part of the communication subsystem. We conjecture that either the underlying subsystem may need itself either to use unbounded counters or to use messages of unbounded size (cf. the result [8, Thm. 22.11] due to Lynch, Mansour and Fekete).

On failure models considered: We have considered only the benign component failures – the processor and link failures never create spontaneous messages and the messages that are sent are delivered without alteration. Malki and Reiter [10] recently explored the use of quorum systems in the presence of Byzantine failures. It would be interesting to examine additional failure models that can be handled by atomic register emulations.

Optimizing the communication efficiency of accessing quorum systems: In our solution we use unreliable broadcast (or simulated broadcast) to achieve substantial asynchrony, concurrency and fault-tolerance. We have argued in the introduction that in contemporary networks the use of hardware-assisted broadcast is more efficient than its linear-in-the-number-of-destinations message complexity suggests. In addition, the results of Peleg and Wool [13] indicate that for many quorum systems a linear number of messages would in fact be required to either reach a single active quorum or to detect a quorum all of whose members have either failed or are inaccessible. It may be interesting to explore a staged approach to broadcast using multicasts in conjunction with quorum systems that do not suffer from the worst case linear number of messages.

Other extensions, uses and optimizations: In this paper we concentrated on the correctness of the solution. There are obvious ways to optimize the solution. For example, instead of sending sets of quorums in the invocation of the lower layer primitive, we can easily send names of well-known quorum systems. It is also easy to reduce the number of unnecessary request deliveries and acknowledgements in the lower layer by piggy-backing cancellation messages onto broadcasts.

Our reconfigurable algorithm implements a single reconfigurer. However note that the reconfigurer need not be a single point of failure – we conjecture that the algorithm can be modified so that the processors that learn of a new configuration start using the current and the new configurations concurrently until (if ever) the reconfigurer enters its *recond-idle* phase. Of course it is also very interesting to extend the reconfigurable algorithm to multiple concurrent reconfigurers.

We are currently pursuing other uses of the lower layer computation primitive. The primitive can be extended easily to handle termination conditions (i.e., preconditions of *respond*) that are defined as a predicate (instead of expressing set containment). We are looking for algorithms that can be expressed naturally in a modular fashion using the generalized primitive as a general-purpose distributed systems building block.

Acknowledgments: The authors thank Alan Fekete and Roberto De Prisco for their comments an an earlier draft. This work was supported by the following contracts: ARPA N00014-92-J-4033 and F19628-95-C-0118, NSF 922124-CCR, and ONR-AFOSR F49620-94-1-01997.

Author email: `lynch@theory.lcs.mit.edu`, `alex@theory.lcs.mit.edu`.

References

[1] H. Attiya, A. Bar-Noy and D. Dolev, "Sharing Memory Robustly in Message Passing Systems", *J. of the ACM*, vol. 42, no. 1, pp. 124-142, 1996.

[2] P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987.

[3] S.E. Deering and D.R. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs", *ACM TOCS*, vol. 8, no. 2, 1990.

[4] S.B. Davidson, H. Garcia-Molina and D. Skeen, "Consistency in Partitioned Networks", *ACM Computing Surveys*, vol. 15, no. 3, pp. 341-370, 1985.

[5] H. Garcia-Molina and D. Barbara, "How to Assign Votes in a Distributed System," *J. of the ACM*, vol. 32, no. 4, pp. 841-860, 1985.

[6] D.K. Gifford, "Weighted voting for Replicated Data", in *Proc. of 7th ACM Symp. on Oper. Sys. Princ.*, pp. 150-162, 1979.

[7] M.P. Herlihy, *Replication Methods for Abstract Data Types*, Doctoral Dissertation, MIT, LCS/TR-319, 1984.

[8] N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[9] N.A. Lynch and M.R. Tuttle, "An Introduction to Input/Output Automata", *CWI Quarterly*, vol.2, no. 3, pp. 219-246, 1989.

[10] D. Malki and M. Reiter, "Byzantine Quorum Systems", TR CS96-8, Inst. of Comp. Sci, the Hebrew Univ. of Jerusalem, July 9, 1996.

[11] M.H. Olsen, E. Oskiewicz and J.P. Warne, "A Model for Interface Groups", *IEEE 10th Symp. on Reliable Distributed Systems*, pp. 98-107, 1991.

[12] J.-F. Paris and P.K.Sloope, "Dynamic Management of Highly Replicated Data", in *IEEE 11th Symp. on Reliable Distr. Systems*, pp. 20-27, 1992.

[13] D. Peleg and A. Wool, "How to be an Efficient Snoop, or the Probe Complexity of Quorum Systems", in *Proc. of the 15th ACM Symp. on Princ. of Distr. Comput.*, pp. 290-299, 1996.

[14] RFC 1112, *Internet Group Multicast Protocol*, Internet Standard Protocol (Recommended).

[15] A.A. Shvartsman, "Dealing with History and Time in a Distributed Enterprise Manager", *IEEE Network*, vol. 7, no. 6, pp. 32-41, 1993.

[16] M. Sloman, "Management: What and Why", in *Network and Distributed Systems Management*, M. Sloman, Ed., Addison-Wesley, 1994.

Optional Appendices

An implementation $\Delta(C)$ of $\Gamma(C)$

Here we present a straightforward implementation of the $\Gamma(C)$ primitive that we call $\Delta(C)$. The implementation uses send/receive point-to-point channels. Each channel is modelled having $send(m)_{i,j}$ and $recv(m)_{j,i}$ actions, and $channel_{i,j}$ state variable for $i, j \in PID$. Such channels have very simple specification (cf. [8]) that is omitted here. Main differences between $\Gamma(C)$ and $\Delta(C)$ are that (1) instead of the global *op*, each processor maintains a state component *op* invocations it initiates, and (2) messages are communicated using the channels with the help of queues *out-q* and *deliver-q*. It is not difficult to see that the composition of $\Delta(C)$ and the channel automata implements $\Gamma(C)$.

Specification of $\Delta(C)$

Data-types:

Operation descriptors: $desc = \langle\ msg,\ con,\ qrm,\ acc[1..n]\ \rangle \in \mathcal{D}_a$,

where $\mathcal{D}_a = M \times \Phi \times \mathcal{Q}^* \times (A \cup \{\perp\})^n$. The selectors for each component is as follows:

msg: message to be broadcast by the primitive
con: the condenser function
sel: read/write quorum selector
$acc[1..n]$: array of accumulated acknowledgements, where n is the number of processors

Operations: $\mathcal{O} = OID \rightarrow \mathcal{D}_a \cup \{\perp\}$

(Other data-types from the Sections 2 and 3.1 as needed)

State: (for each processor $i \in PID$)

$Ops : 2^{OID}$, the set of active operation identifiers, initially empty

$op \in \mathcal{O}$, operations

$out\text{-}q_j$: queues of outgoing messages to individual processors (for $j \in PID$), initially empty

$deliver\text{-}q$: the queue of incoming requests to be delivered locally, initially empty

Auxiliary variables: (computed on global state)

$op(id).dlv$: the set of processors to whom the message was delivered as the result of the primitive invocation id. It is initially empty, and it is computed as: $op(id).dlv : op(id).dlv \cup \{i\}$ in the effects of the *deliver* action.

$op(id).acks \equiv \{pid : op(id).acc[pid] \neq \perp\}$

$op(id).rsp$: when $op(id) \neq \perp$, then $op(id).rsp = \mathit{false}$ iff $id \in Ops$, and $op(id).rsp = \mathit{true}$ iff $id \notin Ops$.

Actions: (for processor i)

Input: $submit(m, \psi, s, id)_i$
$ack(v, id)_i$
$recv(m)_{j,i}$

Output: $respond(r, id)_i$
$deliver(m, id)_i$
$send(m)_{i,j}$

Transitions:

$submit(m, \psi, s, id)_i$
Eff: $op(id) := \langle m, \psi, s, \bot^n \rangle$
$Ops := Ops \cup \{id\}$
for $j \in PID$
do append $\langle m,\ id \rangle$ to $out\text{-}q_j$

$send(m)_{i,j}$
Pre: head$(out\text{-}q_j) = m$
Eff: $out\text{-}q_j :=$ tail$(out\text{-}q_j)$

$recv(m)_{j,i}$ Hidden$(m' \in M, v \in A)$
Eff: **if** $m = \langle m', id\rangle$ **then**
append $\langle m', j, id \rangle$ to $deliver\text{-}q$
else
if $m = \langle v, id\rangle \wedge id \in Ops$ **then**
$op(id).acc[j] := v$

$deliver(m, id)_i$
Pre: head$(deliver\text{-}q) = \langle m, j, id \rangle$
Eff: $deliver\text{-}q_i :=$ tail$(deliver\text{-}q_i)$

$ack(v, id)_i$
Eff: append $\langle v,\ id \rangle$ to $out\text{-}q_j$

$respond(r, id)_i$ Hidden$(Q \in \mathcal{Q}; a \in A^n)$
Pre: $id \in Ops$
$Q \in C.(op(id).sel)$
$Q \subseteq op(id).acks$
$\forall k \in Q : a[k] = op(id).acc[k]$
$\forall k \notin Q : a[k] = \bot$
Eff: $r := (op(id).con)(a)$
$Ops := Ops - \{id\}$

Appendix B: Selected proofs for Section 4

Lemma 4.2 Tags maintained by each processor are monotonically nondecreasing.

Proof: The tags are changed only in the effects of *deliver* actions used in the *propagation* of tags, where the change is effected only if the tag value is lexically increased. □

Lemma 4.3 If for an operation π, t is the tag returned by the *query* phase of the algorithm and $\tau(\pi)$ is the tag used in the *propagation* phase, then (i) if π is a read then $t = \tau(\pi)$, and (ii) if π is a write then $t < \tau(\pi)$.

Proof: Established by the *respond* action that completes the *query* phase of π. □

The main supporting lemma:

Lemma 4.4 If in an execution α, $\pi_1 \prec_{cp} \pi_2$, then (i) if the operation π_2 is a read, then $\tau_\alpha(\pi_1) \leq_{lex} \tau_\alpha(\pi_2)$, and (ii) if the operation π_2 is a write, then $\tau_\alpha(\pi_1) <_{lex} \tau_\alpha(\pi_2)$.

Proof: We consider each of the two cases:

(i) π_1 is a read or a write and π_2 is a read. Let $W_1 \in C.write$ be a write quorum used in propagating $\tau_\alpha(\pi_1)$. Let $R_2 \in C.read$ be a read quorum used the query phase of π_2. By Lemma 3.1, there is at least one processor i such that $i \in W_1 \cap R_2$ and uses its tag_i in the acknowledgement in query phase of π_2. Since tags are monotonically nondecreasing by Lemma 4.2, $tag_i \geq \tau_\alpha(\pi_1)$. Since $\tau_\alpha(\pi_2)$ is computed as the maximum over the acknowledgements received from R_2 and by Lemma 4.3 it follows that $\tau(\pi_1) \leq_{lex} \tau(\pi_2)$.

(ii) π_1 is a read or a write and π_2 is a write. Using a similar argument we can show that $tag_i \geq_{lex} \tau(\pi_1)$ for the processor i. Since $\tau(\pi_2)$ is computed by the action *respond* and from Lemma 4.3 it follows that $\tau(\pi_1) <_{lex} \tau(\pi_2)$. □

We now define the partial order needed to apply Lemma 4.1 in the main theorem for fixed configurations as follows:

Let β be any sequence of read and write operations Π containing no incomplete operations. We define the (irreflexive) partial order $PO = \langle \Pi, \prec \rangle$ on the operations by letting: $\pi_1 \prec \pi_2$ for $\pi_1, \pi_2 \in \Pi$, if

(a) $\tau(\pi_1) <_{lex} \tau(\pi_2)$, or

(b) π_1 is a write and π_2 is a read such that $\tau(\pi_1) =_{lex} \tau(\pi_2)$.

In what follows, we let ρ stand for some read operation, and ω stand for some write operation as needed.

Theorem 4.5 β satisfies the atomicity property.

Proof: The necessary properties for Lemma 4.1 (see lemma statement) are as follows:

1. If π is a write, it has a finite tag $\tau(\pi)$ and is preceded by finitely many other writes. Since Π contains no incomplete operations, there can only be a finite number of reads ρ preceding π with $\tau(\rho) <_{lex} \tau(\pi)$. Similarly, if π is a read, it can only be preceded by finitely many writes ω with $\tau(\rho) \leq \tau(\pi)$, with finitely many other reads preceding or concurrent with these writes.
2. We show this by case analysis. For an operation π, we use $\uparrow\pi$ to denote the invocation event, and $\pi\downarrow$ to denote the response event. We use ρ for read and ω for write events. With two operations, there are four cases:
   (a) $\rho\downarrow$ precedes $\uparrow\omega$ – in this case, $\tau(\rho) <_{lex} \tau(\omega)$ by Lemma 4.4. Thus $\omega \not\prec \rho$ by the *PO* construction.
   (b) $\rho_1\downarrow$ precedes $\uparrow\rho_2$ – in this case, by the definition of *PO*, if $\tau(\rho_1) <_{lex} \tau(\rho_2)$, then $\rho_2 \not\prec \rho_1$, else if $\tau(\pi_1) =_{lex} \tau(\rho_2)$, then, since both are reads, they are *not* ordered by the *PO* construction.
   (c) $\omega\downarrow$ precedes $\uparrow\rho$ – in this case $\tau(\rho) \geq_{lex} \tau(\omega)$ by Lemma 4.4, and $\omega \prec \rho$ by the *PO* construction. Thus $\rho \not\prec \omega$.
   (d) $\omega_1\downarrow$ precedes $\uparrow\omega_2$ – in this case $\tau(\omega_1) < \tau(\omega_2)$ by the same lemma again forcing $\omega_1 \prec \omega_2$ in *PO*. Thus $\omega_2 \not\prec \omega_1$.
3. This follows from the definition of *PO*, since the tags of any two writes are (lexicographically) comparable and are not equal, since they are unique. If π is a read then (a) if its tag is smaller, it implies $\pi \prec \omega$, (b) if its tag is larger, it implies $\omega \prec \pi$, or (c) if it has the same tag, then in this case again $\omega \prec \pi$.
4. The value returned by a read ρ is value written by the last preceding write ω according to $\prec$. This is so because for any such read and write pair, $\tau(\omega) = \tau(\rho)$. (If there is no preceding write, then ρ returns v_0.)

Therefore, by Lemma 4.1, any such β satisfies the atomicity property. □

Appendix C: Selected proofs for Section 5

In the presentation below where necessary, for any state component x, we let $x^{(k)}$ denote the value of the component after k transitions, and $x^{(k+1)}$ its value after the $k+1^{st}$ transition.

Lemma 5.1 Tags maintained by each processor are monotonically increasing, i.e., if for any trace, j and k are transitions such that $j \leq k$, then for all $i \in PID$ we have $tag_i^j \leq_{lex} tag_i^k$.

Proof: The tags are changed only in the effects of *deliver* actions used in the *propagation* of tags, where the change is effected only if the tag value is lexically increased. □

Lemma 5.2 If for on operation π, t is the tag returned by the *query* phase of the algorithm and $\tau(\pi)$ is the tag used in the *propagation* phase, then (i) if π is a read then $t = \tau(\pi)$, and (ii) if π is a write then $t < \tau(\pi)$.

Proof: Established by the *respond* action that completes the *query* phase of π. □

Lemma 5.3 In any reachable state, for any client-level operation π if $i \in op(oid).acks$, where oid is the invocation of the Γ primitive in the *propagation* phase of π, then $\tau(\pi) \leq tag_i$.

Proof: When a processor i acknowledges a propagated tag, it makes $tag_i = \tau(\pi)$ unless $tag_i > \tau(\pi)$. This establishes $\tau(\pi) \leq tag_i$. The invariant is maintained by the monotonicity of tag_p (Lemma 5.1) and by the fact that $\tau(\pi)$ is not changed once it is defined. □

The single reconfiguration processor r maintains the current quorum configuration sequence number $cix.act_r$ and the current configuration $Cfg.Act_r$.

In any global state, the *current* configuration index is defined to be $cix.act_r$.

For any processor p, its configuration is *current*, if $cix.act_p = cix.act_r$.

Lemma 5.4 For any processor p, $cix.act_p \leq cix.act_r$, where r is the reconfigurer.

Proof: – left as an exercise for the reader. □

Lemma 5.5 For any processor p, either $cix.act_p$ is *current*, or $\exists W \in Cfg.Act.write_p : \forall i \in W : cix.act_i > cix.act_p$.

Proof: Operationally: before the new configuration is activated by the *submit* of *"recon-done"*, all members of at least one write quorum of the previous configuration are informed of the new proposed configuration as ensured by the reconfigurer's *"query-install"* phase. □

Lemma 5.6 For any read or write operation π in its *propagation* phase that executes a *submit* at processor p:

(a) If $cix\text{-}used_p$ is not current, then $\forall W \in cfg\text{-}used_p : cix\text{-}used_p < \max_{i \in W}\{Fill(i, oid_p).cix.bid\}$.

(b) If $cix\text{-}used_p$ is current and the *submit* occurs while the reconfigurer is in its *propagation* phase, then $\forall W \in cfg\text{-}used_p : cix\text{-}used_p < \max_{i \in W}\{Fill(i, oid_p).cix.bid\}$.

Proof: (a) Using Lemma 5.5 it can be shown that the following is an invariant for the primitive invocation oid in the *propagation* phase:

If $cix\text{-}used_p < cix.act_r$ at *submit*, then $\exists R \in cfg\text{-}used_p : \forall i \in R : cix\text{-}used_p < Fill(i, oid).cix.bid$.

(b) Since $cix\text{-}used_p$ is current and the *submit* follows the reconfigurer's *respond* to *"query-install"*, then the *respond* to *"propagate"* at p returns z such that $cix\text{-}used_p < z.bid$. □

The following lemmas will be used in the proof of the main multi-part invariant. In the first two lemmas we address the state of a read or write operation that is in the *"propagate"* phase.

Lemma 5.7 If a read or write operation π at processor p is in the *propagation* phase using the primitive oid_p using $cix\text{-}used_p$ that is current, and the reconfigurer is in the *query-install* phase using the primitive oid_r, then $\forall R \in Cfg.Act.read_r : \forall W \in cfg\text{-}used_p : \forall i \in R \cap W$, at least one of the following holds:

(a) $i \notin op(oid_p).dlv \cup op(oid_r).dlv$

(b) $cix\text{-}used_p < Fill(i, oid_p).cix.bid$

(c) $\tau(\pi) \leq Fill(i, oid_r).tag$

Proof: By induction on the length of any execution of the composition of the reader/writer and the reconfig automata.

Base case: the execution is of length 0. Since there are no operations in progress, this case is vacuously satisfied.

Inductive step: assume the invariant holds for all executions consisting of k transitions of the composed automata, and we now consider an execution of length $k+1$. We are using the property of the current configuration that if $R \in Cfg.Act.read_r$ and $W \in Cfg.A.write_r$, then $R \cap W \neq \emptyset$.

submit of *"query-install"* : The effects of the transition establish $op(id).dlv$. If $op(oid_p).dlv = \emptyset$, then the clause (a) is satisfied.

Else consider any $i \in (R \cap W) \cap op(oid_p).dlv$. If such i exists, then $\tau(\pi) \leq tag_i$ (monotonicity) and this establishes the clause (c) since $tag_i = Fill(i, id).tag$.

respond to *"query-install"* : maintains the invariant.

submit of *"propagate"* : This establishes $op(id).dlv = \emptyset$. If $op(oid_r).dlv = \emptyset$, then the clause (a) is satisfied.

Else consider any $i \in (R \cap W) \cap op(oid_r).dlv$. If such i exists, then $cix\text{-}used_p \leq Fill(i, oid_p).cix.bid$. This establishes the clause (b).

respond at p : maintains the invariant.

deliver : The invariant is maintained by the monotonicity of data object tags and the configuration indices.

ack : Any *ack* maintains the invariant. □

Lemma 5.8 If a read or write operation π at processor p is in the *propagation* phase using the primitive oid_p with $cix - used_p$ current, and the reconfigurer is in the *propagation* phase, then either:

(1) $\forall W \in cfg\text{-}used.write_p : cix\text{-}used_p < \max_{i \in W}\{Fill(i, oid_p).bid\}$, **or**

(2) $\tau(\pi) \leq \tau(oid_r)$.

Proof: By induction on the length of any execution of the composition of the reader/writer and the reconfig automata.

Base case: the execution is of length 0. Since there are no operations in progress, this case is vacuously satisfied.

Inductive step: assume the invariant holds for all executions consisting of k transitions of the composed automata, and we now consider an execution of length $k + 1$.

submit of *"propagate"* of π : Since the reconfigurer is in its *"propagate"* phase and $cix\text{-}ised_p$ is current, the clause (1) is established using Lemma 5.6(b).

submit of *"propagate"* by the reconfigurer : Since the reconfigurer is in its *query* phase prior to this *submit* and $cix\text{-}used_p$ is current, Lemma 5.7 invariant holds prior to the *submit*.

Clause 5.7(a) does not hold true, since the reconfigurer proceeds to *propagate*. If clause 5.7(c) holds true prior to this *submit*, then the clause (1) of the lemma is satisfied. If clause 5.7(b) holds true prior to this *submit*, then the clause (2) is satisfied.

respond to *propagate* at p : This maintains the invariant.

respond to *"propagate"* at the reconfigurer : This maintains the invariant.

deliver : The invariant is maintained because of the monotonicity of tags and indices.

ack : The invariant is maintained because of the monotonicity of tags and indices. □

The following simple lemma establishes a property of the propagation tag of the reconfigurer.

Lemma 5.9 If the reconfigurer is in its *"query-install"* phase using the primitive oid_r and if for some read or write operation π, $\forall R \in X.A.read_r : \tau(\pi) \leq \max_{I \in R}\{Fill(i, oid_r).tag\}$, then following the *respond* to *"query-install"* and prior to *submit* of *"propagate"*, the reconfigurer's propagation tag is such that $\tau(\pi) \leq \tau(recon)$.

Proof: The lemma follows from the algorithm specification by the monotonicity of tags and the preconditions and effects of the *respond*. □

The next two lemma establish certain properties of the reconfigurer and reader/writer in their respective *"propagate"* phases.

Lemma 5.10 If the reconfigurer is in its *"propagate"* phase using the primitive oid_r and the propagation tag $\tau(recon)$, then

(a) $\forall i \in op(oid_r).acks : (\tau(recon) \leq tag_i \wedge cix_r \leq cix_p)$, **and**

(b) Following the *respond* to *"propagate"* and prior to *submit* of *"recon-done"*, $\exists W \in Cfg.Bid.write_r$: $\forall i \in W : (\tau(recon) \leq tag_i \wedge cix_r \leq cix_p)$.

Proof: The clause (a) follows from the algorithm specification by the monotonicity of tags. The clause (b) follows from clause (a) and the preconditions on the *respond*. □

Lemma 5.11 If a read or write operation π at processor p is in its *"propagate"* phase using the primitive oid_p with $cix\text{-}used_p$ that is not current, then either:

(a) $\forall W \in cfg\text{-}used_p : cix\text{-}used_p < \max_{i \in W}\{Fill(i, oid_p).cix.bid\}$, **or**

(b) $\exists W \in Cfg.Act.write_r : \forall i \in W : \tau(\pi) \leq tag_i$.

Proof: By induction on the length of any execution of the composition of the reader/writer and the reconfig automata.

Base case: the execution is of length 0. Since there are no completed operations, this case is vacuously satisfied.

Inductive step: assume the invariant holds for all executions consisting of k transitions of the composed automata, and we now consider an execution of length $k+1$.

The following transitions have the potential of affecting the invariant:

submit of *"propagate"* of π : Since $cix\text{-}used_p$ is not current, the clause (a) follows from Lemma 5.6(a).

submit of *"recon-done"* by the reconfigurer : In the state preceding the *submit*, π is in the *"propagate"* phase and the reconfigurer is in the *"propagate"* phase. Here we distinguish two cases:

- $cix\text{-}used_p$ is current in the previous state: in this case we use Lemma 5.8. If the clause 5.8(1) is true, then the clause (a) is satisfied with the. Else, if the clause 5.8(2) is true, it establishes $\tau(\pi) \leq \tau(recon)$. From Lemma 5.10(b) together with the effects of *submit* of *"recon-done"* we establish $\exists W \in Cfg.Act.write_r : \forall i \in W : \tau(\pi) \leq tag_i$ and satisfy the clause (a).
- $cix\text{-}used_p$ is not current in the previous state: Here, by the inductive hypothesis, either the clause (a) or clause (b) hold and are not affected by the *submit* of *"recon-done"* since $cix\text{-}used_p$ is not current in all cases. □

The next two lemmas establish the properties of reader/writer in its *"propagate"* phase when they use a configuration index that is not current.

Lemma 5.12 If a read or write operation π at processor p is in its *"propagate"* phase using the primitive oid_p with $cix\text{-}used_p$ that is not current, and the reconfigurer is in its *"query-install"* phase using the primitive oid_r, then either:

(a) $\forall W \in cfg\text{-}used_p : cix\text{-}used_p < \max_{i \in W}\{Fill(i, oid_p).cix.bid\}$, **or**

(b) $\forall R \in Cfg.Act.read_r : \tau(\pi) \leq \max_{i \in R}\{Fill(i, oid_r).tag\}$.

Proof: By induction on the length of any execution of the composition of the reader/writer and the reconfig automata.

Base case: the execution is of length 0. Since there are no operations in progress, this case is vacuously satisfied.

Inductive step: assume the invariant holds for all executions consisting of k transitions of the composed automata, and we now consider an execution of length $k+1$.

The following transitions have the potential of affecting the invariant:

submit of *"propagate"* of π : Since *cix-used*$_p$ is not current, the clause (a) follows from Lemma 5.6(a).

submit of *"query-install"* : Prior to this transition, π is still in the *"propagate"* phase with *cix-used*$_p$ not current. Therefore Lemma 5.11 applies. If the clause 5.11(a) is true, then the clause (a) is satisfied.

Assume the clause 5.11(b) is true. Then by the intersection property of the read and write quorums in *Cfg.Act*$_r$, the clause (b) is satisfied. □

Lemma 5.13 If a read or write operation π at processor p is in its *"propagate"* phase using the primitive *oid*$_p$ with *cix-used*$_p$ that is not current, and the reconfigurer is in its *"propagate"* phase using the primitive *oid*$_r$, then either:

(a) $\forall W \in \textit{cfg-used}_p : \textit{cix-used}_p < \max_{i \in W}\{\textit{Fill}(i, \textit{oid}_p).\textit{cix}.\textit{bid}\}$, **or**

(b) $\tau(\pi) \leq \tau(\textit{oid}_r)$.

Proof: By induction on the length of any execution of the composition of the reader/writer and the reconfig automata.

Base case: the execution is of length 0. Since there are no operations in progress, this case is vacuously satisfied.

Inductive step: assume the invariant holds for all executions consisting of k transitions of the composed automata, and we now consider an execution of length $k+1$.

The following transitions have the potential of affecting the invariant:

submit of *"propagate"* of π : Since *cix-used*$_p$ is not current, the clause (a) follows from Lemma 5.6(a).

submit of *"propagate"* by the reconfigurer : Prior to this transition, π is in its *"propagate"* phase and the reconfigurer is in its *"query-install"* phase. Therefore Lemma 5.12 applies. If the clause 5.12(a) is true, then the clause (a) is satisfied. Else the clause 5.12(b) is true. □

We now show the main multi-part invariant:

Lemma 5.14 In all reachable states:

I1 $\forall \pi \in \textit{completed}$,

(a) if the reconfigurer is in its *recon-idle* phase :
$\exists W \in \textit{Cfg.Act.write}_r : \forall i \in W : \tau(\pi) \leq \textit{tag}_i$

(b) if the reconfigurer is in its *query-install* phase having invoked Γ using identifier *oid*$_r$: $\forall R \in \textit{Cfg.Act.read}_r : \tau(\pi) \leq \max_{i \in R}\{\textit{Fill}(i, \textit{oid}_r).\textit{tag}\}$

(c) if the reconfigurer is in its *propagate* phase having invoked Γ using identifier *oid*$_r$ and the tag $\tau(\textit{recon})$: $(\tau(\pi) \leq \tau(\textit{recon})) \wedge (\forall i \in \textit{op}(\textit{oid}_r).\textit{acks} : \tau(\textit{recon}) \leq \textit{tag}_i)$

I2 $\forall \pi \notin \textit{completed}$,

(a) If $\pi' \prec_{cp} \pi$ and π at processor p is in the *query* phase having invoked Γ using identifier *oid*, then for any $R \in \textit{cfg-used.read}_p$, then

(1) $\tau(\pi') \leq \max_{i \in R}\{\textit{Fill}(i, \textit{oid}).\textit{tag}\}$, **or**

(2) $\textit{cix-used}_p < \max_{i \in R}\{\textit{Fill}(i, \textit{oid}).\textit{cix}.\textit{bid}\}$.

(b) If π is in the *propagation* phase having invoked Γ using identifier *oid*, then
(1) $\textit{cix-used}_p$ is *current*, **or**
(2) $\exists W \in \textit{Cfg.Act.write}_r : \forall i \in W : \tau(\pi) \leq tag_i$, **or**
(3) $\forall W \in \textit{cfg-used.write}_p : \textit{cix-used}_p < \max_{i \in W}\{\textit{Fill}(i, oid).cix.bid\}$.

I3 If $\pi_1 \prec_{cp} \pi_2$ and $\tau(\pi_2)$ is defined, then
(a) $\tau(\pi_1) \leq \tau(\pi_2)$ when π_2 is a read,
(b) $\tau(\pi_1) < \tau(\pi_2)$ when π_2 is a write.

Proof: By induction on the length of any execution of the composition of the reader/writer and the reconfig automata.

Base case: the execution is of length 0. Since there are no completed operations, this case is vacuously satisfied.

Inductive step: assume each of the three invariants of the lemma hold for all executions consisting of k transitions of the composed automata, and we now consider an execution of length $k+1$. The inductive step is divided into three parts:

Inductive step for **I1a**: Only the following actions can affect the invariant:

respond to *"propagate"* : Here we only need to consider a client-level operation π that becomes *completed* as the result of the *respond* of the *propagation* phase of π. Let *oid* be the identifier of the *query* phase. From the preconditions to *respond*, $\exists W \in \textit{cfg-used.write}_p$ such that $W \subseteq op(oid).acks$. The operation becomes *completed* as the effect of the transition iff $\textit{cix-used}_p \geq z.bid$. By Lemma 5.3, if $i \in op(oid).acks$ then $\tau(oid) = \tau(\pi) \leq tag_i$. If $\textit{cix-used}_p$ is *current* then the invariant **I1a** is re-established. Else if $\textit{cix-used}_p$ is *not current* then we use the inductive hypothesis **I2b**. Since π completes, then the clause (2) must hold, i.e., $\exists W \in \textit{Cfg.Act.write}_r : \forall i \in W : \tau(\pi) \leq tag_i$ and **I1a** is re-established.

submit of *"recon-done"* : Prior to this transition, the reconfigurer is in its *"propagate"* phase. Using the inductive hypothesis for **I1c**, we have $\tau(\pi) \leq \tau(recon)$. Together with the effects of the transition and Lemma 5.10 this re-establishes **I1a**.

deliver or *ack* : The invariant is preserved by monotonicity of tags and indices.

Inductive step for **I1b**: Only the following actions can affect the invariant:

respond to *"propagate"* of π : Prior to this transition, π is in its *"propagate"* phase using the primitive oid_p and configuration index $\textit{cix-used}_p$ and the reconfigurer is in its *"query-install"* phase. We distinguish two cases

- $\textit{cix-used}_p$ is current: Here Lemma5.7 applies. Since π completes, only the clause 5.7(c) is true. This is sufficient to re-establish **I1b**.
- $\textit{cix-used}_p$ is not current: Here Lemma5.12 applies. Since π completes, only the clause 5.12(b) is true. This is sufficient to re-establish **I1b**.

submit of *"query-install"* : Prior to the transition, $\pi \in \textit{completed}$ and the reconfigurer is in its *"recon-idle"* phase. We use the inductive hypothesis for **I1a** and the intersection property of read and write quorums and Definition 5.2 of *Fill* to re-establish **I1b**.

Inductive step for **I1c**: Only the following actions can affect the invariant:

respond to *"propagate"* of π : Prior to this transition, π is in its *"propagate"* phase using the primitive oid_p and configuration index $\textit{cix-used}_p$ and the reconfigurer is in its *"propagate"* phase. We distinguish two cases

- *cix-used*$_p$ is current: Here Lemma5.8 applies. Since π completes, only the clause 5.8(2) is true. This is sufficient to re-establish **I1c**.
- *cix-used*$_p$ is not current: Here Lemma5.13 applies. Since π completes, only the clause 5.13(b) is true. This is sufficient to re-establish **I1b**.

submit of *"propagate"* by the reconfigurer : Prior to the transition, $\pi \in$ *completed* and the reconfigurer is in its *"query-install"* phase. We use the inductive hypothesis for **I1b** and LemmaLemQ7 to re-establish $\tau\pi \le \tau(recon)$ and thus **I1c**.

Inductive step for **I2a**: Only the following actions can affect the invariant:

submit of *"query"* : Consider a new client-level operation π_2 and the *submit* with identifier *id* of its *query*. Assume that there is also an operation π_1 such that $\pi_1 \prec_{cp} \pi_2$ (if no such π_1 exists then **I2a** is preserved).

If *cix-used*$_p$ is current then using the inductive hypothesis for **I1a** we have $\exists W \in Cfg.Act.write_p$: $\forall i \in W : \tau(\pi_1) \le tag_i$. This establishes the clause **I2a**(1).

Else *cix-used*$_p$ is not current. By Lemma 5.5 $\exists W \in Cfg.Act.write_p : \forall i \in W : cix.act_i >$ *cix-used*$_p$. By the intersection property of read and write quorums, this establishes the clase **I2a**(2).

In either case the invariant **I2a** is re-established.

deliver : We only need to consider the actions of the type *deliver*($\langle$ *"query"* $\rangle$, *id*)$_p$. From the code of the composed automata: $op(oid).dlv^{(k+1)} = op(oid).dlv^{(k)} \cup \{i\}$ and $ack\text{-}q_i^{(k+1)} = ack\text{-}q_i^{(k)} \circ \langle\langle val_i, tag_i\rangle, id\rangle$. The effects of this on **I2a** is to move, for the processor i, is to place the value of tag_i on the $ack\text{-}q_i$. This does not change the set of values used to compute the maximum in **I2a** and preserves the invariant.

ack : We only need to consider the actions of the type $ack(\langle val, tag\rangle, id)_i$ in the *query* phase. From the code of the composed automata: $ack\text{-}q_i^{(k+1)} = \mathrm{tal}(ack\text{-}q_i^{(k)})$ and $op(id).ack[i]^{(k+1)} = \langle val, tag\rangle$. This does not change the set of values used to compute the maximum in **I2a**, since the effects of this is to set $op(id).acc[i]$ to the tag that was previously in the queue $ack\text{-}q_i$. The invariant is re-established.

respond to *"query"* : Since either the clause (1) or (2) is true prior to this transition, it is still so as the result of the transition.

Inductive step for **I2b**: Only the following action can affect the invariant:

submit of *"propagate"* : If *cix-used*$_p$ is *current* then the clause (1) is established.

Else *cix-used*$_p$ is not current. Here, by Lemma 5.5 the clause (3) is established.

submit of *"recon-done"* : If some π is in its *"propagate"* phase at processor *p*, then *cix-usedp* is no longer *current*. Prior to the *"recon-done"*, the reconfigurer is in its *"propagate"* phase, and so is π. We consider two cases:

- *cix-used*$_p$ is current prior to *"recon-done"*: Therefore Lemma 5.8 applies.

  If the clause 5.8(1) is true prior to *"recon-done"*, then it is still the case. This establishes the clause **I2b**(3).

  If the clause 5.8(2) is true prior to *"recon-done"*, then $\tau(oid_r) \ge \tau(\pi)$. Together with Lemma 5.10 and the effects of the *submit* of *"recon-done"* implies $\exists W \in Cfg.Act.write_r$: $\forall i \in W : \tau(\pi) \le tag_i$. This establishes the clause **I2b**(2).

- *cix-used*$_p$ is not current prior to *"recon-done"*: Here π is in its *propagate* phase and Lemma 5.13 applies. If the clause 5.13(a) is true, then the clause **I2b**(3) is established. Else the clause 5.13(b) is true. Together with Lemma 5.10 and the effects of the *submit* of *"recon-done"* implies $\exists W \in \mathit{Cfg.Act.write}_r : \forall i \in W : \tau(\pi) \le tag_i$. This establishes the clause **I2b**(2).

respond to *"propagate"* : If *cix-used*$_p$ is *current*, then clause (1) is established.

Assume *cix-used*$_p$ is *not current*. Using the induction hypothesis, if **I2b**(3) is true prior to the transition, then it is still true after. Else if **I2b**(2) is true prior to the transition, then it is still true.

The invariant **I2b** is re-established.

Inductive step for **I3**: Only the following action can affect the invariant:

respond to *"query"* : Here for some client-level operation π_2, the *respond* for a *query* defines new $\tau(\pi_2)$. Prior to the respond, π_2 was in the *query* phase, and we use the inductive hypothesis **I2a**.

If the clause **I2a**(2) is true prior to the transition, then $\tau(\pi_2)$ is still undefined. If the clause **I2a**(1) is true prior to the transition, then $\tau(\pi_2)$ is defined using a read quorum $R \in$ *cfg-used.read*$_p$ as $\max_{i \in R}\{Fill(i, oid).tag\}$.

The above maximum is the value of the variable t used in the computation of $\tau(\pi_2)$ in the effects of *respond*. When π_2 is read, this results in $\tau(\pi_2) = t$, and when π_2 is write, this results in $\tau(\pi_2) > t$. Thus, for any $\pi_1 \prec_{cp} \pi_2$, if π_2 is a read, then $\tau(\pi_1) \le \tau(\pi_2)$, and if π_2 is a write, then $\tau(\pi_1) < \tau(\pi_2)$. Therefore **I3** is maintained.

□

Lemma 5.15 In any execution, if $\pi_1 \prec_{c.p.} \pi_2$, then (i) if π_2 is a read operation, then $\tau(\pi_1) \le_{lex} \tau(\pi_2)$, and (ii) if π_2 is a write operation, then $\tau(\pi_1) <_{lex} \tau(\pi_2)$.

Proof: Using Lemma 5.14(I3) and Lemma 5.2. □

We now use by Lemma 4.1 of [8]. (In what follows, we let ρ stand for some read operation, and ω stand for some write operation as needed.)

Let β be any sequence of read and write operations Π containing no incomplete operations. We define the (irreflexive) partial order $PO = \langle \Pi, \prec \rangle$ on the operations by letting: $\pi_1 \prec \pi_2$ for $\pi_1, \pi_2 \in \Pi$, if

(a) $\tau(\pi_1) <_{lex} \tau(\pi_2)$, or

(b) π_1 is a write and π_2 is a read such that $\tau(\pi_1) =_{lex} \tau(\pi_2)$.

Theorem 5.16 β satisfies the atomicity property.

Proof: Follows the proof of Theorem 4.5. □