Graceful Quorum Reconfiguration in a Robust Emulation of Shared Memory*

Burkhard Englert† Alexander A. Shvartsman‡

Abstract

Providing shared-memory abstraction in message-passing systems often simplifies the development of distributed algorithms and allows for the reuse of shared-memory algorithms in the message-passing setting. A robust emulation of atomic single-writer/multi-reader registers in message-passing systems was developed by Attiya, Bar-Noy and Dolev (1995). This emulation was extended by Lynch and Shvartsman (1997) to multi-writer/multi-reader registers using reconfigurable quorum systems. In this work we present a new atomic multi-writer/multi-reader register service that includes a fault-tolerant reconfiguration service. This new emulation has a substantially improved performance and fault-tolerance characteristics. We introduce the concept of intermediate quorum configurations *and show how they can be used by readers/writers during reconfiguration. The result is that the quorum reconfigurations are* graceful*: readers and writers no longer "busy-wait" during reconfigurations, but are able to complete their operations. An additional advance is that the reconfigurer is eliminated as the single point of failure. When the reconfigurer fails, readers and writers continue using intermediate configurations. In finite executions, read and write operations terminate in bounded time using bounded number of messages (the bounds depend on the "currency" of the configuration at the invoker of the operation). Finally, the service places no restrictions on the installed quorum configuration: a previously installed quorum system can be replaced by an* arbitrary *new quorum system. Our algorithms are specified using I/O Automata; the safety proofs use the partial order techniques and invariants, and the performance is assessed using operational reasoning.*

1. Introduction

Algorithms for multiprocessors are commonly expressed using either the shared-memory paradigm or the message-passing paradigm. For distributed algorithms to be practical, the algorithms must be efficient and scalable, and they must tolerate asynchrony, and component failures. It has been observed that in many cases it is easier to develop efficient fault-tolerant 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. It is likewise advantageous for message-passing algorithms to have access to building blocks providing shared-memory abstraction in distributed settings.

Among the important results in this area are the algorithms of Attiya, Bar-Noy and Dolev [5] who showed that it is possible to emulate atomic shared memory robustly in message-passing systems. They show 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. These algorithms are based on processor majorities and thus are able to tolerate failure patterns where any minority of processors are disabled or are unable to communicate. This result was further optimized by Attiya [4] who improved the message complexity of the bounded time-stamps algorithm.

Motivated by [5], Lynch and Shvartsman [26] developed a robust emulation of *multi-reader/multi-writer* atomic registers using *reconfigurable quorum systems*, where a designated processor acts as the reconfigurer. The approach of [26] recognized that a service providing an atomic register abstraction in a distributed setting needs to support multiple writers as well as multiple readers, and it must be able to ensure atomicity using means that are more flexible and efficient than the majorities. As the result, that approach specified the multi-reader/multi-writer protocol that relies on quorum systems, which in turn can be dynamically changed during the system operation. The system provides an *application* interface used to submit read/write requests, and a *management* interface used to install new quorum systems in response to failures and to changing processor loads. The management requests are submitted at a single reconfigurer that is responsible for initializing and finalizing the installation of new configurations. The protocol [26] is complex and involves several subtle phases. To insure safety of reconfigurations, the protocol restricted the ability of some reads and writes to make progress during reconfigurations. We illustrate why this was necessary in [26] with the help of Figure 1. The example shows the timelines of four processors, a (the reconfigurer), b, c and d, where the arrows represent selected message transmissions. The communication is done using quorum-acknowledged broadcasts (we

*This work was supported by a grant from the NSF CAREER Award and an AFOSR contract.

†University of Connecticut, Dept. of CS and Engineering, Storrs, CT 06269, USA, Email: burkhard@cse.uconn.edu

‡University of Connecticut, Department of Computer Science and Engineering, Storrs, CT 06269, USA and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA, Email: aas@cse.uconn.edu

	Read quorums	Write quorums
Current configuration	$\{a,b,c\},\{c,d\}$	$\{a,b,c\},\{c,d\}$
Next configuration	$\{a,b\},\{b,c,d\}$	$\{a,b\},\{b,c,d\}$

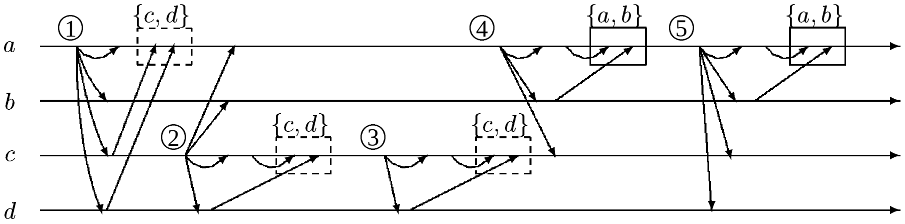

Figure 1. Illustrating the need to prevent writes from completing during reconfigurations in [26].

omit most messages that have no impact on the protocol). The system begins with the current quorum configuration with the read quorums $\{a,b,c\}$ and $\{c,d\}$, and identical write quorums (for simplicity). Responses from these quorums are marked by dashed boxes. Assume that a new configuration is submitted by the reconfigurer a. This next quorum system has the read quorums $\{a,b\}$ and $\{b,c,d\}$, and identical write quorums. Responses from these quorums are marked by solid line boxes. According to the algorithm, the reconfigurer uses a broadcast to query other processors for the latest value and the version of the shared register, see callout (1). Once a complete quorum responds, the reconfigurer accepts the value with the maximum version number. Now we assume that register writes are allowed to complete during the reconfiguration (of course the protocol [26] prevents this). Suppose the processor c starts a write (2) by quering other processors for their latest version numbers and values values. Let $\{c,d\}$ be the first responding quorum. The processor c increments the latest version and propagates the new version and value (3). Note that this new version number is strictly greater than the version that the reconfigurer knows about. The write completes after the quorum $\{c,d\}$ confirms the write (3). Now the reconfigurer propagates its outdated version number and value (4), and after the new quorum $\{a,b\}$ responds, the reconfigurer confirms the installation of the new configuration to all processors (5), and once the quorum $\{a,b\}$ responds, it completes the reconfiguration. The result is that a future read might not return the value last written in $(2,3)$, but the one propagated by the reconfigurer in (4). Hence the atomicity of the shared register is violated. Therefore the algorithms in [26] do not allow the steps (2) or (3) to complete until the reconfiguration completes (5). This ensures the safety of the protocol at the expense of the liveness of reads and writes that are concurrent with a reconfiguration. In particular, the system could starve if the reconfigurer stopped during the installation of a new configuration, effectively leaving the emulated shared register permanently inaccessible. Note that the alternative, which favors reads/writes, and that blocks a concurrent reconfiguration is also not satisfactory.

Contributions. In this paper we present a robust emulation of atomic multi-reader/multi-writer memory using dynamic quorum configurations. The emulation includes a fault-tolerant quorum reconfiguration service that allows a great deal of asynchrony and that does not use quorums for locking or mutual exclusion. The main results in this paper make the following contributions:

1. We present a protocol for multi-reader/multi-writer atomic registers that allows *all* read and write operations to complete in finite number of steps, using bounded number of messages, when the reconfigurations complete as well as when the reconfigurer fails (provided the quorum systems are not disabled).
2. We introduce the concept of *intermediate quorum configurations* and the *quorum-join* operation that, given *any* two quorum configurations, computes the corresponding intermediate configuration.
3. Our protocol ensures the liveness of the multi-reader/multi-writer protocol by using the intermediate configurations in the way that allows concurrent reconfigurations and that tolerates the failure of the reconfigurer, thus eliminating the reconfigurer as the single point of failure.
4. The clients of our management interface can submit *arbitrary* new quorum configurations, regardless of any intersection properties with any of the previous quorum configurations.

Our system is designed in a modular way and is specified as a composition of components. We use Input/Output Automata [27, 25] to specify all components and algorithms. The safety proofs, which are omitted for space reasons and are given in the full paper, use the partial order techniques and invariants [25] . The safety of the system is shown assuming complete asynchrony of the processors and message-passing. The processors may have arbitrary relative speeds (here stopped processors take infinite time to complete a step), and messages may incur arbitrary in-transit delay (here message loss corresponds to infinite delays).

We use operational reasoning to assess the conditional performance of the system. To do this we assume that there is a constant g that represents the upper bound on time it takes for the active (non-stopped) processors to perform a local computation, and the upper bound on message delay for messages that are delivered. We also assume that the quorum systems are not disabled (i.e., we assume that the

processors in at least one read quorum and at least one write quorum are active). In our system, the installed quorum configurations and the intermediate configurations can be sequentially numbered. We define the "distance" between any two such configurations as the difference between their sequence numbers. We show the following:
5. Any reconfiguration of quorums takes time at most $15g$ and at most $6n$ messages, where n is the initial number of processors.
6. Let t be the time such that either all reconfigurations complete by time t, or that the last reconfiguration active at t never completes. Any read or write operation, started at processor p that does not fail, takes at most $10g + d \cdot 5g$ time, and at most $(2d + 4)n$ messages, where d is the distance between the highest configuration anywhere at t and the configuration of p at the invocation of the operation.
7. If a read or a write operation at processor p starts at time t_1 and completes at time t_2, then $t_2 - t_1 \leq 10g + (d+2c) \cdot 5g$, and the number of messages sent is at most $(4 + 2d + 4c)n$, where d is the distance between the highest configuration anywhere at t_1 and the configuration of p at t_1, and c is the number of reconfigurations that are concurrent with the operation.

Developing protocols that meet the design goals in our setting is difficult. We do not assume the availability of reliable broadcasts, thus not all processors may learn of the ongoing installation of a new configuration. Furthermore, since we allow processors to take steps that are arbitrarily long, the reconfigurer in particular may become out of date with respect to reads and writes during the installation of a new configuration. We need a distributed solution that does not rely on the availability of the reconfigurer to take steps at the same pace as any other processor at any given time. Finally, since we allow *arbitrary* new quorum systems to be installed, we cannot rely on any intersection properties among quorums of different configurations.

The new emulation uses a single reconfigurer as in [26], however the rôle of the reconfigurer is different, resulting in the reconfigurer no longer being a single point of failure. In its new rôle, the reconfigurer is responsible for emitting new quorum configurations, and it helps shepherding the new configuration towards its installation. We show that our system is not obstructed by a reconfiguration in progress or by a tardy or stopped reconfigurer. In our system, processors contribute to installations of new configurations and intermediate configurations while participating in routine read/write operations. The overall solutions is specified in terms of the composition of two layers: The lower layer uses the Γ primitive [26] that provides an unordered broadcast-convergecast service. We also show an implementation of Γ using point-to-point channels. The lower level admits other implementations and it is not difficult to optimize its message complexity by replacing broadcasts with multicasts to specific quorums and by cancelling unnecessary pending responses using notices piggybacked on other messages. The higher layer algorithm emulates multi-access registers where dynamic quorum systems are used to ensure atomicity [20, 23, 25].

The solution implemented as a composition of 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. [34]). When a quorum system needs to be reconfigured, this is done using the management interface of our service. Reconfigurations are transparent to the clients that are using the functional read/write interface. The management interface can be used to tune the performance of a distributed system based on current and historical observations [33]. A resource manager can monitor system performance and availability and evolve the quorum system using the management interface.

Related work. The work of [5] shows how to use majorities in implementing atomic registers, and it is extended in [4]. Dynamic quorum-based emulation is given in [26]. Quorum systems [16] are generalizations of majorities. A *quorum system* (also called a *coterie*) is a collection of sets such that any two sets, called *quorums*, intersect [15]. Another 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, and any two write quorums intersect. Quorums have been used to implement distributed mutual exclusion [15] and data replication protocols [12, 18]. Quorums can be used with replicated data in transaction-style synchronization that limits concurrency (cf. [8]). Many other replication techniques use quorums [1, 6, 7, 13, 14, 17]. An additional level of fault-tolerance in quorum-based approaches can be achieved using the byzantine quorum approach [28, 3].

We have recently used the techniques in [26] and in this work to develop a way of integrating dynamic quorums within a group communication service [11]. That work introduces the notion of primary configurations and provides a dynamic primary configuration group communication service. The group communication service also allows one to implement atomic registers, however in that work we require that the new quorums have specific intersection properties with previous configurations, whereas in this paper we allow for *arbitrary* new configurations to be installed.

Considering the fault tolerance of quorum assignments, there exists a variety of previous research. Probabilistic approaches such as [2, 24, 30, 31], develop methods to determine the likelihood that progress is achieved given that a non-adaptive quorum system is used. Processors are assumed to fail with a known probability, so a quorum assignment can be selected maximizing the probability of progress. This method can also be used with our emulation to allow a system monitor to evaluate the current system and to make decisions concerning its replacement. The deterministic approach in [9] generates a static quorum assignment that guarantees to mask a predetermined number of failures. Other approaches achieve adaptive deterministic fault-tolerance by having each processor, based on information about processor failures, compute the processors in its quorum.

Another important approach to quorum adaptation is dynamic voting [19, 21, 22]. In [21] no single processor acts as a reconfigurer and the approach relies on locking and requires that at least a majority of all the processors in some previously updated quorum (or half of all the processors in some previously updated quorum plus the distinguished site) are still alive. The approach in [22] does not rely on locking, but requires at least a predefined *Min_quorum_size* number of processors to always be alive. The decentralized on-line quorum adaptation of [7] assumes the use of Sanders [32] mutual exclusion algorithm, which again relies on locking. [7] allows up to $n - 1$ processor failures but no link failures. Our approach takes a distributed system management view where reconfiguration must be graceful and asynchronous, and it must not obstruct client operations. Furthermore, the quorums do not evolve spontaneously, but are to be evolved in response to specific system policies and observations.

2. Solution Structure, Models and Notation

The emulation system is architected in terms of two main protocol layers.The higher layer provides the reconfigurable atomic read/write register service to its users that has two interfaces. The application interface provides its client with read/write access to atomic registers, and the management interface allows a system manager (a user or a system) to reconfigure the quorums by submitting new quorum configurations.

Functional interface: The clients of the service submit *read* requests (or *write* requests) at any processor i of the system. Once the operation completes, the client is informed by means of the *read-confirm*(v) event containing the read value v (or *write-confirm* event concluding the write). From the standpoint of the client, these read and write operations are independent from any quorum reconfiguration.

Management interface: The management interface allows an external system monitor to adapt the quorum system in response to the the changes in the environment, e.g., by reacting to failures and load imbalances. The monitor submits arbitrary new quorum configurations by means of the *recon*(C) request at the designated reconfigurer processor r. When the reconfiguration completes, the monitor is informed via the *recon-confirm* event.

We do not specify the clients of the system, i.e., the users of the read/writer service and the monitor.

The lower level provides the broadcast-convergecast service (the Γ primitive) to the higher level. It is presented in Section 5.

2.1. Model and Conventions

We use the following message-passing model in this work. There are n asynchronous processors with unique identifiers in the set *PID*. For simplicity we assume $PID = \{1..., n\}$. Processor communicate at the level of abstraction of the *network layer* using *point-to-point* messages, i.e., in normal operations, any processor can send messages to any other processor, the delivery is unreliable, but the messages are not corrupted. In the cases where a message is sent to all processors, broadcast can be used without assuming any atomic, FIFO or causal properties.

The following failure model is used. Processor crashes and restarts are approximated by subjecting the processors to unbounded delays with the additional assumption that the state components defined in the specification are stored in non-volatile storage. Link failures may make some nodes unreachable (some of the time or permanently).

In safety proofs we do not make *any* 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. To evaluate the performance of the algorithms, we assume that either point-to-point messages are delivered and locally processed in bounded time (unknown to the processors), or not delivered at all, and that the quorum systems are not disabled, i.e., at least one quorum is able to respond.

For the rest of the paper we define the following data types and conventions: $OID = \cup_{i \in PID} OID_i$ is the set of unique operation identifiers, where OID_i is the set of identifiers generated by processor i and for $i \neq j$ we have $OID_i \cap OID_j = \emptyset$. $\mathbf{N}$ is the naturals starting with 0. V is the set of register values, with a distinguished $v_0 \in V$. For any set S we define $S_\perp$ to be $S \cup \{\perp\}$, where $\perp$ is a distinguished null value.

2.2. Programming Notation and Methodology

The specifications in this paper are done in terms of I/O automata [25, 27]. Each automaton models a state machine with states and transitions between states, where actions are associated with sets of state transitions. There are input, output and internal actions labeled by **Inp**, **Out** and **Int** respectively. A particular action is enabled iff the preconditions (labeled **Pre:**) of that action are satisfied. The statements given as effects (labeled **Eff:**) are executed as a program started on the existing state and atomically producing the next state as the result of the transition.

The automata are *input-enabled*, where the preconditions of the input actions are always true, and we omit the preconditions "**Pre:** *true*" from the specifications. We make use of the compositions of automata that yield other automata (see [25] for details). When we compose two automata, the actions that are unique to each automaton remain unchanged in the composition. When two automata include identically named actions, one of the actions must be an input action and the other an output action. In the composition this produces an action with the same name, whose precondition is the precondition of the output action, and whose effects is the sequential composition of the effects of the two actions. In our specifications, it is always possible to establish through static "compile-time" checking that the effects of the two actions being composed do not interfere with each other. Composition is associative and commutative. We use $\circ$ to denote the infix composition operator, and we use the product notation Π to denote compositions of several automata.

An *execution* α of an I/O automaton A is a 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. We say that automaton A *implements* automaton B when the set of the traces of A is a subset of the set of the traces of B. In the performance analysis we consider finite executions.

3. Intermediate Configurations and Graceful Reconfiguration

By *graceful reconfiguration* we mean that the read and write operations are able to successfully complete during the reconfiguration, even if the reconfiguration is permanently stalled because of a reconfigurer failure. Graceful reconfiguration is implemented with the help of *intermediate quorum configurations*. As we will show, intermediate configurations obviate the need for read/write operations to "busy-wait" during reconfigurations.

In this section we let $\mathcal{P}$ denote a finite set (of processor identifiers). We define:

Definition 3.1 Let $\mathcal{R}, \mathcal{W} \subset 2^{\mathcal{P}}$ such that $\forall R_i \in \mathcal{R}$, $\forall W_i W_j \in \mathcal{W}$, $R_i \cap W_j \neq \emptyset$, and $W_i \cap W_j \neq \emptyset$, then $\mathcal{C} = \langle \mathcal{R}, \mathcal{W} \rangle$ is a *quorum configuration* of $\mathcal{P}$, with $\mathcal{R} = \mathcal{C}.read$, $\mathcal{W} = \mathcal{C}.write$.

We define the *quorum-join* operation:

Definition 3.2 Let $\mathcal{Q}, \mathcal{Q}' \subset 2^{\mathcal{P}}$. We define the quorum-join of $\mathcal{Q}$ and $\mathcal{Q}'$ to be $\mathcal{Q} \bowtie \mathcal{Q}' \equiv \{X \cup Y : X \in \mathcal{Q} \wedge Y \in \mathcal{Q}'\}$. We define the quorum-join of quorum configurations $\mathcal{C} = \langle \mathcal{R}, \mathcal{W} \rangle$ and $\mathcal{C}' = \langle \mathcal{R}', \mathcal{W}' \rangle$ to be $\mathcal{C} \bowtie \mathcal{C}' \equiv \langle \mathcal{R} \bowtie \mathcal{R}', \mathcal{W} \bowtie \mathcal{W}' \rangle$.

We show that quorum-join of two quorum configurations is also a quorum configuration:

Lemma 3.1 Let $\mathcal{P}$ be a set, $\mathcal{C} = \langle \mathcal{R}, \mathcal{W} \rangle$ be a quorum configuration of $\mathcal{P}$. Then
1. $\forall X \subset \mathcal{P}, \forall R_i \in \mathcal{R}$: $(R_i \cup X) \cap W_i \neq \emptyset$, $\forall W_i \in \mathcal{W}$: $(W_i \cup X) \cap R_i \neq \emptyset$.
2. $\forall X, Y \subset \mathcal{P}, \forall W_i W_j \in \mathcal{W}$: $(W_i \cup X) \cap (W_j \cup Y) \neq \emptyset$.
3. $\forall X, Y \subset \mathcal{P}, \forall R_i \in \mathcal{R}, \forall W_i \in \mathcal{W}$: $(R_i \cup X) \cap (W_j \cup Y) \neq \emptyset$.

Theorem 3.2 Let $\mathcal{C} = \langle \mathcal{R}, \mathcal{W} \rangle$, $\mathcal{C}' = \langle \mathcal{R}', \mathcal{W}' \rangle$ be quorum configurations of $\mathcal{P}$, then
$\mathcal{C} \bowtie \mathcal{C}'$ is a quorum configuration of $\mathcal{P}$.

Our new algorithms (formally presented in Section 6) use *intermediate quorum configurations*, expressed in terms quorum-joins, to prevent the problem described in the example in Figure 1. If a processor has a previously installed configuration $\mathcal{C}$, and it learns of a new proposed configuration $\mathcal{C}'$, then, instead of "busy-waiting" until the installation of $\mathcal{C}'$ is finalized, it proceeds with its read/write operations using the intermediate configuration $\mathcal{C} \bowtie \mathcal{C}'$. The individual quorum intersection properties of both $\mathcal{C}$ and $\mathcal{C}'$ are preserved in $\mathcal{C} \bowtie \mathcal{C}'$ (Lemma 3.1). The use of intermediate quorum configurations, as we show in Section 7.1, makes it safe to proceed with reads and writes during the installation of a new configuration. Furthermore, this has the positive effect of "helping" the reconfigurer in installing new configurations, since the messages sent by readers/writers propagate new configurations. Finally, the sizes of the quorums in quorum-joins are no more than twice the maximum size of the original quorums.

Theorem 3.3 If $\mathcal{Q}_1, \mathcal{Q}_2 \subset 2^{\mathcal{P}}$, then $\max_q\{|q| : q \in \mathcal{Q}_1 \bowtie \mathcal{Q}_2\} \leq 2\max_q\{|q| : q \in \mathcal{Q}_1 \vee q \in \mathcal{Q}_2\}$.

4. Formal System Structure

We specify systems in a modular way as compositions of automata and we define the following automata and their compositions:

Reader/Writer: This automaton specifies the algorithm for reads and writes. The automaton at processor i is denoted by $\mathcal{A}^{(i)}$. There are n reader/writer automata, one for each $i \in \{1, \ldots, n\}$.

Reconfigurer: This automaton specifies the reconfigurer algorithm. One of the n processors, r, is selected to act as the reconfigurer, who initiates installations of new configurations. This automaton is denoted by $\mathcal{R}ec$.

The broadcast/convergecast specification: This broadcast/convergecast used by $\mathcal{A}^{(i)}$ and $\mathcal{R}ec$ is specified by automaton $\Gamma^{(i)}$. The Γ primitive is defined as the composition $\Gamma = \Pi_{i-1}^{n} \Gamma^{(i)}$.

Communication channels: The low-level unidirectional message-passing channel from processor i to j is denoted by $ch_{i,j}$.

The broadcast/convergecast implementation: The broadcast/convergecast is implemented by the automata $\Delta^{(i)}$ at each $i \in \{1, \ldots, n\}$ using the channels. Formally, Γ is implemented by Γ_{impl} that is defined as the composition $\Gamma_{impl} = \Pi_{i=1}^{n} \Delta^{(i)} \circ \Pi_{1 \leq i,j \leq n} ch_{i,j}$

The atomic Read/Write service (the system): We define the system $\mathcal{S}$ that provides that atomic service as the composition of all $\mathcal{A}^{(i)}$ automata ($1 \leq i \leq n$), the reconfigurer $\mathcal{R}ec$ and the Γ primitive: $\mathcal{S} = \Pi_{i=1}^{n}(\mathcal{A}^{(i)}) \circ \mathcal{R}ec \circ \Gamma$. We use $\mathcal{S}$ to prove the safety of our emulation in Section 7.1.

System implementation: To evaluate the performance of the system, we define the system implementation, called $\mathcal{S}_{impl}$, as the composition of all $\mathcal{A}^{(i)}$ automata ($1 \leq i \leq n$), the reconfigurer $\mathcal{R}ec$ and the implementation Γ_{impl}: $\mathcal{S}_{impl} = \Pi_{i=1}^{n}(\mathcal{A}^{(i)}) \circ \mathcal{R}ec \circ \Gamma_{impl}$. The analysis is in Section 7.2.

We now formally define Γ and Γ_{impl} (Section 5), and the algorithms for reader/writer $\mathcal{A}^{(i)}$ and the reconfigurer $\mathcal{R}ec$ (Section 6).

5. The Broadcast/Convergecast Primitive Γ

The Γ primitive was introduced in [26], and we use it for quorum-acknowledged broadcasts in our protocol and for showing the safety of our solution. In the full paper, we also formally present the implementation of Γ, called Γ_{impl}, and suggested in [26] and we use Γ_{impl} in assessing the performance of the protocol. The Γ primitive constitutes the lower layer of the overall emulation.The service specified by the primitive provides the client with the ability to perform *quorum-acknowledged broadcasts*, and it returns to the client the results of the *condenser function* that is computed on the responses to the broadcast.

5.1. The Implementation $\Delta^{(i)}$

In the full paper we present a straightforward implementation of the Γ primitive. The implementation uses send/receive point-to-point channels. Each channel is modeled having $send(m)_{i,j}$ and $recv(m)_{j,i}$ actions, for $i, j \in PID$. Such channels have very simple specification.Recall

Data-types:

$\mathbf{N}^2$, configuration indices with selectors
act and bid (variables are x, z, cix)
$\mathcal{C}^2$, configuration pairs with selectors
act and bid (variables are X, Z)
Acknowledgment values for the query phase
are of the type $M \times Any \times \mathcal{T} \times \mathbf{N}^2 \times \mathcal{Q}^2$.
The selectors are:
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 $\in \mathbf{N}^2$, the configuration index pair
cfg $\in \mathcal{C}^2$, the quorum configuration pair
$\mathcal{U} = \mathcal{N} \times \mathcal{N} \times 2^{\mathcal{Q}}$ with selectors:
act: the index of the active quorum
bid: the index of the proposed quorum
qrm: the currently used configuration

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
$\xi \equiv \lambda(a).(\langle a[k].cix, a[k].cfg\rangle :$
$\forall j : a[k].cix \geq a[j]$.*cix*), max index and its config.

State of the reader/writer: (for each $i \in$ *PID*)
$tag \in \mathcal{T}$, initially $tag = \langle 0, 0\rangle$
$val \in V$, initially $val = v_0 \in V$
$prop\text{-}tag \in \mathcal{T}$, tag used in propagating results,
initially $\langle 0, 0\rangle$
prop-val $\in Any$, initially $\perp$
status $\in$ { *query-ready, query-active, prop-ready,*
prop-active, prop-done }, initially *query-ready*
request $\in$ { ⟨*"read"* ⟩, ⟨ ⟩} ∪ ({⟨*"write"* ⟩} × A)
ack-q, a finite sequence of $M \times ID$, initially empty
used $\in U$

Figure 2. Reader/writer $\mathcal{A}^{(i)}$ specification; Part I, data types and states

that we have defined the implementation Γ_{impl} as the composition of $\Delta^{(i)}$ and the channel automata. Main differences between Γ and each $\Delta^{(i)}$ are that in $\Delta^{(i)}$, (1) instead of the global *op*, each processor maintains a state component *op* for invocations it initiates, and (2) messages are communicated using the channels with the help of queues *out-q* and *deliver-q*. We show that Γ_{impl} *implements* Γ.

Theorem 5.1 Γ_{impl} implements Γ.

We next assess the conditional performance of Γ_{impl} (as suggested in [26]).

Theorem 5.2 Suppose in any execution of Γ_{impl}: (a) there is a constant upper bound g on the time required for a processor to read all received messages, perform a local computation, and send replies, (b) the same upper bound holds on time required for a message to be delivered if it is ever to be delivered, and (c) there exists a set of processors $Q \in q.qrm$ such that they receive the request and their condensed acknowledgments are delivered to the invoker of the *submit*. Then at most time $5g$ passes between the *submit* transition and the matching *respond* transition, and there are at most $2n$ messages sent as the result of the *submit*.

Finally note that Γ_{impl} is intentionally designed to be very simple to show that Γ is easily implementable.

6. The Shared-Memory Emulation

In this section we present the algorithm that implements the resilient multi-reader/multi-writer atomic register service. We present the solution for one emulated register. The solution is extended to multiple emulated registers by using instances of the algorithm in parallel. We present the emulation algorithms in three parts. First we describe the representation of the registers and the quorum configurations. Then we present the reader/writer automaton $\mathcal{A}^{(i)}$, and follow with the reconfigurer automaton $\mathcal{R}ec$.

6.1. Registers and Configurations

The register is represented by its value val and its tag tag, and it is replicated at all processors. Each tag is a pair consisting of a sequence number *seq* and a processor identifier *pid*. The tags are compared lexicographically ($<_{lex}$).

Each processor p also maintains pairs of quorum configurations and configuration indices. A configuration index, *cix*$_p$, is a pair of configuration sequence numbers $\langle cix.act, cix.bid\rangle_p$. They are such that *cix.act*$_p$ is the sequence number of the active configuration at p, *cix.bid*$_p$ is the sequence number of the proposed configuration at p. The indices are used to compare configurations and to detect the installation of new configurations.

Each configuration index corresponds to a configuration pair, *cfg*$_p$ = $\langle cfg.act, cix.bid\rangle_p$, where *cfg.act*$_p$ is the active configuration at p and *cfg.bid*$_p$ is the proposed configuration at p. When *cix.act*$_p$ = *cix.bid*$_p$, it implies that $cfg.act_p = cfg.bid_p$, and that the proposed configuration is accepted as active. Configuration index pairs are compared lexicographically. When $cix.act_p < cix.bid_p$, the configuration is *intermediate*. When using read or write components of intermediate configurations, processors use joins (⋈) of the appropriate components of *cfg.act*$_p$ and *cfg.bid*$_p$.

6.2. The Reader/Writer Automaton $\mathcal{A}^{(i)}$

We give the code of the algorithm for a reader/writer in Figure 2 and in Figure 3. Figure 2 shows the common datatypes and states, while Figure 3 shows the transitions.

The readers and the writers use the same algorithm. The only difference between the reads and the writes is that a writer assigns a new tag by incrementing the maximum tag found, while a reader simply uses the maximum tag.

Each read or write operation consists of two phases in each of which Γ is invoked one or more times. Iterations occur only when the used configuration, i.e., the index-pair of active and proposed configuration at the invoking processor, is less than (lexicographically) the maximum configuration index returned to the invoking processor. If a higher index is detected, it is adopted and the primitive is invoked again using the configuration(s) corresponding to the higher index pair. This ensures that obsolete configurations can be detected by a processor that wants to perform a read or a write. The first phase, *query*, uses active read quorums if $cix.act = cix.bid$ and the intermediate configuration if $cix.act < cix.bid$. The second phase, *propagation*, uses active write quorums if $cix.act = cix.bid$ and the intermediate configuration if $cix.act < cix.bid$. Variable π (subscripted when necessary) will be used to uniquely identify the client-level read or write operations occurring in some execution. Formally we make the following definition:

Definition 6.1 The phases of an operation π are defined as

Transitions of the reader/writer:

```
Inp $write(v)_p$
  Eff: $request$:= $\langle$"write", $v$ $\rangle$
Inp $read_p$
  Eff: $request$:= $\langle$"read" $\rangle$
Out $submit(\langle$"query" $,z, Z\rangle, \langle\lambda(a).($"query-ack"$), \sigma, \xi\rangle, q, id)_p$
  Pre: $status$ = $query\text{-}ready$
       $request \neq \langle\ \rangle$
       $cix.act = cix.bid \Rightarrow q = cfg.act.read$
       $cix.act < cix.bid \Rightarrow q = cfg.act.read \bowtie cfg.bid.read$
  Eff: $status$ := $query\text{-}active$
       $used := \langle cix.bid, cix.act, q\rangle$
Inp $respond(\langle$"query-ack", $\langle v,t\rangle, \langle z,Z\rangle\rangle, id)_p$
  Eff: if $\langle used.act, used.bid\rangle \geq z$ then
         if $request = \langle$"write",$w\rangle$ then
           $prop\text{-}val := w$; $prop\text{-}tag := \langle t.seq+1, p\rangle$
         else
           $prop\text{-}val := v$; $prop\text{-}tag := t$
         $status$ := $prop\text{-}ready$
       else
         $cix := z$;  $cfg := Z$
         $status$ := $query\text{-}ready$
Inp $deliver(\langle$"query", $z, Z\rangle, id)_p$
  Eff: append $\langle\langle$"query-ack", $val, tag, cix, cfg\rangle, id\rangle$ to $ack\text{-}q$
       if $z >_{lex} cix$ then
         $cix := z$; $cfg := Z$
Out $submit(\langle$"propagate", $v, t, z, Z\rangle, \langle\lambda(a).($"prop-ack"$), \xi\rangle, q, id)_p$
  Pre: $status$ = $prop\text{-}ready$
       $cix.act = cix.bid \Rightarrow q = cfg.act.write$
       $cix.act < cix.bid \Rightarrow q = cfg.act.write \bowtie cfg.bid.write$
       $v = prop\text{-}val$; $t = prop\text{-}tag$
  Eff: $status$ := $prop\text{-}active$
       $used.act := cix.act$
       $used.bid := cix.bid$
       $used.qrm := q$
Inp $respond(\langle$"prop-ack", $\langle z,Z\rangle\rangle, id)_p$
  Eff: if $\langle used.act, used.bid\rangle \geq z$ then
         $status$ := $prop\text{-}done$
       else
         $cix := z$;  $cfg := Z$
         $status$ := $prop\text{-}ready$
Inp $deliver(\langle$"propagate", $v, t, z, Z\rangle, id)_p$
  Eff: append $\langle\langle$"prop-ack", $cix, cfg\rangle, id\rangle$ to $ack\text{-}q$
       if $t >_{lex} tag$ then
         $val := v$;;  $tag := t$
       if $z >_{lex} cix$ then
         $cix := z$;;  $cfg$ := Z
Inp $deliver(\langle$"query-install", $z, Z\rangle, id)_p$
  Eff: append $\langle\langle$"install-ack", $val, tag\rangle, id\rangle$ to $ack\text{-}q$
       if $z >_{lex} cix$ then
         $cix := z$;  $cfg := Z$
Inp $deliver(\langle$"recon-done", z, Z$\rangle, id)_p$
  Eff: append $\langle\langle$"recon-ack"$\rangle, id\rangle$ to $ack\text{-}q$
       if $z >_{lex} cix$ then
         $cix := z$;  $cfg := Z$
Out $read\text{-}confirm(v)_p$
  Pre: $v$ = $prop\text{-}val$
       $status$ = $prop\text{-}done$; $request = \langle$"read"$\rangle$
  Eff: $request := \langle\ \rangle$
       $status$ := $query\text{-}ready$
Out $write\text{-}confirm_p$
  Pre: $status$ = $prop\text{-}done$
       $request = \langle$"write", $*\rangle$
  Eff: $request := \langle\ \rangle$
       $status$ := $query\text{-}ready$
Out $ack(r, id)_p$
  Pre: head($ack\text{-}q$) = $\langle r, id\rangle$
  Eff: $ack\text{-}q$ := tail($ack\text{-}q$)
```

Figure 3. Reader/writer $\mathcal{A}^{(i)}$ specification; Part II, transitions

follows: The operation π is in its $query$ phase after the transition of the $submit$ of *"query"* and prior to the $submit$ of *"propagate"*; π is in its $propagate$ phase after the transition of the $submit$ of *"propagate"* and prior to the response to its client. □

The writer (reader) accepts a client *write* (*read*) request and invokes Γ by using the *submit* action to query all processors in a read quorum (active or intermediate) for their tags. When this *query* phase completes with the *respond* action, a writer on one hand, constructs the propagation tag *prop-tag* whose sequence number is the successor of the sequence number of the maximum tag returned and whose second component is the processor's identifier. It then invokes Γ to propagate *prop-tag* and the new value *prop-val* to all processors in a write quorum (active or intermediate). A reader, on the other hand, simply invokes Γ to propagate the maximum tag and the associated value.

Each processor has a queue, *ack-q*, that is used for acknowledgments to be sent out subsequently for the corresponding *deliver* transitions.

6.3. The Reconfigurer Automaton $\mathcal{R}ec$

The reconfigurer automaton $\mathcal{R}ec$ at processor r maintains the quorum configuration sequence pair cix_r and the configuration pair cfg_r. In any global state, the configuration index at any processor p is defined to be *current*, if $cix_p \geq cix_r$. The subtle point of this definition is that if readers/writers are current, their configuration indices can not only be equal, but also be greater than the configuration indices at the reconfigurer. This is essential in proving the correctness of the emulation. In Figure 4, we define the transitions of the reconfigurer.

The reconfigurer has three phases. In each Γ is invoked once. In the *query-install* phase at *"submit"* of $query$-$install$, the *Join* of a read quorum and a write quorum in the active configuration are informed about the proposed configuration and queried about the register value with the maximum tag. In the *propagate* phase it propagates this tag and the associated value to a write quorum in the new configuration. In the *recon-idle* phase it announces to a write quorum in the new configuration that the reconfiguration is complete.

Note that the incrementing of $cix.bid_r$ occurs at *"respond"* to $install$-ack ($cix.act_r$ at *"respond"* to $recon$-ack), i.e., after the reconfigurer has received confirmation that the *Join* of a read and a write quorum (a write quorum) has received the new configuration index. This is used in the safety proofs and is the reason, why, as said above, processors can have configuration indices that are greater than the configuration indices at the reconfigurer.

We define the phases of the reconfigurer formally as follows:

Definition 6.2 The reconfigurer r is in its *query-install* phase after the transition of the *submit* of *"query-install"* and prior to the *submit* of *"propagate"*. The reconfigurer is in its *propagate* phase after the transition of the *submit* of *"propagate"* and prior to the *submit* of *"recon-done"*. The reconfigurer is in its *recon-idle* phase after the transition of the *submit* of *"recon-done"* and prior to the *submit* of the next *"query-install"*; r is also in its *"recon-idle"* phase prior to the *submit* of the first *"query-install"*. □

State of the reconfigurer r:

The components are the same as for the reader/writer, except that *request* is omitted, and *status* is given as: $status \in \{idle, new\text{-}config, query\text{-}active, query\text{-}done, prop\text{-}active, prop\text{-}done, recon\text{-}comp\}$, initially *idle*.

Transitions of the reconfigurer:

```
Inp recon$(C)_r$
  Eff:  cfg.bid := $C$
        status := new-config
Out submit(⟨"query-install", $z, Z$ ⟩, ⟨λ (a).("install-ack"), $\sigma$ ⟩, $q, id)_r$
  Pre:  status = new-config
        $z = \langle cix.act, cix.bid + 1\rangle \wedge Z =$cfg
        $q = cfg.act.read \bowtie cfg.act.write$
  Eff:  status := query-active
Inp respond(⟨"install-ack", $\langle v, t\rangle$ ⟩, $id)_r$
  Eff:  prop-val := $v$; prop-tag := $t$
        cix.bid := cix.act$+1$
        status := query-done
Out submit(⟨"propagate", $v, t$ ⟩, λ (a).("prop-ack"), $q, id)_r$
  Pre:  status = query-done
        $v =$ prop-val $\wedge\ t =$ prop-tag
        $q =$ cfg$.bid.write$
  Eff:  status := prop-active
Inp respond(⟨"prop-ack" ⟩, $id)_r$
  Eff:  status := prop-done

Out submit(⟨"recon-done", $z, Z$ ⟩,
        λ (a).("recon-ack"), $q, id)_r$
  Pre:  status = prop-done
        $z = \langle cix.bid, cix.bid\rangle$
        $Z = \langle$cfg$.bid$, cfg$.bid\rangle$
        $q =$ cfg$.bid.write$
  Eff:  status := recon-comp
Inp respond(⟨"recon-ack" ⟩, $id)_r$
  Eff:  status := recon-ready
        cix := ⟨cix.bid, cix.bid ⟩
        cfg := ⟨cfg.bid, cfg.bid ⟩
Out recon-confirm$_r$
  Pre:  status = recon-ready
  Eff:  status := idle
```

(The actions *deliver* and *ack* are identical to those in Figure 3.)

Figure 4. Specification of the reconfigurer $\mathcal{R}ec$.

7. System Analysis

In this section we show correctness of the emulation algorithms and assess the system performance. To show the atomicity of the emulation we use the system $\mathcal{S}$. The performance is shown for the implementation $\mathcal{S}_{impl}$.

7.1. Correctness (Safety)

We state the atomicity theorem for the system $\mathcal{S}$ and outline its proof. The proof uses an approach similar to [26], and is the most technically challenging part of this work (for the complete proof see the full paper).

Theorem 7.1 $\mathcal{S}$ implements an atomic multi-writer multi-reader register.

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

A client-level read (write) operation is invoked by its corresponding *read* (*write*) event. The response event of the read (write) operation is its corresponding *read-confirm* (*write-confirm*) event. We define *CP*, the *client-preceding* order as follows:

Definition 7.1 If in an execution α, the confirm event of the operation π_1 precedes the request event of the operation π_2, then $\pi_1 <_{cp} \pi_2$.

Suppose for some execution α the actions of an operation π include the actions of Γ for some id, starting with the *submit* event and including the *respond* event. Then since the identifier id is unique it also uniquely identifies the client-level operation π. Therefore we can let $\tau(id)$ stand for $\tau(\pi)$, (where the propagation tag $\tau(\pi)$ is defined for an operation π in the *respond* action uniquely identified by id, or when $\tau(\pi)$ is propagated by the Γ primitive using unique identifier id.)

We show the atomicity of the read and write operations for any execution by using Lemma 13.16 of [25]. To be able to use this Lemma, we show that there exists a partial order of read and write operations in a sequence of actions of a read/write register that satisfies certain conditions.

To simplify our proofs of the correctness of $\mathcal{S}$, we use a succinct and effective way of expressing the eventuality of certain outcomes based on the current knowledge, a "*fill*" notion (also used in [26]). "*fill*" predicts the acknowledgment vector for a current invocation and thereby allows us to simplify our invariants and reduces the size of their proofs.

The *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 value is the queued acknowledgment; if the *deliver* has not occurred, then the value is the acknowledgment that would be produced if the deliver occurred as the next event.

Definition 7.2 For the invocation of the Γ primitive with the unique identifier id, let $\mu_p : M \times States_p \rightarrow M$ 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 *fill*$(p, id) \equiv$

if $op(id) = \perp$ **then** $\perp$
else if $p \in op(id).acks$ **then** $op(id).acc[p]$
else if $\exists m : \langle m, id\rangle \in ack\text{-}q_p$ **then** (the unique) m
else $\mu_p(m, state_p)$

The key to the proof is a multi-part invariant Lemma 7.2, shown in Figure 5 which we now explain and whose proof is in the full paper. Part **I3** is the most important part; 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** says that for any completed read or write operation π, if no new quorum system is being processed by the reconfigurer r, then there exists a proposed write quorum such that all processors in it reflect either π or some other operation that supersedes it.

Lemma 7.2 In all reachable states:

I1 If $\pi \in completed$, then:

(a) If r is in its *recon-idle* phase, then:
$\exists W \in$ *cfg.bid*$.write_r : \forall i \in W : \tau(\pi) \leq_{lex} tag_i$

(b) If r is in its *query-install* phase having invoked Γ using identifier oid_r, then: $\forall R \in$ *cfg.act*$.read_r$: $\tau(\pi) \leq_{lex} \max_{i \in R}\{fill(i, oid_r).tag\}$

(c) If r is in its *propagate* phase having invoked Γ using identifier oid_r and tag τ_r, then:

1. $(\tau(\pi) \leq \tau_r) \wedge (\forall i \in op(oid_r).acks : \tau_r \leq_{lex} tag_i)$

or

2. $\exists W \in$ *cfg.bid.write*$_r$: $\forall i \in W$: $\tau(\pi) \leq_{lex} tag_i$.

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 $R \in$ *used*$.qrm_p$, either

1. $\tau(\pi') \leq_{lex} \max_{i \in R}\{fill(i, oid).tag\}$, **or**
2. $\langle used.act, used.bid\rangle_p <_{lex}$ $\langle \max_{i \in R}\{fill(i, oid_p).cix.act\}$, $\max_{i \in R}\{fill(i, oid_p).cix.bid\}\rangle$.

(b) If π is in the *propagation* phase having invoked Γ using identifier oid, either:

1. $\langle used.act, used.bid\rangle_p$ is *current*, **or**
2. $\exists W \in cfg.bid.write_r$: $\forall i \in W : \tau(\pi) \leq_{lex} tag_i$, **or**
3. $\forall W \in used.qrm_p$:$\langle used.act, used.bid\rangle_p <_{lex}$ $\langle \max_{i \in W}\{fill(i, oid).cix.act\}$, $\max_{i \in W}\{fill(i, oid).cix.bid\}\rangle$.

I3 If $\pi_1 \prec_{cp} \pi_2$ and $\tau(\pi_2)$ is defined, then:

(a) $\tau(\pi_1) \leq_{lex} \tau(\pi_2)$ when π_2 is a read,

(b) $\tau(\pi_1) <_{lex} \tau(\pi_2)$ when π_2 is a write.

Figure 5. Main Invariant

Part **I1b** says that if the reconfigurer r has 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** says that if the reconfigurer r has 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 **or** that there is a proposed write quorum such that all processors in it reflect either π or some other operation that supersedes it.

Part **I2a** says 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 an operation preceding this one in the client-preceding order – this is expressed with the help of the *fill* notation, or (2) the operation will detect that its configuration is obsolete – the guarantee of detection is expressed using *fill*.

Part **I2b** says that for any operation π in its *propagation* phase, at least one 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 will detect that its configuration is obsolete – again this guarantee of detection is expressed using *fill*.

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

The partial order required by Lemma 13.16 [25] and yielding the proof of Theorem 7.1 is now constructed in the following way: For an execution α of $\mathcal{S}$ containing no incomplete operations, let sequence β be the projection of α containing the invocation/response events of read and write operations. Let Π be the set of operations in β. 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 if (b) π_1 is a write, π_2 is a read, and $\tau(\pi_1) =_{lex} \tau(\pi_2)$.

7.2. Conditional Performance Analysis

In this section we assess the performance of the composed system, called $\mathcal{S}_{impl}$, that differs from $\mathcal{S}$ only in that it uses Γ_{impl} instead of Γ (recall that Γ_{impl} formally implements Γ). For $\mathcal{S}_{impl}$ we show that the reconfiguration is not obstructed by the concurrent client level read/write operations, and, more importantly, that the client level read/write operations are not obstructed by a tardy or stopped reconfigurer – the reconfiguration is *graceful*.

To assess the performance of the atomic multi-writer/multi-reader service, we carry over the assumptions of Theorem 5.2. In particular, we use the same upper bound g on local processor operations and message delays, and we assume that the quorum systems are not disabled.

The next theorem shows that the performance of any reconfiguration does not depend on any client-level operations.

Theorem 7.3 Any reconfiguration takes time at most $15g$ and at most $6n$ messages.

Finally we show that reads and writes are not obstructed by reconfigurations.

Definition 7.3 Let cix_1 and cix_2 be two configuration index pairs. We define the *distance* $dist(cix_1, cix_2)$ between these pairs as $|(cix.act_1 + cix.bid_1) - (cix.act_2 + cix.bid_2)|$.

Intuitively, given an execution of the system, the distance between any two configuration index pairs gives the number of different configurations (including intermediate) that came into existence since the least (with respect to the lexicographical order) of these two index pairs became known.

Theorem 7.4 In any execution, if (a) all reconfigurations complete by time t **or** if (b) a reconfiguration does not complete by time t, but the reconfigurer does not perform any further steps after time t, then any processor p that does not permanently stop, completes its read or write operation in time $10g + dist(cix_p, cix_t) \cdot 5g$, using at most $(2\ dist(cix_p, cix_t) + 4) \cdot n$ messages, where cix_p is the configuration index pair at p at the invocation of the operation and cix_t is the maximum of all index pairs that exist anywhere in the system at t.

Theorem 7.5 If an execution contains a read or a write operation at processor p that starts at time t_1 and completes at time t_2, then $t_2 - t_1 \leq 10g + (dist(cix_p, cix_{t_1}) + 2c) \cdot 5g$, and the number of messages sent is at most

$(4 + 2\, dist(cix_p, cix_{t_1}) + 4c) \cdot n$, where cix_p is the configuration index pair at p at t_1 and cix_{t_1} is the maximum of all index pairs that exist anywhere in the system at t_1, and c is the number of reconfigurations that are concurrent with the operation.

8. Conclusions and Discussion

We presented a robust service that emulates atomic multi-writer/multi-reader registers in message passing systems. The service uses dynamically reconfigurable quorum systems to ensure atomicity in a way that does not rely on locking or mutual exclusion. The reconfigurations are *graceful*, since they do not obstruct concurrent reads or writes. The emulation works with any quorum system and does not require that any two quorums from distinct quorum systems have a non-empty intersection. The emulation contains no single points of failure.

Acknowledgments: We thank Maurice Herlihy and Nancy Lynch for insightful comments, and Idit Keidar for several valuable observations.

References

[1] D. Agrawal and A. El Abbadi, “Resilient Logical Structures for Efficient Management of Replicated Data”, TR, Univ. of California Santa Barbara, 1992.

[2] Y. Amir, A. Wool, “Evaluating Quorum Systems over the Internet”, Proc. of 26th Intl. Symp. on Fault-Tolerant Computing, pp. 26-35.

[3] L. Alvisi, D. Malkhi, L. Pierce, and M. Reiter, “Fault detection for Byzantine quorum systems”,(extended abstract), Proc. of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications, 1999.

[4] H. Attiya, “Efficient and Robust Sharing of Memory in Message-Passing Systems”, *Distributed Algorithms, 10th International Workshop, WDAG’96*, 1996.

[5] 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.

[6] M. Bearden, R. P. Bianchini Jr., “The Synchronization Cost of On-line Quorum Adaptation”, in *10th (ISCA) International Conference on Parallel and Distributed Computing Systems (PDCS’97)*.

[7] M. Bearden, R. P. Bianchini Jr., “A Fault-tolerant Algorithm for Decentralized On-line Quorum Adaptation”, in *Proc. 28th Intl. Symp. on Fault-Tolerant Computing Systems*.

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

[9] A. Bouabdallah, “On Mutual Exclusion in Faulty Distributed Systems”, in *Operating Systems Review (ACM SIGOS)*, 28(1), Jan. 1994, pp. 80-87.

[10] R. De Prisco, A. Fekete, N. Lynch and A. Shvartsman, “A Dynamic View-Oriented Group Communication Service”, in *Proc. of 16th ACM Symp. on Principles of Distributed Computing*, 1998.

[11] R. De Prisco, A. Fekete, N. Lynch, A. Shvartsman, “A Dynamic Primary Configuration Group Communication Service”, to appear in *13th International Conference of Distributed Computing*, 1999.

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

[13] A. El Abbadi, D. Skeen and F. Cristian, “An Efficient Fault-Tolerant Protocol for Replicated Data Management”, in *Proc. of the Fourth ACM Symp. on Princ. of Databases*, pp. 215-228, 1985.

[14] A. El Abbadi and S. Toueg, “Maintaining Availability in Partitioned Replicated Databases”, *ACM Trans. on Database Systems*, vol. 14, no. 2, pp. 264-290, 1989.

[15] 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.

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

[17] K. Goldman and N. Lynch, “Nested Transactions and Quorum Consensus”, in *Proc. of the 6th ACM Symp. on Princ. of Distr. Comput.*, pp. 27-41, 1987

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

[19] M.P. Herlihy, “Dynamic Quorum Adjustment for Partitioned Data”, *ACM Trans. on Database Systems*, 12(2), June 1987, pp. 170-194.

[20] M.P. Herlihy and J.M. Wing. “Linearizability: A correctness condition for concurrent objects”, *ACM TOPLAS*, vol. 12, no. 3, pp. 463-492, 1990.

[21] S. Jajodia and D. Mutchler, “Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database”, in *ACM Trans. Database Systems*, 15(2):230-280, 1990.

[22] E. Lotem, I. Keidar, and D. Dolev, “Dynamic Voting for Consistent Primary Components”, in *Proc. ACM Symp. on Principles of Distributed Systems*, 1997.

[23] L. Lamport, “On Interprocess Communication: Part I and II”, *Dist. Comput.*, vol. 1, pp. 77-101, 1986.

[24] M. Liu, D. Agrawal and A. El Abaddi, “On the Implementation of the Quorum Consensus protocol”, *Proc. Parallel and Distributed Computing Systems*, 1995.

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

[26] N.A. Lynch and A.A. Shvartsman, “Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts”, in *Proc. 27th Intl. Symp. on Fault-Tolerant Computing Systems*, 1997.

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

[28] D. Malki and M. Reiter, “Byzantine Quorum Systems”, In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pp. 569-578, 1997.

[29] 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.

[30] D. Peleg and A. Wool, “The Availability of Quorum Systems”, *Information and Computation*, 123(2), Dec. 1995, pp. 210-223.

[31] S. Rangarajan, S. Tripathi, “A Robust Distributed Mutual Exclusion Algorithm”, *Distributed algorithms, Proceedings 5th Intl. Workshop,* WDAG ’91, Delphi, 1991, Springer-Verlag, pp. 295-308.

[32] B. Sanders, “The Information Structure of Distributed Mutual Exclusion Algorithms”, *ACM Transactions on Computer Systems*, 5(3), Aug. 1987, pp.284-299.

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

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