Multi-writer Regular Registers in Dynamic Distributed Systems with Byzantine Failures

Silvia Bonomi, Amir Soltani Nezhad
Università degli Studi di Roma "La Sapienza",
Via Ariosto 25, 00185 Roma, Italy
bonomi@dis.uniroma1.it
amir.soltaninezhad@gmail.com

MIDLAB TECHNICAL REPORT 7/11 - 2011

Abstract

In this paper, we address the problem of building of a multi-writer/ multi-reader regular register storage resilient to byzantine failures in a distributed system affected from churn. A protocol implementing such a register in a synchronous system is proposed and some feasibility constraints on the arrival and departure of processes are given. The protocol is proved to be correct under the assumption that the constraint on the churn is satisfied, and we show that the implementation satisfies the wait-freedom property as soon as the number of writers is bounded and known.

Keywords: Churn, Dynamic system, Multi-writer Regular Register, Byzantine Failures, Synchronous System, Wait-Freedom.

1 Introduction

In recent years, the ever cheaper and more powerful hardware, together with the always increasing availability of bandwidth led *software as a service* computing paradigm to transform many distributed applications into services offered by clouds providers. Among all the possible cloud services, distributed storage like Amazon Simple Storage Service (S3) [4] is one of the most popular ones, due to its capability to provide simple read and write interfaces ensuring a certain degree of consistency and a well defined service availability level. Note that, usually such a kind of services is regulated by specific contracts (i.e. *Service Level Agreement*), and thus service providers must guarantee such a level of quality of service despite any types of failures including malicious ones.
A common approach to ensure storage availability is to keep a fixed number of replicas each one hosted at a separate server aligned, and many protocols have been proposed to build byzantine-fault-tolerant (BFT) storage services on top of a message-passing system. However, they do not consider the possibility to have changes in the set of servers hosting replicas. Servers can leave due to the ordinary or unexpected maintenance procedures, and new replicas need to be set up (i.e. join) in order to maintain a minimum number of active replicas needed to provide the service. These changes caused by joins and departures of servers (*churn* phenomenon), if not properly mastered, can either block protocols or violate the safety of the storage.

In this paper, we consider a distributed system composed of n servers implementing a distributed storage service, where at any given time t, the number of servers that can be inactive (i.e. the number of servers that is joining or leaving the service) is limited to a certain percentage cn, where $c \in [0, 1]$. In this environment, we present a BFT implementation of a distributed storage offering the multi-writer regularity semantics, which is able to resist the bounded churn and tolerates up to f byzantine failures. Moreover, we will prove that if the protocol works under synchrony assumptions, it ensures the wait-freedom property as soon as the number of writers is bounded by a finite integer m.

The rest of the paper is contributed as follows: in Section 2, we define the system model. Section 3 provides the multi-writer/multi-reader regular register specification while in Section 4, we detail the algorithm and the correctness proofs. Section 5 presents the related works, and finally Section 6 concludes the paper.

2 System Model

Distributed System. We consider a distributed system composed of a *universe of clients* U_c (i.e. the client system) and of a disjoint *universe of servers* U_s (i.e. the server system).
The client system is composed of an arbitrary number of processes (i.e. $U_c = \{c_1, c_2, \dots c_m\}$) while the server system is dynamic, i.e. processes may join and leave the system at their will. In order to model processes continuously arriving to and departing from the server system, we assume the *infinite arrival model* [13], i.e. the set of processes that can participate in the server system (also called *server-system population*) is composed of a potentially infinite set of processes $U_s = \{\dots s_i, s_j, s_k \dots\}$. However, the server system is composed, at each time, of a finite subset of the server-system population. We assume that each server in the server-system population has a unique identifier (i.e. its index).
A server enters the server system by executing the connect() procedure. Such an operation aims at connecting the new process to both clients and servers, which already belong to the system. A server leaves the distributed system by means of the disconnect() operation. In the following, we will assume that the disconnect() operation is a passive operation i.e., processes do not take any specific actions, and they just stop executing algorithms.
Initially, every server $s_i \in U_s$ is in the *down* state; s_i changes its state from *down* to *up* as soon as it invokes the connect() operation. When a server s_i disconnects itself from the server system, it changes again its state coming back to *down*.
Processes belonging to the distributed system (both clients and servers) communicate only by exchanging messages on top of authenticated communication primitives. As in [5], in the following, we assume the existence of a protocol managing the arrival and the departure of servers in the distributed system; such a protocol is also responsible for the connectivity maintenance among the processes belonging to the distributed system.
Synchronous system. The distributed system is synchronous. In particular, (i) there exists a known upper bound on processing delays (i.e. the duration of every computational step can be bounded) and (ii) processes are equipped with a broadcast and a point-to-point communication primitive, which have the following specifications:

- There exists a known and finite bound δ such that every message broadcast at some time t is delivered up to time $t + \delta$ (TimelyBroadcastDelivery).
- There exists a known and finite bound $\delta' < \delta$ such that every message sent at some time t is delivered up to time $t + \delta'$ (TimelyChannelDelivery).

Distributed Computation. A distributed computation run on top of the distributed system involve the participation of a subset of the up servers. We identify as $C_s(t)$ the subset of up servers of U_s that participate in the distributed computation at time t (i.e. the *server-computation set*). We assume that at time t_0, when the server-computation set is set up, n servers belong to the server computation (i.e. $|C_s(t_0)| = n$).
When a server s_i wants to join the distributed computation, it has to execute the join() operation. We assume that each server s_i can invoke a join() operation if and only if it is *up* and it has terminated the connect() procedure. Let us note that a join() operation, invoked at some time t, is generally not instantaneous and takes time to be executed: how much this time is, depends on the specific implementation provided for the join() operation. However, from time t when the server s_i joins the server-computation set, it can receive and process messages sent by any other processes participating in the computation, and it changes its state from *up* to *joining*.
A server s_i remains in the *joining* state until it terminates the execution of the join() operation (i.e. until it gets the join_Confirmation event), and we denote as $J(t)$ the set of servers that are in the joining state at time t. As soon as the join_Confirmation event occurs, s_i changes its state from *joining* into *active*. In the following, we will denote as $A(t)$ the set of *active servers* at time t.
When a server s_j participating in the distributed computation wishes to leave, it stops executing the server protocols (i.e. the leave operation is passive) and comes back to the *up* state. Without loss of generality, we assume that if a server leaves the computation and later wishes to re-join, it executes again the join() operation with a new identity. In Figure 1 it is shown the state-transition diagram of a correct server.

Let us note that at each time t, the set of servers participating in the distributed computation is partitioned into active processes and joining processes. i.e.

$$C_s(t) = A(t) \cup J(t).$$

Moreover, let us remark that (i) there may exist processes belonging to the server system, which never join the distributed computation (i.e. they execute the connect() procedure, but they never invoke the join() operation), and (ii) there may exist processes that even after leaving the server computation, still remain inside the server system.
Failure Model. As in [11] and [12], we assume that clients can fail only by crashing, while servers can suffer arbitrary failures. Servers that obey their specification are said to be *correct*. On the contrary, a *faulty* server can deviate arbitrarily from its specification. We assume at most f servers can be faulty at any time during the

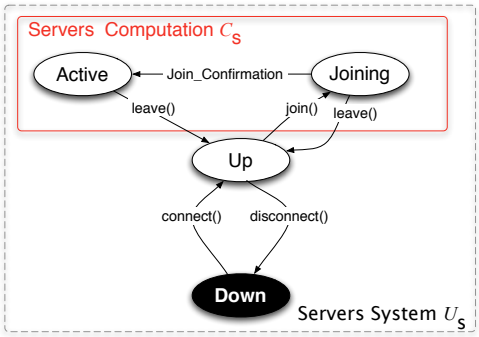

Figure 1: State-transition diagram of a Correct Server

whole computation. It is important to note that servers know the value f, but they are not able to know the subset of $C_s(t)$ representing the faulty processes.

Non-Quiescent Churn. We assume that at time t_0 at the beginning of the computation, n processes participate and are active in the computation (i.e. $|A(t_0)| = n$). Then, the server computation starts, and it can alternate periods of churn and periods of stability. More specifically, there exist some periods T_{churn} in which servers join and leave the computation, and then there exist some periods $T_{stability}$ where the computation becomes stable, and no join or leave operations are triggered. However, no assumption is made about how long T_{churn} and $T_{stability}$ are. We assume that during each churn period T_{churn}, the churn is continuous, that is at each time $t \in T_{churn}$, cn processes leave the computation and cn invoke the join() operation (where $c \in [0, 1]$ is a percentage of servers).

As a consequence, the number of servers participating in the server-computation set remains always constant and equal to n. However, considering that (i) leave operations as passive (i.e. they are instantaneous) and (ii) join operations takes time, the set of active servers, i.e. the set of processes that effectively stores the value of the register, has a variable size (i.e. each time $t \in T_{churn}$, $|A(t)| \leq n$.

Let us finally remark that in this churn model, there is no guarantee that a server remains permanently in the computation and additionally, this model is general enough to encompass both (i) a distributed computation prone to continuos churn i.e., there exists a time t (with $t = t_0$) after which churn holds forever, and (ii) a distributed system prone to quiescent churn i.e., there exists a time t after which stability holds forever.

3 Multi-Writer/Multi-Reader Regular Registers

A register is a shared variable accessed by a set of processes through two operations, namely read() and write(), that allows them to read the value contained in the variable or to modify such a value. Registers have been introduced by Lamport [9] and in this paper, we will consider a multi-writer/multi-reader regular register as specified in [15].

Basic Definitions. Each register operation can be characterized by two events occurring at its boundary: an $invocation$ event and a $reply$ event. These events occur at two time instants (invocation time and reply time). According to these time instants, it is possible to state when two operations are concurrent with respect to the real time execution.

Given two operations op and op', having respectively invocation times $t_B(op)$ and $t_B(op')$ and return times $t_E(op)$ and $t_E(op')$, we say that op *precedes* op' ($op \prec op'$) iff $t_E(op) < t_B(op')$. If op does not precede op' and op' does not precede op then they are *concurrent* ($op||op'$). Given an operation op, we will say that op is *completed* if both the $invocation$ event and a $reply$ event occur, otherwise we will say that it is $failed$. We will consider a failed operation op as concurrent with all the other operations started at some $t > t_B(op)$. Considering all the operations H invoked on the register and the precedence relation introduced so far, it is possible to define the *execution history*, denoted as $\hat{H} = (H, \prec)$, as a partial order between the operations of H induced by the precedence relation.

Note that, for a given execution history containing concurrent operations, it is possible to find several linearizations where concurrent operations are ordered differently. In the following, we introduce the concepts of *consistent permutation* to identify these linearizations and the notion of *legal permutation* to identify the permutations that are *valid* for the register semantics (i.e. where a read returns the value written by the last write in the sequence).

Definition 1 (Permutation π Consistent with $\hat{H}$) *Let $\hat{H} = (H, \prec)$ be the execution history of a register $\mathcal{R}$, then a* permutation *π of operations belonging to H is* consistent *with $\hat{H}$ if, for any pair of operations op, op' in π, op precedes op' in π whenever op precedes op' in $\hat{H}$.*

Definition 2 (Legal Permutation π) *Let $\hat{H} = (H, \prec)$ be the execution history of a register $\mathcal{R}$ and let π be a* permutation *of operations belonging to H, then π is* legal *if any* read()/join() *operation op in π returns the value written by last* write() *preceding respectively op in π.*

In order to simplify the notation, for any given read()/join() operation op, let us denote as $write(op)$ the set of all the write() operations w, issued on the register,

such that w does not follow op in the execution history (i.e. $write(op) = \{w \in H | w$ is a write $\wedge\ (t_B(w) < t_E(r))\}$).

Definition 3 (Multi-Writer Regularity 2 (MWR2)) *Let $\hat{H} = (H, \prec)$ be an execution history of a multi-writer regular register and let π be a permutation of the operations in H. Let $r \in H$ be a* read() *operation issued by a client c_i and let π_r be the projection of π onto $write(r) \cup \{r\}$, then r satisfies MWR2 if :*

- *π_r is legal*
- *π_r is consistent with $\hat{H}$*

Informally to satisfy MWR2, we require that for each pair of read() operations, the write() operations, which do not follow both of them are perceived in the same order.

Specification In the following, we will say that an algorithm implements a MW-MR regular register if the following conditions hold:

- Termination: If a correct client issues an operation on the register, it eventually returns from that operation.
- MW- Validity: any read() operation satisfies MWR2.

Moreover, we will say that an algorithm satisfies the wait-freedom property if and only if it implements the register guaranteeing that any process (both client or server) can complete any operation in a finite number of steps, regardless of the execution speeds or failures experienced by other processes [8].

4 Regular Register Implementation

The principles behind the design of our algorithms are (i) minimizing the interactions in the set of servers to avoid faulty ones to compromise the state of the register and (ii) providing algorithms requiring the minimum number of communication steps, to minimize the number of replicas necessary to ensure the register availability. In particular, in order to satisfy MWR2 consistency, write() operations are totally ordered according to the pair $< sn, id >$ where sn is the sequence number of the operation, and id is the identifier of the client issuing the operation.
Moreover, in order to enforce the wait-freedom property of our algorithm, in the following we will assume that:

1. up to m clients can write on the regular register while all the clients are allowed to read.

2. At time t_0 all the servers belonging to the distributed computation know the set of writers.

Local variables at a client c_i Each client c_i maintains just one local variable, denoted $cl\_replies_i$, where it stores the replies received from servers during the execution of the read() operation. Moreover, the writer clients also maintain the sn_i integer variable representing the sequence number to associate to each write() operation.

Local variables at a server s_i Each server s_i maintains the following local variables.

- Two variables denoted $value_i$ and sn_i; $value_i$ contains the local copy of the regular register, while sn_i is the associated with the sequence number.
- A boolean $active_i$, initialized to $false$, that is switched to $true$ just after s_i has joined the system.
- A set variable $writers_i$ used during write() operations, where s_i stores the identities of the writers clients.
- A variable denoted $last\_writer_i$ where s_i stores the identity of the writer that has updated the register more recently.
- Two set variables, denoted $replies_i$ and $reply\_to_i$, that are used in the period during which s_i joins the system. The local variable $replies_i$ contains the 3-uples $< id, value, sn >$, which s_i has received from other processes during its join period, while $reply\_to_i$ contains the processes that are joining the system concurrently with s_i (as far as s_i knows).

The join() **operation.** The algorithm implementing the join operation is described in Figure 2. The server s_i first initializes its local variables (line 01), and waits for a period of δ time units (line 02). This waiting period is necessary to avoid the server to lose messages related to possibly concurrent write operations (cfr. [5] for details).

If $value_i$ has not been updated during this waiting period (line 03), s_i broadcasts (with the broadcast() operation) an INQUIRY(i) message to the servers that are in the computation (line 05) and waits for 2δ time units, i.e., the maximum round trip delay (line 06)[1].

When this period terminates, s_i updates its local variables $value_i$, sn_i and $last\_writer_i$

[1]The statement wait(2δ) can be replaced by wait$(\delta + \delta')$, which provides a more efficient join operation; δ is the upper bound for the dissemination of the message sent by the reliable broadcast that is a one-to-many communication primitive, while δ' is the upper bound for a response that is sent to a process whose id is known, using a one-to-one communication primitive. So, wait(δ) is related to the broadcast, while wait(δ') is related to point-to-point communication. We use the wait(2δ) statement to make easier the presentation.

```
operation join(i):
(01) value_i ← ⊥; sn_i ← −1; active_i ← false;
     writers_i ← ∅; last_writer_i ← ⊥;
     replies_i ← ∅; reply_to_i ← ∅;
(02) wait(δ);
(03) if (value_i = ⊥) then
(04)   replies_i ← ∅;
(05)   broadcast INQUIRY(i);
(06)   wait(2δ);
(07)   < j, val, sn, lw > ← select_most_recent(replies_i)
(08)   if (sn > sn_i)
(09)     then sn_i ← sn;
(10)          value_i ← val;
(11)          last_writer_i ← lw;
(12)   end if
(13) end if;
(14) active_i ← true;
(15) for each j ∈ reply_to_i do
(16)   send REPLY (< i, value_i, sn_i, last_writer_i >) to p_j;
(17) endfor
(18) return(join_Confirmation).
-------------------------------------------------------------------
(19) when INQUIRY(j) is delivered:
(20)       if (active_i)
(21)         then send REPLY (< i, value_i, sn_i, last_writer_i >) to p_j
(22)         else reply_to_i ← reply_to_i ∪ {j}
(23)       end if.

(24) when REPLY(< j, value, sn, lw >) is received:
(25)       replies_i ← replies_i ∪ {< j, value, sn, lw >, }
```

Figure 2: The join() protocol for a synchronous system (server code)

to the values obtained from the select_most_ recent() function. More in detail, such a function considers every 4-tuple contained in the set $replies_i$ and selects the value v such that there exists at least $f+1$ occurrences of the pair $< v, sn >$ in $replies_i$. In the case that more than one pair appear in the set with $f+1$ occurrences, it selects the pair having the highest pair $< sn, lw >$ according to the lexicographic order (lines 07-12), on the contrary, if there not exists a pair $< v, sn >$ in $replies_i$ occurred at least $f+1$ times, it returns the value null.
After s_i got a value, it becomes active (line 14), which means that it can answer the inquiries it has received from the other servers, and does it if $reply\_to \neq \emptyset$ (line 15). Finally, s_i returns the join_Confirmation event to indicate the end of the join() operation (line 18).

When a server s_i receives a message INQUIRY(j), it answers s_j by sending back a REPLY$(< i, register_i, sn_i, last_writer_i >)$ message containing its local variable if it is active (line 21). Otherwise, s_i postpones its answer until it becomes active (line 22 and lines 14-15). Finally, when s_i receives a message REPLY(<

$j, value, sn, lw >$) from a server s_j it adds the corresponding 4-tuple to its set $replies_i$ (line 25).

The read() **operation.** The algorithms for the read operation associated with the regular register (both client and server side) are described in Figure 3.

After having initialized its local variables, the client c_i broadcasts a READ(i) message to make inquiries from the servers about the current value of the regular register (line 02) and waits for 2δ time units, i.e., the maximum round trip delay (line 03). When this period terminates, c_i selects the value for which it has at least $f + 1$ same replies (line 04). Finally, C_i returns the value (line 06).

```
operation read(i):
(01) cl_replies_i ← ∅;
(02) broadcast READ(i);
(03) wait (2δ);
(04) < j, val, sn > ← select_most_recent(cl_replies_i)
(05) sn_i ← sn;
(06) return(val).
-------------------------------------------------------
when REPLY(< j, val, sn >) is delivered:
(07) cl_replies_i ← cl_replies_i ∪ {< j, val, sn >};
```

(a) Client Protocol

```
when READ(j) is delivered:
(01) if (active_i)
(02)    then send REPLY (< i, value_i, sn_i >) to p_j;
(03)    else reply_to_i ← reply_to_i ∪ {j};
(04) end if.
```

(b) Server Protocol

Figure 3: The read() protocol for a synchronous system

When a server delivers a READ(j) message, it answers by sending back a REPLY message containing its local copy of the register together with the sequence number if it is active (line 02), otherwise it postpones its answer until it becomes active (Figure 3 line 03 and Figure 2 line 16).

Finally, when a client c_i delivers a REPLY message, it puts the received value and sequence number in its $cl\_replies_i$ set (line 07).

The write() **operation.** The algorithms for the write operation associated with the regular register (both client and server side) is described in Figure 4. The write() operation is initiated from a writer client c_w that first issues a read() operation to obtain an updated sequence number, and then it disseminates to all the servers currently in the computation, the pair $< value, sn >$ (lines 02 - 03). In order to guarantee the correct delivery of that value, the writer is required to wait for δ time units before terminating the write operation (line 04).

```
operation write(v):
(01)  read(i);
(02)  sn_i ← sn_i + 1;
(03)  broadcast WRITE(i, < v, sn_w >);
(04)  wait (δ);
(05)  return(ok).
```

(a) Client Protocol

```
when WRITE(j, < val, sn >) is delivered:
(01)     if ((j ∈ writers_i) ∧ ((sn, j) > (sn_i, last_writer_i))
(02)       then value_i ← val;
(03)            sn_i ← sn;
(04)            last_writer_i ← j;
(05)     end if;
```

(b) Server Protocol

Figure 4: The write() protocol for a synchronous system

When a message WRITE$(j, < val, sn >)$ is delivered to a server s_i, it checks if the message comes from the writer client (line 01) and if it so, it takes into account the pair (val, sn) if it is more up-to-date than its current pair (lines 02-05).

Due to the lack of space, we report here only the main theorem, and we omit the proofs that can be found in the appendix A.

Theorem 1 Termination. *If a server invokes the* join() *operation, and does not leave the system for at least* 3δ *time units, or a client invokes the* read() *operation, or invokes the* write () *operation and does not crash, then it terminates the invoked operation.*

Theorem 2 *Let* $\mathcal{R}$ *be a regular register and let* $\hat{H} = (H, \prec)$ *be an execution history of* $\mathcal{R}$ *generated by the algorithm in Figures 2 - 4. If* $c < \frac{n-(m+2)f}{4\delta n}$ *then any* read() *operation satisfies MWV2.*

5 Related Work

To the best of our knowledge, this is the first algorithm implementing a regular register with multiple writers in the presence of both churn and byzantine failures.

Byzantine fault tolerant systems based on quorums. Traditional solutions to build byzantine storage can be divided into two categories: replicated state machines [14] and byzantine quorum systems [11], [12]. Replicated state machines uses $2f + 1$ server replicas and require that every non-faulty replica agrees to process requests in the same order [14]. Quorum systems, introduced by Malkhi-Reiter in [11], do not rely on any form of agreement, and they just need a subset of the replicas (i.e. *quorums*) to be involved simultaneously. The authors provide

a simple wait-freedom implementation of a safe register using $5f$ servers. [3] proposes a protocol for implementing a single-writer and multiple-reader atomic register that holds wait-freedom property using just $3f + 1$ servers. This is achieved at the cost of longer (two phases) read and write operations.

Registers under quiescent churn. In [10], [7] and [6], a Reconfigurable Atomic Memory for Basic Object (RAMBO) is presented. RAMBO works on top of a distributed system where processes can join and fail by crashing. To guarantee the reliability of data, in spite of network changes, RAMBO replicates data at several network locations and defines *configurations* to manage small and transient changes. For the large changes in the set of participating processes, RAMBO defines a *reconfiguration* procedure whose aim is to move the system from an existing configuration to a new one by changing the membership of the read quorums and of the write quorums. Such a reconfiguration is implemented by a distributed consensus algorithm. Thus, the notion of churn is abstracted by a sequence of configurations. Moreover, to ensure liveness of the system, RAMBO assumes that there exist stability periods long enough to allow the algorithm to converge (i.e., assumption of quiescent churn).

In [1] Aguilera et al. show that a crash resilient atomic register can be realized without consensus, and thus on a fully asynchronous distributed system provided that the number of reconfigurations is finite, and thus the churn is quiescent. Configurations are managed by taking into account any changes (i.e. join and failure of processes) suggested by the participants and the quorums are represented by any majority of processes. To ensure the liveness of read and write operations, the authors assume that the number of reconfigurations is finite, and that there is a majority of correct processes in each reconfiguration.

6 Conclusion and Future Work

This paper presented an implementation of a multiple-readers/multiple-writers regular register variable on top of a distributed server system characterized by both churn and byzantine failures. We have shown that our algorithm works in the presence of bounded churn and bounded communication delays. Moreover, we have shown that the algorithms provide wait-freedom guarantees as soon as the number of writers is bounded. As a future work, we are investigating how to extend the current algorithm to let it work under weaker synchrony requirements.

References

[1] Aguilera M. K., Keidar I., Malkhi D., Shraer A., Dynamic atomic storage without consensus, *in Proceedings of 28th Annual ACM Symposium on Principles of Distributed Computing (PODC) 2009*.

[2] Aguilera M., Chen W.,Toueg S. Failure Detection and Consensus in the Crash-recovery Model. *Distributed Computing*, 13(2), 99-125, 2000.

[3] Aiyer A. S., Alvisi L., Bazzi R. A. Bounded Wait-Free Implementation of Optimally resilient Byzantine Storage without (Unproven) Cryptographic assumptions *in Proceedings of 21th International Symposium on Distributed Computing* (DISC), 2007.

[4] Amazon's Simple Storage Service. Available at `http://aws.amazon.com/s3`.

[5] Baldoni R., Bonomi S., Kermarrec A.M., Raynal M., Implementing a Register in a Dynamic Distributed System, *in Proceedings of the 29th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2009.

[6] Chockler G., Gilbert S., Gramoli V., Musial P. M. and Shvartsman A., Reconfigurable distributed storage for dynamic networks *Journal Parallel Distributed Computing*, 69(1), 100-116, 2009.

[7] Gilbert S., Lynch N., and Shvartsman A., RAMBO II: Rapidly Reconfigurable Atomic Memory for Dynamic Networks, *in Proceedings of International Conference on Dependable Systems and Networks (DSN)*, 2003.

[8] Maurice Herlihy Wait-free synchronization *ACM Transaction on Programming Languages and Systems* (TOPLAS) 13 (1), pp. 124-149, 1991

[9] Lamport. L., On Interprocess Communication, Part 1: Models, Part 2: Algorirhms, *Distributed Computing*, 1(2):77-101, 1986.

[10] Lynch, N. and Shvartsman A., RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks, *in Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, 2002.

[11] Malkhi D., Reiter M. K. Byzantine Quorum Systems, *Distributed Computing* 11(4), 203-213, 1998.

[12] Martin J., Alvisi L., Dahlin M.. Minimal Byzantine Storage, *in Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, 2002.

[13] Merritt M. and Taubenfeld G., Computing with Infinitely Many Processes, *in Proceedings of the 14th Int'l Symposium on Distributed Computing (DISC)*,

[14] Schneider Fred B. , Implementing Fault-Tolerant Services Using the State Machine Approach, *ACM Computing Surveys*, 22(4), 299-319, 1990

[15] Shao C., Pierce E. and Welch J.L. *Multi-writer Conditions for Shared Memory Objects* in Proc. 17th Int'l Symposium on Distributed Computing (DISC'03) LNCS #2848, pp. 106-120, 2003.

Appendix A - Correctness Proofs

Theorem 1 Termination. *If a server invokes the* join() *operation, and does not leave the system for at least* 3δ *time units, or a client invokes the* read() *operation, or invokes the* write () *operation and does not crash, then it terminates the invoked operation.*

Proof The termination of the join(), read() and write() operations follows from the fact that the wait() statement terminates. $\square_{Theorem\ 1}$

Lemma 1 $\forall t:\ |A(t)| \geq n(1-3\delta c)$.

Proof Let first consider the case $t_0 = 0$ where all the processes are active (i.e. $|A(t_0)| = n$), and let us consider the case where a churn period starts from time t_1. Due to definition of c, at time t_1, nc processes leave the system and nc processes invoke the join operation; hence, $|A(t_0+1)| = n - nc$. During the second time unit, nc new processes enter the system and replace the nc processes that left the system during this time unit. In the worst case, the nc processes that left the system are processes that were present at time t_0 (i.e., they are not processes that entered the system between t_0 and t_0+1). So, $|A(t_0+2)| \geq n - 2nc$. Note that, the cardinality of the set of active processes continues to decrease until processes that have invoked the join() at time t_1 terminate the operation and in the worst case, it happens at time $t_1 + 3\delta$. Thus, considering a churn period T_{churn} longer than 3δ time units , i.e. the longest period needed to terminate a join operation, it follows that $|A(t_0+3\delta)| \geq n - 3\delta nc = n(1-3\delta c)$. It is easy to see that the previous reasoning depends only on (1) the fact that there are n processes at each time t, and (2) the definition of the churn rate c, from which it is possible to conclude that $\forall t:\ |A(t)| \geq n(1-3\delta c)$. $\square_{Lemma\ 1}$

Lemma 2 *If* $\forall t:\ |A(t)| - (\delta nc) > (m+2)f$ *then the* select_most_recent *function returns always a value different from* null.

Proof (**Sketch**) Let us consider the worst case scenario where at time t all the m writers issue a write() operation and then crash before they terminate. Let us now consider a server s_i that invokes its join() operation at the same time t.
Joining the system, s_i executes the algorithm shown in Figure 2 and, after waiting δ time units, it sends an INQUIRY message. Any correct active servers, receiving such a message will answer by sending back its local copy of the register. Thus, at

time $t + 3\delta$ when the join() operation terminates, s_i stores in its $reply_i$ variable a set of at least $|A(t + 2\delta)| - (\delta cn)$ values sent by active processes.

Due to the property of the broadcast primitive, if a client sends a WRITE message and then crashes, such a message is not guaranteed to be delivered to all the servers and in the worst case, it is delivered only to f servers (the maximum number of replicas that can be updated and whose update cannot be useful for a reader or a joiner). Thus, s_i might receive mf replies from these servers storing the value written by the failed writes. In addition, f byzantine servers might provide wrong values.

However, considering that $|replies_i| \geq |A(t + 2\delta)| - (\delta cn) > (m + 2)f$, it means that there exist at least $f + 1$ servers not affected by the crashed writers, storing the same copies of a value and the claim follows. $\square_{Lemma\ 2}$

Corollary 1 *If* $c < \frac{n-(m+2)f}{4\delta n}$ *then the* select_most_recent *function always returns a value different from* null.

Proof It follows directly from Lemma 1 and Lemma 2 by considering that $\forall t : |A(t)| - (\delta nc) > (m + 2)f$ and knowing that in the worse case $\forall t : |A(t)| \geq n(1 - 3\delta c)$. $\square_{Corollary\ 1}$

Lemma 3 *If* $c < \frac{n-(m+2)f}{4\delta n}$*, then when a server* s_i *terminates the execution of* join()*, its local variable* $value_i$ *contains the last value written in the regular register (i.e., the last value before the* join() *invocation), or a value whose write is concurrent with the* join() *operation.*

Proof (**Sketch**) Let s_i be a process that issues a join() operation. It always executes the wait(δ) statement at line 02. Then, there are two cases according to the value of the predicate $value_i = \perp$ evaluated at line 03 of the join operation.

- $value_i \neq \perp$. it is possible to conclude that s_i has received a WRITE() message and accordingly updated $value_i$. As (1) the write operation lasts 3δ time units, (2) the join operation lasts at least 3δ time units, and (3) the message WRITE() -sent in the last part of the write - takes at most δ time units, it follows from $value_i \neq \perp$ that the join() and the write() operations overlap, i.e., they are concurrent and then can appear in the permutation π in any order, which proves the lemma for that case.

- $value_i = \perp$. In that case, s_i broadcasts an INQUIRY(i) message and waits for 2δ time units. Let t be the time at with s_i broadcasts the INQUIRY(i) message. At the end of the 2δ round trip upper bound delay, s_i updates

$value_i$ with the value returned by the select_most_recent function evaluated on the set of replies it has received. We consider two sub-cases.

- **Case 1:** No write is concurrent with the join operation. As $\forall t : |A(t)| - (\delta cn) > (m+2)f$ (Lemma 2), at least $f+1$ servers, which have the copies of the last written value, answer the inquiry of s_i and consequently, s_i sets $value_i$ to that value by 2δ time units after the broadcast, which proves the lemma.
- **Case 2:** There is (at least) one write issued by a client c_j concurrent with the join operation. In that case, s_i can receive both WRITE() messages and REPLY() messages. According to the values received at time $t+3\delta$, s_i will update $value_i$ to the value written by a concurrent update, or the value written before the concurrent writes.

$\square_{Lemma\ 3}$

Theorem 2 *Let $\mathcal{R}$ be a regular register and let $\hat{H} = (H, \prec)$ be an execution history of $\mathcal{R}$ generated by the algorithm in Figures 2 - 4. If $c < \frac{n-(m+2)f}{4\delta n}$, then any* read() *operation satisfies MWV2.*

Proof The proof follows from Lemma 3 by considering that a read() operation is a particular case of a join(). $\square_{Theorem\ 2}$