An Algorithm for implementing BFT Registers in Distributed Systems with Bounded Churn

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

July 21, 2011

MIDLAB TECHNICAL REPORT 5/11 - 2011

Abstract

Distributed storage service is one of the main abstractions provided to the developers of distributed applications due to its capability to hide the complexity generated by the messages exchanged between processes. Many protocols have been proposed to build byzantine-fault-tolerant storage services on top of a message-passing system, but they do not consider the possibility to have servers joining and leaving the computation (*churn* phenomenon). This phenomenon, if not properly mastered, can either block protocols or violate the safety of the storage. In this paper, we address the problem of building of a safe register storage resilient to byzantine failures in a distributed system affected from churn. A protocol implementing a safe register in an eventually synchronous system is proposed and some feasibility constraints on the arrival and departure of the processes are given. The protocol is proved to be correct under the assumption that the constraint on the churn is satisfied.

Keywords: *Bounded Churn, Safe Register, Byzantine Failures, Eventually Synchronous System.*

1 Introduction

Dependable storage is a pillar of many complex modern software systems (from avionics to cloud computing environments) and byzantine-fault-tolerance (BFT) is one of the main techniques employed to ensure both storage correctness and highly available accesses. Such properties have to be guaranteed despite any types of failures, including malicious ones. Availability is achieved by keeping aligned a fixed number of replicas each one hosted at a separate server.

Looking at large-scale distributed systems such as peer-to-peer systems, interconnected data centers etc., storage implementations have to withstand various types, patterns, degrees and rates of arrival to and departure of processes from the system (i.e. they have to deal with *churn*. As an example, in the context of cloud computing, a storage system is an unmanaged service (e.g. Elastic Block Store of Amazon[1]) ensuring high availability. The storage is implemented through a specific replication pattern where servers hosting replicas are selected autonomically from the server cloud. From time to time a cloud provider executes maintanance operations on the server cloud, e.g., roll-out of security patch operations, that generates a continuous and unpredictable restarts of servers that can take hours [1]. Therefore, a rollout operation translates into servers that join and leave the storage service (i.e., server churn). As a consequence, a correct and highly available storage has to be ready to autonomously tolerate servers churn as well as byzantine server behavior.
Note that, the autonomous behavior of servers, characterizing the churn action, cannot be considered as a byzantine behavior. Byzantine servers, in fact, try to make the storage service deviate from its correct behavior either maliciously or accidentally. On the contrary, server behavior in case of join and leave are well defined: processes are correct, but are temporarily unavailable; as soon as they come back to be available, they start again to work correctly. It is easy to see that if the number of servers leaving the storage service is above a given threshold, data can be lost or compromised, or storage operations cannot terminate.

In this paper, we consider a distributed system that withot churn is composed of n servers implementing a storage service, then due to the effect of churn up to J servers can be joining or leaving the service, however such number of serves is guaranteed never be below $n-J$ and eventually tends to come back to n. In this environment, we present a BFT implementation of a safe register, which is able to resist f byzantine failures and a churn of at most J servers (with $J \leq \lfloor \frac{n-5f}{3} \rfloor$). The protocol is based on quorums of size $n - f - J$, and works on top of a very general system model where churn is non-quiescent (i.e., the system model alternates infinitely often periods of no churn and periods of churn), and there is an unknown time t after which communication becomes synchronous for a period long enough to allow the BFT protocol to progress (eventually synchronous system). The algorithm presented in this paper can be also seen as an extension of quorum-based BFT algorithms [14] to ensure tolerance to servers churn.
Let us finally remark that the model of service implementation presented in this paper reflects quite well the structure of service implementaed in a cloud environment. In such environment a storage service is configured, by the cloud provider, to work in a normal working situation with a set of replicas n, defined at the beginning of the computation. However, such replicas can be affected by bounded churn due to unpredictable leaves (i.e. crash failures, maintenance operations etc.) and later on, new replicas can be set up by the provider to substitute the old ones with the aim to resume normal working situation. This is the kind of environment that this paper wants to investigate when considering the presence of byzantine proceses.

[1]http://aws.amazon.com/ebs/

The rest of the paper is contributed as follows: in Section 3, we define the system model. Section 4 provides the safe register specification while in Section 5, we detail the algorithm and the correctness proofs. Section 2 presents the related works, and finally Section 7 concludes the paper.

2 Related Work

To the best of our knowledge, this is the first work that addresses the contruction of a register resisting byzantine failures and churn in a non-synchronous system based on quorums. In the prior works, we studied the same problem from a structural point of view [7] and in an environment with crash failures [6].

Byzantine fault tolerant systems based on quorums. Traditional solutions to build byzantine storage can be divided into two categories: replicated state machines [17] and byzantine quorum systems [8], [14], [15]. Replicated state machines uses $2f+1$ server replicas and require that every non-faulty replica agrees to process requests in the same order [17]. Quorum systems, introduced by Malkhi-Reiter in [14], do not rely on any form of agreement they need just a sub-sets 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. [4] proposes a protocol for implementing a single-writer and multiple-reader atomic register that holds wait-freedom property with using just $3f+1$ servers. This is achieved at the cost of longer (two phases) read and write operations. In this paper, our objective has been to design an algorithm that follows the Malkhi-Reiter's approach (i.e., single-phase operations), and that is able to tolerate both f failures and concurrent running join of at most J servers at any time, using less than $5(f+J)$ servers. This number of replicas would have been indeed necessary if we consider churning servers as byzantine processes. Leveraging from the difference between the behavior of a byzantine server and a churning one, the algorithm presented in this paper needs just $5f+3J$ server replicas.

Registers under quiescent churn. In [13], [10] and [9], a Reconfigurable Atomic Memory for Basic Object (RAMBO) is presented. RAMBO works on the 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 large changes in the set of participant 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.

In [2] 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 all the changes (i.e. join and failure of processes) suggested by the participants and the quorums are represented by any majority of processes. To ensure 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.

Relationship between the churn model and the crash-recovery one In crash-recovery model processes may recover after a crash and each process is usually augmented with stable storage and, as in the crash failure model, the set of processes that will be part of the system is known in advance [3]. At a first glance, our churn model could resemble crash-recovery one (i.e., a server

that leaves and re-joins the regular register computation could be seen as a crash and a recovery of a process), they differ in several fundamental aspects. In the model presented in this paper: (1) there is no assumption of initial knowledge about the set of processes, which will be part of the computation, (2) processes may join the application at any time, (3) processes may crash and later restart with another identifier an infinite number of times without relying on stable storage, which is an extremely important point when considering servers are virtual machines that can migrate from one physical machine to another one, and then the stable storage of the former machine could not be available anymore. Therefore the model presented in this paper is more general than crash recovery one. Let us finally remark that we are not aware of any BFT protocol working in a crash-recovery environment.

3 System Model

The distributed system is composed of a *universe of clients* U_c (i.e. the clients system) and of a disjoint *universe of servers* U_s (i.e. the servers system). The clients system is composed of a finite arbitrary number of processes (i.e. $U_c = \{c_1, c_2, \dots c_m\}$) while the servers system is dynamic, i.e. processes may join and leave the system at their will. A server enters the servers system by executing the connect() procedure. Such an operation aims at connecting the new process to both clients and servers that 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 to execute algorithms. In order to model processes continuously arriving to and departing from the servers system, we assume the infinite arrival model (as defined in [16]). The set of processes that can participate in the servers system (also called *server-system population*) is composed of a potentially infinite set of processes $U_s = \{\dots s_i, s_j, s_k \dots\}$, each one having a unique identifier (i.e. its index). However, the servers system is composed, at each time, of a finite subset of the server-system population. Initially, every server $s_i \in U_s$ is in the *down* state as soon as s_i invokes the connect() operation, it changes its state from *down* to *up*. When the server s_i disconnects itself from the servers system, it changes again its state coming back to *down*.

Clients and servers can communicate only by exchanging messages through reliable and authenticated FIFO channels. As we did in [5], in the following, we assume the existence of a protocol managing the arrival and the departure of servers from the distributed system, such a protocol is also responsible for the connectivity maintenance among the processes belonging to the distributed system. As in [14], [15], we assume that clients are correct and servers can suffer arbitrary failures. To simplify the presentation, let us assume the existence of a global fictional clock not accessible from processes.

Distributed Computation. Several distributed computations run on top of the distributed system, involve the participation of a subset of the servers set of the servers system. To simplify the presentation, let us assume that there exists only one distributed computation run in our system. We identify as $C_s(t)$ the subset of processes belonging to the servers system U_s that are participating in the distributed computation at time t (i.e. the *server-computation set*). At time t_0, when the server-computation set is set up, n servers belong to the servers computation (i.e. $|C_s(t_0)| = n$). A server s_i, belonging to the servers system that wants to join the distributed computation has to execute the join_Server() operation. Such an operation invoked at some time t

is not instantaneous and takes time to be executed; how much this time is, depends on the specific implementation provided for the join_Server() 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, which are participating in the computation, and it changes its state from *up* to *joining*.
When a server s_j participating in the distributed computation wishes to leave the computation, it stops to execute the server protocols (i.e. the leave_Server 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_Server() operation with a new identity.

It is important to notice that (i) there may exist processes belonging to the servers system that never join the distributed computation (i.e. they execute the connect() procedure, but they never invoke the join_Server() operation) and (ii) there may exist processes, which even after leaving the servers computation, still remain inside the servers system (i.e. they are correct, but they stop to process messages related to the computation). To this aim, it is important to identify the subset of processes that are actively participating in the distributed computation and the ones that are joining.

Definition 1 (Joining Servers Set) *A server is* joining *from the time it invokes the* join_Server() *operation until the time it terminates such operation.* $J(t)$ *denotes the set of servers that are execution the* join_Server() *operation at time* t*.*

In the following, we refer as J the maximum value of $J(t)$ for any t.

Definition 2 (Active Servers Set) *A server is* active *in the distributed computation from the time it returns from the* join_Server() *operation until the time it leaves.* $A(t)$ *denotes the set of servers that are active at time* t*, while* $A([t,t'])$ *denotes the set of servers that are active during the whole interval* $[t,t']$ *(i.e.* $s_i \in A([t,t'])$ *iff* $s_i \in A(\tau)$ *for each* $\tau \in [t,t']$*).*

A server s_i changes its state from *joining* into *active* as soon as it gets the join_Confirmation event, and remains in such a state until it decides to leave the server-computation set (thus, coming back to the *up* state).
Note that, at each time t the set of servers participation in the distributed computation is partitioned into active processes and joining processes. i.e.

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

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 whole computation[2]. It is important to note that servers know the values f and J, but they are not able to know the subset of C_s representing the faulty processes. In Figure 1 it is shown the state-transition diagram of a correct server.

[2]Note that, f is an upper bound on the number of faulty processes. As a consequence, during the computation, there may exists periods where less than f byzantine servers participate in the computation. Moreover, our assumption does not implies that the set of faulty servers is static but we admit it can change during the whole computation.

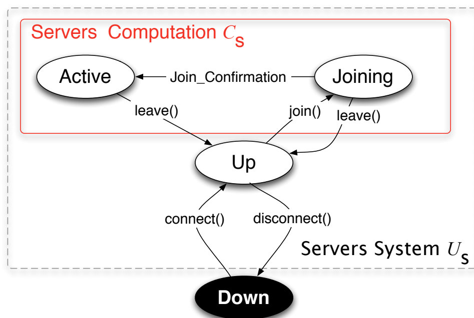


Figure 1: State-transition diagram of a Correct Server

Non-Quiescent Bounded Churn. The servers computation alternates periods of churn and periods of stability. More specifically, there exist some periods T_{churn} in which servers join and leave the computation, 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 at time t_0 all the servers participating in the server computation are active (i.e. $|A(t_0)| = n$). Moreover, we assume that the churn affecting the servers computation is bounded by an integer value $J \geq 0$ and the number of servers participating in the servers computation can change in the interval $[(n - J), n]$ (i.e. $\forall t, |C_s(t)| \in [(n - J), n]$). Finally, we assume that in the distributed computation there are always at least $n - J$ active servers (i.e.$\forall t, |A(t)| \geq n - J$). The above equality implies that the servers computation is configured to work with n servers event though it tolerates that up to J servers can leave and later on they can be replaced by up to J new joining servers. Thus the value J represents the upper bound on the churn.

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 non-quiescent 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.

4 Register Specification

A register is a shared variable accessed by a set of processes, i.e. clients, through two operations, namely read() and write(). Informally, the write() operation updates the value stored in the shared variable while the read() obtains the value contained in the variable (i.e. the last written value). Every operation issued on a register is, generally, not instantaneous and it 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 respectively) according to the fictional global time.

An operation op is *complete* if both the invocation event and the reply event occur (i.e. the process executing the operation does not crash between the invocation and the reply).

Given two operations op and op', their 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 op and op' are *concurrent* ($op||op'$). Given a write(v) operation, the value v is said to be written when the operation is complete. In case of concurrency while accessing the shared variable, the meaning of *last written value* becomes ambiguous. In this paper, we will consider a single-writer/multiple-reader safe register which is specified as follows [12][3]:

- Termination: If a correct process (either a client or a server) participating in the computation invokes an operation and does not leave the system, it eventually returns from that operation.
- Validity: a read() not concurrent with any write() returns the last written value before its invocation. In the case of concurrency, a read() may return any value.

As a specialization of the generic model of the computation presented in the previous Section, we consider in this paper a safe register computation, i.e. the join_Server() operation, executed by servers, has the aim to provide new servers with the state of the register. Concerning the departures from the computation, we consider the leave operation as an implicit operation; when a server s_i leaves the computation, it just stops to send and process messages related to the register computation. To simplify the notation, whenever not strictly necessary, we use the term join() instead of join_Server().

5 Safe Register Implementation

A register is maintained by the set of active servers. No agreement abstraction is assumed to be available at a server. Clients do not maintain any register information; they can just trigger operations and interact with servers through message exchanges. Moreover, we assume that each server has the same role in the distributed computation (i.e. no server acts as a coordinator) and when it issues a join() operation at some time t, the server does not leave the computation before time $t + 3\delta$.

Eventually synchronous communication model. Due to the impossibility of implementing a register in a fully asynchronous system prone to non-quiescent churn [5], in this paper we will assume a partial synchronous system, i.e. there exists a time t after which a synchrony period holds long enough to ensure the correct progress of protocol implementation. In particular, eventual synchrony implies that each message sent at some time t' after t, by a process p, is delivered within δ time units by every process belonging to the distributed system in the interval $[t', t' + \delta]$.

Quorums. The basic idea of the algorithm is to extend the opaque masking quorums mechanism, defined by Malkhi and Reiter [14], to implement a safe register in a dynamic distributed system with byzantine failures. In particular, both join(), read() and write() operations are executed on quorums of servers participating in the distributed computation of size $n - f - J$.

[3]Interestingly, safe registers have the same computational power as regular registers and atomic registers. This means that it is possible to implement a multi-writer/multi-reader atomic register from single-writer/single-reader safe registers as shown in [11].

5.1 A protocol for eventually synchronous dynamic system

Each reader client c_i maintains the following variables:

- one integer variable, denoted $read_sn_i$, representing the sequence number to associate to each read() operation. Initially the variable is set to 0.
- a set variable, denoted as $cl_replies_i$, used to collect answers sent by servers and initially empty.

Moreover, the writer client c_w also maintains:

- two integer variables sn_w and $count_i$, representing respectively the sequence number to associate to each write() operation and the number of tentative write() operations. Initially both the variables are set to 0.
- an array of sets variable, denoted as $write_ack_i[]$, used to collect the servers that have acknowledged its last write.
- an array of sets variable, denoted as $confirmation_i[]$, used to collect the servers that have confirmed its last write.

Each server s_i has the following local variables.

- A set W_i that stores the writers identifiers.
- Two variables denoted $register_i$ and sn_i; $register_i$ contains the local copy of the safe register, while sn_i is the associated sequence number.
- A boolean $active_i$, initialized to *false*, that is switched to *true* just after s_i has joined the system.
- Two set variables, denoted $replies_i$ and $reply_to_i$, that are used in the period during which s_i is joining the system. The local variable $replies_i$ contains the 4-tuple $< id, value, sn, r_sn >$ that s_i has received from other servers during its join period, while $reply_to_i$ contains the IDs of servers, which are joining the system concurrently with s_i (as far as s_i knows).
- dl_prev_i is a set where (while it is joining the system) p_i records the processes that have acknowledged its inquiry message, while they were not yet active (so, these processes were joining the system too). When it terminates its join operation, p_i has to send them a reply to prevent them to be blocked forever.

In order to simplify the pseudo-code notation, let us consider the function most_frequent($replies$). Such a function is used by both clients and servers to select the most frequent pair $< val, sn >$ occurred in the set $replies_i$. In the case that more than one pair with the same frequency exist, the function returns the pair having the highest sn.

```
operation join(i):
(01)  register_i ← ⊥; sn_i ← −1; active_i ← false; replies_i ← ∅;
(02)  reply_to_i ← ∅; dl_prev_i ← ∅; read_sn_i ← 0;
(03)  broadcast INQUIRY(i, 0);
(04)   wait until(|replies_i| ≥ (n − f − J));
(05)  let < val, sn >← most_frequent(replies_i);
(06)  if (sn > sn_i) then sn_i ← sn; register_i ← val end if
(07)  active_i ← true;
(08)  for each < j, r_sn >∈ reply_to_i ∪ dl_prev_i do
(09)               do send REPLY (< i, register_i, sn_i >, r_sn) to p_j
(10)  end for;
(11)  return(ok).
-----------------------------------------------------------------------
(12)  when INQUIRY(j, r_sn) is delivered:
(13)          if (active_i) then send REPLY (< i, register_i, sn_i >, r_sn) to p_j
(14)                        else  reply_to_i ← reply_to_i ∪ {< j, r_sn >};
(15)                               send DL_PREV (i, r_sn) to p_j
(16)          end if.

(17)  when REPLY(< j, value, sn >, r_sn) is received:
(18)          if (read_sn_i = r_sn) then
(19)             if (∃ < j, −, −, r_sn >∈ _replies_i) then
(20)                replies_i ← replies_i/{< j, −, −, r_sn >};
(21)             endif
(22)             replies_i ← replies_i ∪ {< j, val, sn, r_sn >};
(23)          endif

(24)  when DL_PREV(j, r_sn) is received: dl_prev_i ← dl_prev_i  ∪ {< j, r_sn >}.
```

Figure 2: The join() protocol for an eventually synchronous system (code for s_i)

The join() operation (Figure 2). The server s_i broadcasts an INQUIRY ()) message to inform the other servers, which it is entering the distributed computation set, and wants to obtain the value of the safe register (line 03).

Then, after it has received "enough" replies (line 04), s_i selects among the set of received values, the one occurred with the highest frequency (line 05). Moreover, s_i updates its local copy of the register (line 06), it becomes active (line 07), and sends a reply to the processes in the set $reply_to_i$ (line 08-10). It also sends such a reply message to the servers in its dl_prev_i set, in order to prevent them from waiting forever. In addition to the term $< i, register_i, sn_i >$, a reply message sent to a server s_j, from a server s_i, carries also the read sequence number r_sn that identifies the corresponding request issued by s_j.

When s_i delivers an INQUIRY(j, r_sn), it always sends back a message to p_j. It sends a REPLY() message if it is active (line 13), and a DL_PREV() if it not active yet (line 15). Moreover, in case s_i is not active, it stores the inquiry received from s_j in the $reply_to_j$ variable, to remember to answer later, as soon as it becomes active (line 14).

When s_i receives a REPLY$(< j, value, sn >, r_sn)$ message from a server s_j, if the reply message is the first answer to its INQUIRY$(i, read_sn)$ message s_i adds $< j, value, sn, 0 >$ to the set of replies that it has received so (line 22). On the contrary, s_i updates the information already received from s_j with the new value (line 20 - 22).

Finally, when s_i receives a message DL_PREV(j, r_sn), it adds its content to the set dl_prev_i (line 24), in order to remember that it has to send a reply to s_j when it becomes active (lines 08-10).

```
operation read(i):
(01)  read_sn_i ← read_sn_i + 1;
(02)  cl_replies_i ← ∅;
(03)  repeat
(04)      broadcast READ(i, read_sn_i);
(05)  until ((|cl_replies_i| ≥ n − f − J)
(06)  let < val, sn >← most_frequent(cl_replies_i);
(07)  if (sn > sn_i)
(08)      then sn_i ← sn;
(09)           value_i ← val
(10)  end if;
(11)  return(val).
-----------------------------------------------------------
when REPLY(< j, val, sn >, r_sn) is delivered:
(12)  if (read_sn_i = r_sn) then
(13)     if (∃ < j, −, −, r_sn >∈ cl_replies_i) then
(14)        cl_replies_i ← cl_replies_i/{< j, −, −, r_sn >};
(15)     endif
(16)     cl_replies_i ← cl_replies_i ∪ {< j, val, sn, r_sn >};
(17)  endif
```

(a) Client Protocol

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

(b) Server Protocol

Figure 3: The read() protocol for an eventually synchronous system

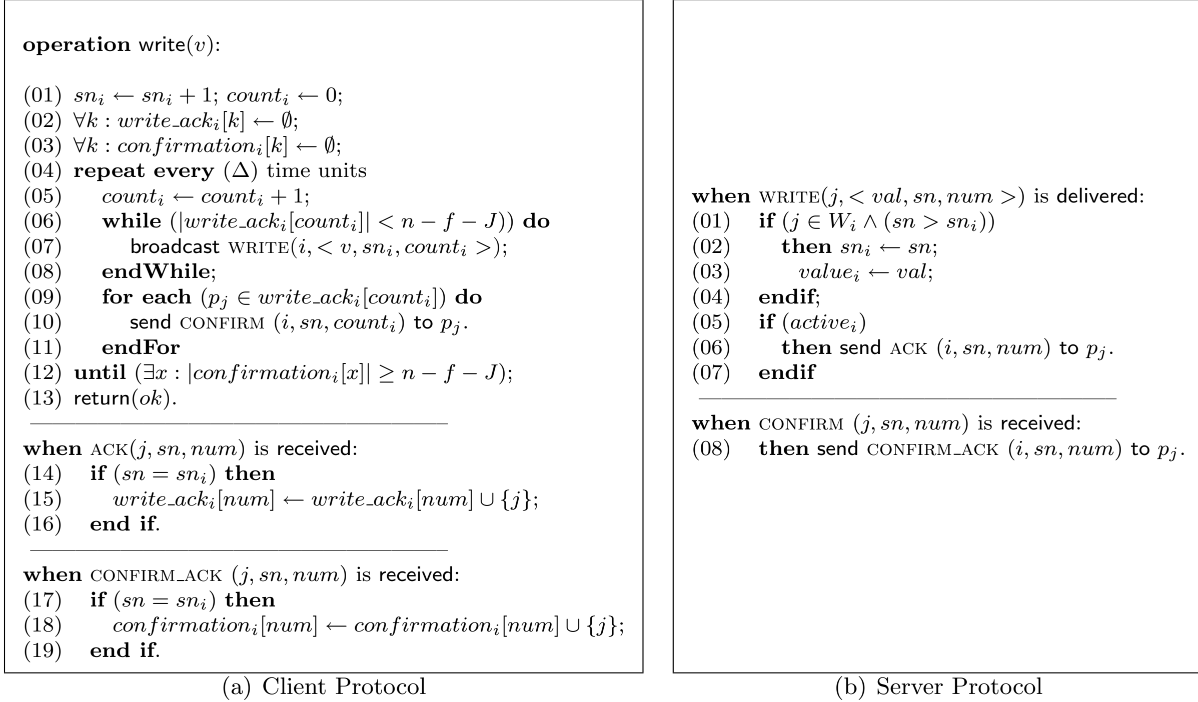
The read() operation (Figure 3). The algorithm for the read() operation is a simplified version of the join() algorithm. The main difference between the two algorithms is the "stubborn" retransmission mechanism used by the read() (lines 03-05). This mechanism is necessary because (i) a read() broadcast message could not be received both by a leaving server and by a joining one and (ii) a reply message sent by a leaving server could not reach the client. This might block the client read protocol that could not reach the expected number of replies $(n - f - J)$ [4]. Resending the read message periodically will ensure that the message eventually reaches enough servers due to the arrival of either a stability period or a synchrony period.

Note that, the same problem does no happen during the join() execution thanks to the DL_PREV mechanism. When a servers s_i joins, in fact, its inquiry is delivered to all the servers belonging to the distributed computation and if new processes arrive, they become aware about the join of s_i due to the DL_PREV message.

The write() operation (Figure 4). Similarly to the read() operation, also the write() is implemented by repeating the value dissemination until the writer gets acknowledgements from a quorum of $n - f - J$ processes (lines 06 - 08). In addition, in order to terminate the write(), the client

[4] Let us recall that the communication primitives work in a best-effort fashion on top of FIFO channels. Thus, there is no guarantee that a message m sent at some time t by a process p is delivered to other processes in case p leaves the computation.

must also need to receive a confirmation that the acknowledgement are effectively sent by a quorum of processes that belongs to the distributed computation for a sufficient long time to disseminate correctly the new value (lines 09 - 12)



(a) Client Protocol (b) Server Protocol

Figure 4: write() protocol for an eventually synchronous system

When a message WRITE($j, < val, sn, num >$) is delivered to a server s_i, it takes into account the pair (val, sn) if it is more up-to-date than its current pair and only if the message came from an authenticated channel of one writer (line 01). Then, if s_i is active, it sends back an ACK (i, sn) message to the writer (line 06).

When the client receives an ACK (j, sn) message from the server s_j, it adds s_j to its set $write_ack_i$ if this message is an answer to its last write operation(line 09).

Finally, when the client receives a CONFIRM_ACK (j, sn, num), it just takes into account the confirmation received by server s_j.

Correctness Proofs.

Definition 3 *A quorum system $\mathcal{Q} \subseteq 2^{|C_s|}$ is a non-empty set of subsets of C_s, every pair of which intersect. Each $Q \in \mathcal{Q}$ is called a* quorum.

Definition 4 (Opaque Masking Quorum) *Let $B \subset C_s$ be the subset of faulty processes participating in the distributed computation. A quorum system $\mathcal{Q}$ is an opaque masking quorum system if:*

- **(P1)** $\forall\ Q_w, Q_r \in \mathcal{Q} : |(Q_w \cap Q_r)/B| \geq |(Q_r \cap B) \cup (Q_r/Q_w)|$
- **(P2)** $\forall\ Q_w, Q_r \in \mathcal{Q} : |(Q_w \cap Q_r)/B| > |(Q_r \cap B)|$

- **(P3)** $\exists\, Q \in \mathcal{Q} : Q \cap B = \emptyset$.

Lemma 1 *Let n be the number of processes participating in the distributed computation at any time t and let f be the maximum number of byzantine processes participating in the computation. If $n \geq 5f + 3J$ then $\mathcal{Q} = \{|Q_i| = n - f - J\}$ is an opaque masking quorum for the safe register computation.*

Proof Note that, considering a quorum Q_i composed of $n-f-J$ processes, property $P3$ is always guaranteed. Moreover, $P1$ implies $P2$ and thus in the following we will show only $P1$.

Let Q_w and Q_r be respectively two quorums associated to a $\mathsf{write}(v)$ operation *op* and a $\mathsf{read}()/\mathsf{join}()$ operation *op'*.

Let $X = (Q_r/Q_w)$ be the set of processes belonging to the computation and not affected from *op*; the number of this processes is $|X| = n - |\mathcal{Q}_w|$.
The quorum Q_r can be represented as $\mathcal{Q}_r = X \cup (\mathcal{Q}_r \cap \mathcal{Q}_w)$ and considering that X and $\mathcal{Q}_r \cap \mathcal{Q}_w$ are disjoint sets, we can deduce the following: $|\mathcal{Q}_r| = |X| + |\mathcal{Q}_r \cap \mathcal{Q}_w| \Rightarrow |\mathcal{Q}_r \cap \mathcal{Q}_w| = |\mathcal{Q}_r| - |X| = |\mathcal{Q}_r| - n + |\mathcal{Q}_w|$.
Considering that $|\mathcal{Q}_r| = |\mathcal{Q}_r| = n - f - J$, we get $|\mathcal{Q}_r \cap \mathcal{Q}_w| = ((n - f - J) - n + (n - f - J)) = n - 2f - 2J$.

Note that, in the worst case, $(Q_r \cap B) = B$ and it is a disjoint set from (Q_r/Q_w).

As a consequence, $|(Q_r \cap B) \cup (Q_r/Q_w)| = f + n - (n - f - J) = 2f + J$.

Therefore, $|(Q_w \cap Q_r)/B| \geq |(Q_r \cap B) \cup (Q_r/Q_w)| \Rightarrow$
$(n - 2f - 2J) - f \geq 2f + J \Rightarrow n \geq 5f + 3J$.

$\Box_{Lemma\ 1}$

Theorem 1 Safety. *Let us assume that $n \geq 5f + 3J$. Given the algorithm in Figures 2 - 4, then a* $\mathsf{read}()$ *operation that is not concurrent with any* $\mathsf{write}()$*, returns the last value written before the* $\mathsf{read}()$ *invocation.*

Proof (Sketch) Let $\mathsf{write}_\alpha(v)$ be the α-th write operation invoked on the register, and $W_\alpha(t)$ the set of processes that, at time t, have the corresponding value v in their local copy of the safe register (to simplify the reasoning, and without loss of generality, we assume that no two write operations write the same value).

Let t_0 be the starting time of the computation. From the initialization statement, it follows that n servers initially defining the system are active and store the initial value of the safe register (say v_0). Consequently, we have $|A(t_0)| = |W_0(t_0)| = n > n - f - J$. Let $t_y = t_0 + y$ (the time instant that is y time units after t_0). Let us consider the worst case scenario where from time t_1 a churn periods starts. At time t_1, c_1 servers leave the system and c_1 servers invoke the $\mathsf{join}()$ operation. All the servers that leave were active at time t_0 and their local copy of the register contained v_0. Since $n \geq 5f + 3J$, it follows that $J \leq \lfloor \frac{n-5f}{3} \rfloor$. As a consequence, in the worst case, $c_1 = J = \lfloor \frac{n-5f}{3} \rfloor$ and $|A(t_1)| \geq n - J$. If follows, at most one correct server in $A(t_1)$ (and then also in $W_0(t_1)$) can leave before any server entering the computation terminates its join operation.

Let s_i be the first servers that terminates its $\mathsf{join}()$ operation. Then two cases can happen: (i) no $\mathsf{write}()$ operation is concurrent with the join of s_i or (ii) there is at least a $\mathsf{write}()$ operation concurrent with the join of s_i.
Case 1: no concurrency with $\mathsf{write}()$ operations. If no $\mathsf{write}()$ operation is concurrent with the join of s_i, then all of the replies received by s_i, except at most f, come from correct servers in

$W_0(t_1)$. Each of these servers (in the worse case $n - J - 2f$) stores the last value written (namely, the initial value v_0) in its local copy of the register together with the sequence number 0. Thus, when s_i executes the lines 05-06 of the join() operation (Figure 2), it updates its local variable with the value v_0 (i.e. the last value written).

Case 2: There exists at least one write() operation concurrent with the join. In this case, the join() operation is allowed to return any value.

The same reasoning can be applied for the subsequent write operations and we have that at the end of the join operation, each servers got a valid value. Moreover, considering that a read() operation is just a simplified version of the join, the same reasoning can be applied and the claim follows. $\square_{Theorem\ 1}$

Lemma 2 *Let us assume that (1) $n \geq 5f + 3J$, and (2) a server that invokes the* join() *operation remains in the system for at least 4δ time units. If a server process s_i invokes the* join() *operation, and does not leave the computation, this join operation terminates.*

Proof Let us observe that, in order to terminate its join() operation, a server process s_i has to wait until its set $replies_i$ contains $n - f - J$ elements (line 04, Figure 2). Empty at the beginning of the join operation (line 01, Figure 2), this set is filled in by s_i when it receives the corresponding REPLY() messages (line 22 of Figure 2).

A server s_j sends a REPLY() message to s_i if (i) either it is active and has received an INQUIRY message from s_i, (line 13, Figure 2), or (ii) it terminates its join() operation and $< i, - > \in reply_to_j \cup dl_prev_j$ (lines 08-10, Figure 2).

Let us suppose by contradiction that $|replies_i|$ remains smaller than $n - f - J$ and let us consider the worst case scenario where there is no stability periods before the synchrony assumptions hold. This means that s_i does not receive enough REPLY() carrying the appropriate sequence number. Let t be the time at which the system becomes synchronous and let us consider a time $t' > t$ at which a new server process s_j invokes the join operation. At time t', s_j broadcasts an INQUIRY message (line 03, Figure 2). As the system is synchronous from time t, every process present in the system during $[t', t' + \delta]$ receives such INQUIRY message by time $t' + \delta$. As it is not active yet, when it receives s_j's INQUIRY message, the process s_i executes line 14 of Figure 2 and sends back a DL_PREV message to s_j.

Due to the assumption that every process that joins the system remains inside for at least 3δ time units, s_j receives s_i's DL_PREV and executes consequently line 24 (Figure 2) adding $< i, - >$ to dl_prev_j. Due to the assumption that there are always less equal than $\lfloor \frac{n-5f}{3} \rfloor$ joining servers in the computation, we have that at time $t' + \delta$ at least $n - (\lfloor \frac{n-5f}{3} \rfloor) = n - J$ processes receive the INQUIRY message of s_j. Note that, each active and correct server will execute line 13 (Figure 2) and sends a REPLY message to s_j. As a consequence, at least $n - f - J$ servers will answer to s_j's inquiry. Due to the synchrony of the system, s_j receives these messages by time $t' + 2\delta$ and then stops waiting and becomes active (line 07, Figure 2). Consequently (lines 08-10) s_j sends a REPLY to s_i as $i \in reply_to_j \cup dl_prev_j$. In δ time units, s_i receives that REPLY message and executes line 22, Figure 2.

Note that, due to the broadcast property, each process, participating in the computation (either it is active or joining) at time $t_B(join_i)$ when s_i issued the join operation, is guaranteed to receive eventually the message if it remains in the computation. Moreover, due to dl_prev messages, all the processes joining after s_i will be notified about the joining state of s_i (i.e. they will receive a

dl_prev messages from a server whose state is joining).
Considering that (i) active processes participating in the computation at time $t_B(join_i)$ can be replaced along time by processes joining after $t_B(join_i)$, (ii) all these servers are aware of s_i and (iii) there always exist at least $n - J$ active processes participating in the computation and that at most f of them can be faulty, then there always exist enough processes along time able to reply to the inquiry of s_i. Thus, s_i will eventually receive a reply from any of them so it will fill in its set $replies_i$, terminating its join operation. $\Box_{Lemma\ 2}$

Lemma 3 *Let us assume that (1) $n \geq 5f + 3J$, and (2) a server that invokes the* join() *operation remains in the system for at least 4δ time units. If a client c_i invokes a* read() *operation and does not leave the system, this read operation terminates.*

Proof The proof of the read termination is the same as that of Lemma 2. The read operation, in fact, is a simplified case of the join algorithm where the operation is initiated from a client and the chain of messages INQUIRY(), DL_PREV(), REPLY() is replaced by a READ() message retransmission mechanism.

$\Box_{Lemma\ 3}$

Lemma 4 *Let us assume that (1) $n \geq 5f + 3J$, and (2) a server that invokes the* join() *operation remains in the system for at least 4δ time units. If a client process c_i invokes* write() *and does not leave, this write operation terminates.*

Proof (**Sketch**) Before terminating the write of a value v with a sequence number sn a client process c_i has to wait until there exists at least one entry of the array $confiramation_i$ containing at least $n - f - J$ elements.
Empty at the beginning of the write operation, this set is filled in when the CONFIRM_ACK$(-, sn)$) messages are delivered to c_i. Such a message is sent by every active server process s_j such that s_j receives the corresponding CONFIRM message from c_i.

Suppose by contradiction that c_i never fills in $confiramation_i$. This means that there note exist any count such that all the servers that have acknowledge the write have also sent the confirm message to c_i.
Let us consider the time t at which the system becomes synchronous, i.e., every message sent by any process p_j at time $t' > t$ is delivered by time $t' + \delta$ (either if p_j is a client or a server).

Since, after t the system is synchronous and considering that the client c_i continuously retransmits the WRITE() message, then any active server s_j receives such a message in at most δ time units. As a consequence, at least $n - f - J$ servers will execute line 06 of Figure 2 sending back an ACK$(-, snb)$ message to c_i. Such a messages will be delivered by c_i latest at time $t + 2\delta$ and thus c_i will exit from the loop in lines 06-08. At the same time, c_i will also send a CONFIRM message to all the servers from which it has received the ack and suck messages will be delivered by time $t + 3\delta$ triggering the send of the corresponding CONFIRM_ACK.

As (1) by assumption a process that joins the system does not leave for at least 4δ time units and (2) the system is now synchronous, the chain of messages WRITE, ACK, CONFIRM, CONFIRM_ACK will lead c_i to execute line 18 and adds s_j to the set $confirmation_i$. $\Box_{Lemma\ 4}$

From Lemma 2, Lemma 3 and Lemma 4 we have:

Theorem 2 Termination. *Let us assume that $n \geq 5f + 3J$. Given the algorithm in Figures 2 - 4, if a process invokes* join(), read() *or* write (), *and does not leave the system, it terminates its operation.*

operation read(i):
(01) $read_sn_i \leftarrow read_sn_i + 1$;
(02) $cl_replies_i \leftarrow \emptyset$;
(03) **repeat**
(04) broadcast READ$(i, read_sn_i)$;
(05) **until** $\big((|cl_replies_i| \geq 2f + 1)$
(06) **let** $< val, sn > \leftarrow$ most_frequent$(cl_replies_i)$;
(07) return(val).

---

when CL_REPLY$(< j, val, sn >, r_sn)$ is delivered:
(08) **if** $(read_sn_i = r_sn)$ **then**
(09) **if** $(\exists < j, -, -, r_sn > \in cl_replies_i)$ **then**
(10) $cl_replies_i \leftarrow cl_replies_i / \{< j, -, -, r_sn >\}$;
(11) **endif**
(12) $cl_replies_i \leftarrow cl_replies_i \cup \{< j, val, sn, r_sn >\}$;
(13) **endif**

(a) Client Protocol

when READ(j, r_sn) is delivered:
(01) **if** $(\neg reading[j])$
(02) **then** $reading[j] \leftarrow$ true;
(03) $read_replies_i[j] \leftarrow \emptyset$;
(04) **else if** $(last_read[j] < r_sn)$
(05) **then** $read_replies_i[j] \leftarrow \emptyset$;
(06) **end if**.
(07) **if** $(active_i)$
(08) **then if** $(last_read[j] \leq r_sn)$
(09) **then** $last_read[j] \leftarrow r_sn$:
(10) broadcast REPLY $(< i, value_i, sn_i >, r_sn)$;
(11) $read_replies_i[j] \leftarrow read_replies_i[j] \cup value_i$
(12) **end if**.
(13) **else** $reply_to_i \leftarrow reply_to_i \cup \{< j, r_sn >\}$;
(14) **end if**.

---

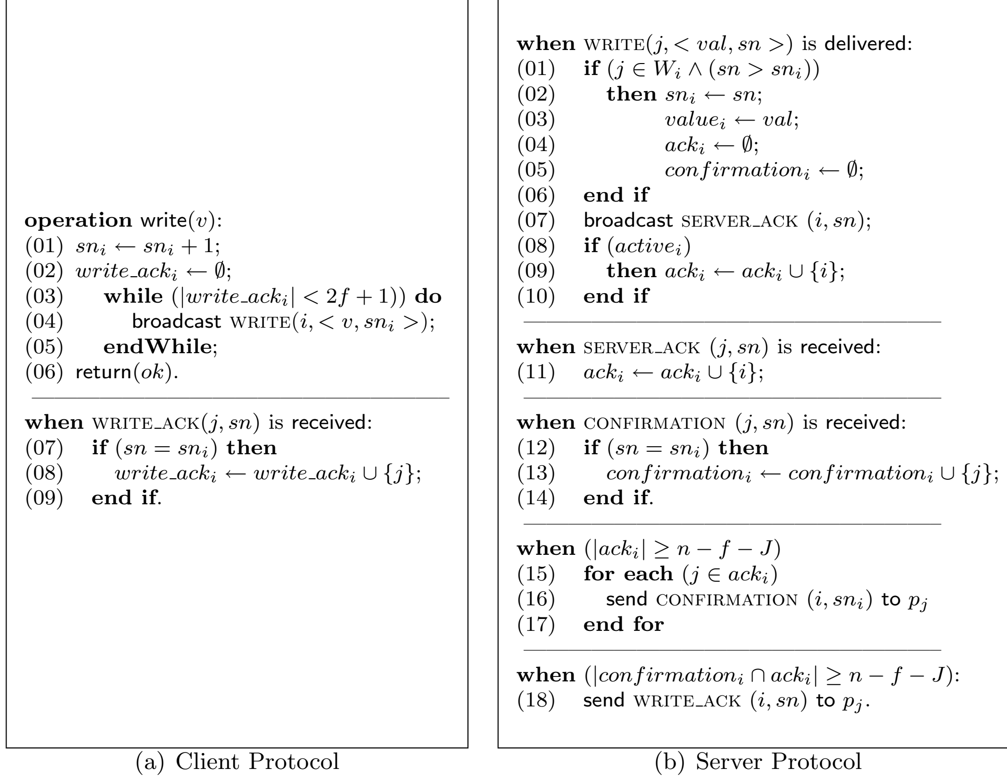
when REPLY$(< j, val, sn >, r_sn)$ is delivered:
(15) **if** $(last_read[j] < r_sn)$
(16) **then** $read_replies_i[j] \leftarrow read_replies_i[j] \cup value_i$

---

when $(\exists\ _j : |read_replies_i[j]| \geq n - f - J)$:
(17) **let** $< val, sn > \leftarrow$ most_frequent$(read_replies_i[j])$;
(18) send CL_REPLY $(< i, val, sn >, last_read[j])$ to c_j;
(19) $reading[j] \leftarrow$ false;

(b) Server Protocol

Figure 5: The read() protocol for an eventually synchronous system



(a) Client Protocol (b) Server Protocol

Figure 6: write() protocol for an eventually synchronous system

6 Weakening Clients Knowledge Assumptions

The protocol proposed in Figures 2-4 assumes that all the processes (both clients and servers) participating in the distributed computation know (i) the maximum number of byzantine servers f, (ii) the number n of servers participating in the distributed computation and (iii) the maximum number of non-active servers J.
However, the latter two points can be weakened by assuming that only servers to know such a values. These values are, in fact, two configuration parameters of the servers computation, defining the robustness of the service. As a consequence, assuming that only servers know such values is not a strong assumption.

In this section, we will show how it is possible to modify the read() and the write() implementations to let servers dynamic behavior be transparent to clients. The basic idea of the algorithms proposed in this section is to let servers act on behalf of the clients. In particular, before answering to the client, each servers must be sure that the current operation has been acknowledged/executed by a quorum of servers. As a consequence, the client needs only to know the maximum number of byzantine replicas f and it waits until it receives at least $2f + 1$ answers. Note that, this modification does not impact the relation about n, f and J, even though it increased the algorithms complexity in terms of messages and latency.

The read() operation (Figure 5). In addition to the data structure used in the previous read() algorithm, each server s_i needs also to maintain the following local variables:

- an array of boolean $reading_i[]$, initialized to false, where the j-th entry is set to true when s_i receives a new read request from client c_j.
- an array of integer $last_read_i[]$ where the j-th entry corresponds to the last read request issued by client c_j.
- a set variable $read_replies_i$ where s_i stores values forwarded by other servers.

The client behaves like in the previous algorithm but it waits only $2f+1$ answers. Servers works as follows: when a server s_i receives a READ() message from a client c_j, it starts to read the value of the variable from the other active servers. In particular, it first checks if the message corresponds to a new read request and in case it resets its local variables (lines 01-06). Then it forwards its local value trough a REPLY() message (line 10) and consider its contribution (line 11). When a server s_i receives a REPLY() message from a server s_j, it checks if the message is related to the current read it is considering and if it so, it consider the contribution sent by s_j.

As soon as a servers has received at least $n-f-J$ values from other servers, it select the most frequent one and answers to the client.

The write() operation (Figure 6). As for the read() operation, we have modified the write algorithm by letting first the servers execute the operation and only after they got acknowledgement from a quorum of $n-f-J$ other servers, they send back an ack to the client.

7 Conclusion

In this paper, we have provided an implementation of a distributed storage in the presence of both servers churn and byzantine servers. In a computation composed of a constant number of n servers, the protocol is able to tolerate at most J joining servers if $J \leq \lfloor \frac{n-5f}{3} \rfloor$, where f is the maximum number of byzantine servers. The protocol works in an eventually synchronous environment, so it keeps the safety during arbitrarily long (but finite) periods of asynchrony and churn, while it is able to quickly terminate as soon as the system gets into synchrony bounds.

We decided to extend Malki-Reiter's protocol for its simplicity and because operations are short in time. Other algorithms (e.g. [15]) reduce indeed the number of servers needed for handling f byzantine failures, however, this is done at the cost of multistep read and write operations. When facing a dynamic system, such length matter as the leaving of processes during read and write operations can impact their safety and liveness. We plan to investigate this tradeoff in the future work.

Acknowledgement

This work is partially supported by the European projects SOFIA, GreenerBuildings and SM4All.

References

[1] Adya A., Dunagan J., Wolman A. Centrifuge: integrated lease management and partitioning for cloud services. *In Proceedings of the 7th USENIX conference on Networked systems design and implementation (NSDI)*, 2010.

[2] 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.*

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

[4] 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.

[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] Baldoni R., Bonomi S., Raynal M., Implementing a Regular Register in an Eventually Synchronous Distributed System prone to Continuous Churn *IEEE Transactions on Parallel and Distributed Systems* (TPDS), 2011 `http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.97`

[7] Baldoni R., Bonomi S., Soltani Nezhad A. Regular Registers in Dynamic Distributed Systems with Byzantine Processes: Bounds and Performance Analysis *Technical report - MIDLAB 3/11 - 2011.* A short version of this paper will appear in PODC 2011.

[8] Bazzi R. A., Synchronous Byzantine Quorum Systems, *Distributed Computing* 13(1), 45-52, 2000.

[9] 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.

[10] 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.

[11] Haldar S. and Vidyasankar K., Constructing 1-writer Multireader Multivalued Atomic Variables from Regular Variables. *JACM*, 42(1), 186-203, 1995.

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

[13] 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.

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

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

[16] Merritt M. and Taubenfeld G., Computing with Infinitely Many Processes, *in Proceedings of the 14th Int'l Symposium on Distributed Computing (DISC)*, 2000.

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