Implementing a Register in a Dynamic Distributed System

Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, Michel Raynal

PUBLICATION INTERNE
N° 1913

IMPLEMENTING A REGISTER IN A DYNAMIC DISTRIBUTED SYSTEM

R. BALDONI S. BONOMI A.-M. KERMARREC M. RAYNAL

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

Implementing a Register in a Dynamic Distributed System

R. Baldoni* S. Bonomi** A.-M. Kermarrec*** M. Raynal****

Systèmes communicants
Projet ASAP

Publication interne n ˚ 1913 — November 2008 — 19 pages

Abstract: Providing distributed processes with concurrent objects is a fundamental service that has to be offered by any distributed system. The classical shared read/write register is one of the most basic ones. Several protocols have been proposed that build an atomic register on top of an asynchronous message-passing system prone to process crashes. In the same spirit, this paper addresses the implementation of a regular register (a weakened form of an atomic register) in an asynchronous dynamic message-passing system. The aim is here to cope with the net effect of the adversaries that are asynchrony and dynamicity (the fact that processes can enter and leave the system). The paper focuses on the class of dynamic systems the churn rate c of which is constant. It presents two protocols, one applicable to synchronous dynamic message passing systems, the other one to asynchronous dynamic systems. Both protocols rely on an appropriate broadcast communication service (similar to a reliable brodcast). Each requires a specific constraint on the churn rate c. Both protocols are first presented in an as intuitive as possible way, and are then proved correct.

Key-words: Asynchronous message-passing system, Churn, Dynamic distributed system, Eventually synchronous distributed system, Infinite arrival model, Regular register, Synchronous system.

(Résumé : tsvp)

* Universitá La Sapienza, Roma, Italy, `Roberto.Baldoni@dis.uniroma1.it`
** Universitá La Sapienza, Roma, Italy, `Sivia.Bonomi@dis.uniroma1.it`
*** INRIA Bretagne Atlantique, Campus de Beaulieu, 35042, Rennes Cedex, France, `anne-marie.kermarrec@irisa.fr`
**** IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, `raynal@irisa.fr`

Centre National de la Recherche Scientifique (UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique et en Automatique – unité de recherche de Rennes

Implémentation d'un registre dans un système dynamique

Résumé : Ce rapport présente deux protocoles qui implémentent un registre régulier dans des systèmes dynamiques respctivement synchone et inéluctablement asynchrone.

Mots clés : Churn, registre régulier, Système asynchrone, Système synchrone.

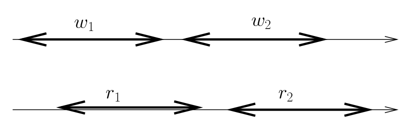
1 Introduction

On registers A *concurrent object* is an object that can be accessed by several processes. Among the concurrent objects, a *register* is certainly one of the most basic ones. A register provides the processes with two operations one that allows them to read the value of the object, the other one that allows them to define the new value of the object. According to the value domain of the register, the set of processes that are allowed to read it, the ones that are allowed to write it, and the specification of which value is the value returned by a read operation, a family of types of registers can be defined.

As far as the last point (the value returned by a read operation) is concerned, Lamport has defined three types of register [20].

- a *safe* register can be written by one writer only. Moreover, a read operation on such a register returns its current value if no write operation is concurrent with that read. In case of concurrency, the read can return *any* value of the value domain of the register (which means that a read concurrent with a write can return a value that has never been written). This type of register is very weak.
- A *regular* register can have any number of writers and any number of readers. The writes appear as if they were executed sequentially, this sequence complying with their real time order (i.e., if two writes w_1 and w_2 are concurrent they can appear in any order, but if w_1 terminates before w_2 starts, w_1 has to appear as being executed before w_2).

  As far as a read operation is concerned we have the following. If no write operation is concurrent with a read operation, that read operation returns the current value kept in the register. Otherwise, the read operation returns any value written by a concurrent write operation of the last value of the register before these concurrent writes. A regular register is stronger than a safe register, as the value returned in presence of concurrent write operations is no longer arbitrary.

  Nevertheless, a regular register can exhibit what is called a *new/old inversion*. The figure on the right depicts two write operations w_1 and w_2 and two read operations r_1 and r_2 that are concurrent (r_1 is concurrent with w_1 and w_2, while r_2 is concurrent with w_2 only). According to the definition of register regularity, it is possible that r_1 returns the value written by w_2 while r_2 returns the value written by w_1.
- An *atomic* register is a regular register without new/old inversion. This means that an atomic register is such that all its read and write operations appear as if they have been executed sequentially, this sequential total order respecting the real time order of the operations. (Linearizability [18] is nothing else than atomicity when we consider objects defined by a sequential specification).

Interestingly enough, safe, regular and atomic registers have the same computational power. This means that it is possible to implement a multi-writer/multi-reader atomic register from single-writer/single-reader safe registers. There is a huge number of papers in the literature discussing such transformations (e.g., [5, 7, 16, 21, 27, 29, 30] to cite a few).

On message-passing dynamic distributed systems The advent of new classes of applications (social networks, security, smart objects sharing etc) and technologies (VANET, WiMax, Airborn Networks, DoD Global Information Grid, P2P) are radically changing the way in which distributed systems are perceived. Such emerging systems have a composition, in term processes participating to the system, that is self-defined at run time depending, for example, on their will to belong to such a system, on the geographical distribution of processes etc. Therefore one of the common denominators of such emerging systems is the dynamicity dimension that introduces a new source of unpredictability inside a distributed system.

As a consequence, any specification of a distributed computing abstraction (e.g., registers, communication, consensus) has to take this dynamicity into account and protocols implementing them have to be correct despite the fact that processes enter and leave the system at will. This dynamicity makes abstractions more difficult to understand and master than in traditional distributed systems in which a system starts and remains forever with the same set of processes. The *churn* notion has been introduced as a system parameter that aims at capturing this dynamicity and makes it tractable by distributed protocols (e.g., [19, 22, 24]). Although numerous protocols have been designed for dynamic distributed message-passing systems, very few papers (such as [2, 4, 28]) strive to present models suited to such systems, and extremely few dynamic protocols have been proved correct. Up to now, the most common approach used to address dynamic systems is mainly experimental.

Contribution and roadmap This paper addresses the implementation of a regular register abstraction in a synchronous and eventually synchronous message-passing distributed system subject to a constant churn. We focus on regular registers as they have the same computability power as the atomic registers but are easier to implement in a traditional distributed system [3]. Moreover, interestingly enough, regular registers allow solving some basic coordination problems such as the consensus problem in asynchronous systems prone to crash but equipped with an appropriate leader election service [11, 14]. Therefore this paper makes a step in giving a clear specification of the abstraction of regular register in a dynamic context and provides two formally correct implementations of the specification.

To that end, Section 2 first introduces base definitions. Then, the paper considers three type of dynamic systems, namely synchronous dynamic systems (Section 3), fully asynchronous dynamic systems (Section 4), and eventually synchronous dynamic systems (Section 5). It presents two protocols that build a regular register (in synchronous and eventually synchronous systems) and shows that it is impossible to build a regular register in a fully asynchronous dynamic system. The dynamicity attribute of the three models is defined from the churn parameter.

2 Base definitions

2.1 Dynamic system

The processes In a dynamic system, entities may join and leave the system at will. Consequently, at any point in time, the system is composed of the processes (nodes) that have joined and have not yet left the system. In order to model processes continuously arriving to and departing from the system, we assume the infinite arrival model (as defined in [25]). In each run, infinitely many processes $\Pi = \{\dots, p_i, p_j, p_k, \dots\}$ may a priori join the system, but at any point in time the number of processes is bounded. Processes are sequential processing entities (sometimes called nodes). Moreover, we assume that the processes are uniquely identified (with their indexes).

Time model The underlying time model is the set of positive integers.

Entering and leaving the system When a process p_i enters the system it executes the operation $\mathsf{join}()$. That operation, invoked at some time τ, is not instantaneous: it consumes time. But, from time τ, the process p can receive and process messages sent by any other process that belongs to the system.

To explicit the "begin of a $\mathsf{join}()$" notion (time τ), let us consider the case of mobile nodes in a wireless network. The beginning of its $\mathsf{join}()$ occurs when a process (node) enters the geographical zone within which it can receive messages. The end of the $\mathsf{join}()$ depends on the code associated with that operation. The important point is that the process p is in the listening mode from the beginning of $\mathsf{join}()$. If the code of the join() operation contains waiting periods, the process p can receive and process messages during these waiting periods. So, a process is in the listening mode since the invocation of join(), and proceeds to the active mode only when the join() operation terminates. It remains in the active mode until it leaves the system.

A process leaves the system in an implicit way. When it does, it leaves the system forever and does not longer send or receive messages. From a practical point of view, if a process wants to re-enter the system, it has to enter it as a new process (i.e., with a new name). Let us observe that, while, from the application point of view, a crash is an involuntary departure, there is no difference with a voluntary leave from the model point of view. So, considering a crash as an unplanned leave, the model can take them into account without additional assumption.

Definition 1 *A process is* active *from the time it returns from the* join() *operation until the time it leaves the system.* $A(\tau)$ *denotes the set of processes that are active at time* τ*, while* $A[\tau_1, \tau_2]$ *denotes the set of processes that are active during the time interval* $[\tau_1 : \tau_2]$ *(hence,* $A(\tau) = A[\tau, \tau]$*).*

Churn rate The dynamicity of the joins and leaves of the processes is captured by the system parameter called *churn*. As in a lot of other papers we consider here the *churn rate*, denoted c, defined as the percentage of the nodes that are "refreshed" at every time unit ($c \in [0, 1]$). This means that, while the number of processes remains constant (equal to n), in every time unit $c \times n$ processes leave the system and the same number of processes join the system. It is shown in [19] that this assumption is fairly realistic for several classes of applications built on top of dynamic systems.

2.2 Regular register

The notion of a regular register defined in the introduction has to be adapted to a dynamic system. We consider that a protocol implements a regular register in a dynamic system if the following properties are satisfied.

- Liveness: If a process invokes a read or a write operation and does not leave the system, it eventually returns from that operation.
- Safety: A read operation returns the last value written before the read invocation, or a value written by a write operation concurrent with it.

It is easy to see that these properties boil down to the classical definition if the system is static. Moreover, it is assumed that a process invokes the read or write operation only after it has returned from its join() invocation.

3 Regular register in a synchronous system

3.1 Assumptions

This section presents a protocol that implements a one-writer/multi-reader regular register[1] in a dynamic system where the churn rate c is constant, the number n is known by every process, and the processes can access a global clock[2]. The protocol assumes that the churn rate is such that $c < 1/(3\delta)$ (where δ is a bound on communication delays defined below). Let us observe that, while relating c to δ, this constraint is independent of the system size n (as we will see, this will be different for eventually synchronous systems).

[1]Actually, the protocol works for any number of writers as long as the writes are not concurrent. Considering a single writer makes the exposition easier.

[2]The global clock is for ease of presentation. As we are in a synchronous system, this global clock can be implemented by synchronized local clocks.

3.2 Synchronous system

The system is synchronous in the following sense. The processing times of local computations (but the wait statement) are negligible with respect to communication delays, so they are assumed to be equal to 0. Contrarily, messages take time to travel to their destination processes.

Point-to-point communication The network allows a process p_i to send a message to another process p_j as soon as p_i knows that p_j has entered the system. The network is reliable in the sense that it does not loose, create or modify messages. Moreover, the synchrony assumption guarantees that if p_i invokes "send m to p_j" at time τ, then p_j receives that message by time $\tau + \delta'$ (if it has not left the system by that time). In that case, the message is said to be "sent" and "received".

Broadcast It is assumed that the system is equipped with an appropriate broadcast communication subsystem that provides the processes with two operations, denoted broadcast() and deliver(). The former allows a process to send a message to all the processes in the system, while the latter allows a process to deliver a message. Consequently, we say that such a message is "broadcast" and "delivered". These operations satisfy the following property.

- Timely delivery: Let τ be the time at which a process p invokes broadcast(m). There is a constant δ ($\delta \geq \delta'$) (known by the processes) such that if p does not leave the system by time $\tau + \delta$, then all the processes that are in the system at time τ and do not leave by time $\tau + \delta$, deliver m by time $\tau + \delta$.

Such a pair of broadcast operations has first been formalized in [15] in the context of systems where process can commit crash failures. It has been extended to the context of dynamic systems in [10].

3.3 A protocol for synchronous dynamic systems

The principle that underlies the design of the protocol is to have *fast reads* operations: a process willing to read has to do it locally. From an operational point of view, this means that a read is not allowed to use a wait() statement, or to send messages and wait for associated responses. Hence, albeit the proposed protocol works in all cases, it is targeted for applications where the number of reads outperforms the number of writes.

Local variables at a process p_i Each process p_i has the following local variables.

- Two variables denoted $register_i$ and sn_i; $register_i$ contains the local copy of the regular register, while sn_i is the associated sequence number.
- A boolean $active_i$, initialized to $false$, that is switched to $true$ just after p_i has joined the system.
- Two set variables, denoted $replies_i$ and $reply\_to_i$, that are used during the period during which p_i joins the system. The local variable $replies_i$ contains the 3-uples $< id, value, sn >$ that p_i has received from other processes during its join period, while $reply\_to_i$ contains the processes that are joining the system concurrently with p_i (as far as p_i knows).

Initially, n processes compose the system. The local variables of each of these processes p_k are such that $register_k$ contains the initial value of the regular register[3], $sn_k = 0$, $active_k = true$, and $replies_k = reply\_to_k = \emptyset$.

[3]Without loss of generality, we assume that at the beginning every process p_k has in its variable $register_k$ the value 0

The join() **operation** When a process p_i enters the system, it first invokes the join operation. The algorithm implementing that operation, described in Figure 1, involves all the processes that are currently present (be them active or not).

First p_i initializes its local variables (line 01), and waits for a period of δ time units (line 02); This waiting period is explained later. If $register_i$ has not been updated during this waiting period (line 03), p_i broadcasts (with the broadcast() operation) an INQUIRY(i) message to the processes that are in the system (line 05) and waits for 2δ time units, i.e., the maximum round trip delay (line 06)[4]. When this period terminates, p_i updates its local variables $register_i$ and sn_i to the most uptodate values it has received (lines 07-08). Then, p_i becomes active (line 10), which means that it can answer the inquiries it has received from other processes, and does it if $reply\_to \neq \emptyset$ (line 11). Finally, p_i returns ok to indicate the end of the join() operation (line 12).

operation join(i):
(01) $register_i \leftarrow \perp$; $sn_i \leftarrow -1$; $active\_i \leftarrow false$; $replies_i \leftarrow \emptyset$; $reply\_to_i \leftarrow \emptyset$;
(02) wait(δ);
(03) **if** ($register_i = \perp$) **then**
(04) $\quad replies_i \leftarrow \emptyset$;
(05) $\quad$ broadcast INQUIRY(i);
(06) $\quad$ wait(2δ);
(07) $\quad$ **let** $< id, val, sn > \in replies_i$ **such that** $(\forall < -, -, sn' > \in replies_i : \; sn \geq sn')$;
(08) $\quad$ **if** $(sn > sn_i)$ **then** $sn_i \leftarrow sn$; $register_i \leftarrow val$ **end if**
(09) **end if**;
(10) $active_i \leftarrow true$;
(11) **for each** $j \in reply\_to_i$ **do** send REPLY $(< i, register_i, sn_i >)$ to p_j;
(12) return(ok).

---

(13) **when** INQUIRY(j) is delivered:
(14) $\quad$ **if** $(active_i)$ **then** send REPLY $(< i, register_i, sn_i >)$ to p_j
(15) $\quad\quad$ **else** $reply\_to_i \leftarrow reply\_to_i \cup \{j\}$
(16) $\quad$ **end if**.

(17) **when** REPLY$(< j, value, sn >)$ is received: $replies_i \leftarrow replies_i \cup \{< j, value, sn >\}$.

Figure 1: The join() protocol for a synchronous system (code for p_i)

When a process p_i receives a message INQUIRY(j), it answers p_j by return sending back a REPLY$(< i, register_i, sn_i >)$ message containing its local variable if it is active (line 14). Otherwise, p_i postpones its answer until it becomes active (line 15 and lines 10-11). Finally, when p_i receives a message REPLY$(< j, value, sn >)$ from a process p_j it adds the corresponding 3-uple to its set $replies_i$ (line 17).

The read() **and** write(v) **operations** The algorithms for the read and write operations associated with the regular register are described in Figure 2. The read is purely local (i.e., fast): it consists in returning the current value of the local variable $register_i$.

[4]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.

The write consists in disseminating the new value v (together with its sequence number) to all the processes that are currently in the system (line 01). 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 02).

operation $\mathsf{read}()$: $\mathsf{return}(register_i)$. % issued by any process p_i %

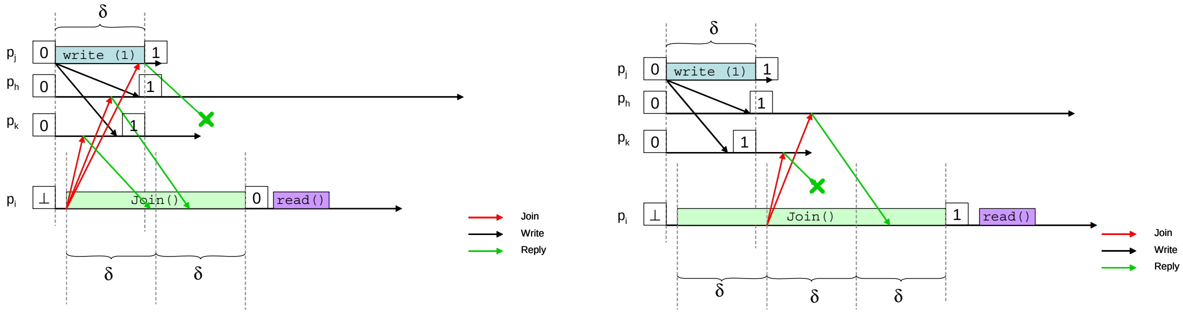
---

operation $\mathsf{write}(v)$: % issued only by the writer p_w%
(01) $sn_w \leftarrow sn_w + 1$; $register_i \leftarrow v$ broadcast $\mathsf{WRITE}(v, sn_w)$;
(02) $\mathsf{wait}(\delta)$; $\mathsf{return}(ok)$.

(03) **when** $\mathsf{WRITE}(< val, sn >)$ is delivered: % at any process p_i %
(04) **if** $(sn > sn_i)$ **then** $register_i \leftarrow val$; $sn_i \leftarrow sn$ **end if**.

Figure 2: The $\mathsf{read}()$ and $\mathsf{write}()$ protocols for a synchronous system

Why the $\mathsf{wait}(\delta)$ statement at line 02 of the $\mathsf{join}()$ operation? To motivate the $\mathsf{wait}(\delta)$ statement at line 02, let us consider the execution of the $\mathsf{join}()$ operation depicted in Figure 3(a). At time τ, the processes p_j, p_h and p_k are the three processes composing the system, and p_j is the writer. Moreover, the process p_i executes $\mathsf{join}()$ just after τ.The value of the copies of the regular register is 0 (square on the left of p_j, p_h and p_k), while $register_i = \perp$ (square on its left). The 'timely delivery" property of the broadcast invoked



(a) Without $\mathsf{wait}(\delta)$

(b) With $\mathsf{wait}(\delta)$

Figure 3: Why $\mathsf{wait}(\delta)$ is required

by the writer p_j ensures that p_j and p_k deliver the new value $v = 1$ by $\tau + \delta$. But, as it entered the system after τ, there is no such a guarantee for p_i. Hence, if p_i does not execute the $\mathsf{wait}(\delta)$ statement at line 02, its execution of the lines 03-09 can provide it with the previous value of the regular register, namely 0. If after obtaining 0, p_i issues another read it obtains again 0, while it should obtain the new value $v = 1$ (because 1 is the last value written and there is no write concurrent with this second read issued by p_i).

The execution depicted in Figure 3(b) shows that this incorrect scenario cannot occur if p_i is forced to wait for δ time units before inquiring to obtain the last value of the regular register.

3.4 Proof of the synchronous protocol

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

Proof The read() operation trivially terminates. The termination of the join() and write() operations follows from the fact that the wait() statement terminates. $\Box_{Lemma\ 1}$

Let us recall that $A[\tau, \tau+x]$ denotes the set of processes that are active (at least) during the period $[\tau, \tau+x]$.

Lemma 2 *Let* $c < 1/3\delta$. $\forall \tau : \ |A[\tau, \tau+3\delta]| \geq n(1-3\delta c) > 0$.

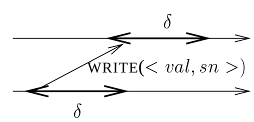
Proof Let us first consider the case $\tau_0 = 0$. We have $|A(\tau_0)| = n$. Then, due to definition of c, we have $|A[\tau_0, \tau_0+1]| = n - nc$. During the second time unit, nc new processes enter the system and replace the nc processes that left the system during that time unit. In the worst case, the nc processes that left the system are processes that were present at time τ_0 (i.e., they are not processes that entered the system between τ_0 and τ_0+1). So, we have $|A[\tau_0, \tau_0+2]| \geq n - 2nc$. If we consider a period of 3δ time units, i.e. the longest period needed to terminate a join operation, we obtain $|A[\tau_0, \tau_0+3\delta]| \geq n - 3\delta nc = n(1-3\delta c)$. Moreover, as $c < 1/3\delta$, we have $|A[\tau, \tau+3\delta]| \geq n(1-3\delta c) > 0$.

It is easy to see that the previous reasoning depends only on (1) the fact that there are n processes at each time τ, and (2) the definition of the churn rate c, from which we conclude that $\forall \tau : \ |A[\tau, \tau+3\delta]| \geq n(1-3\delta c)$. $\Box_{Lemma\ 2}$

Lemma 3 *Let* $c < 1/3\delta$. *When a process* p_i *terminates the execution of* join(), *its local variable* $register_i$ *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 Let p_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 $register_i = \perp$ evaluated at line 03 of the join operation.

- Case $register_i \neq \perp$. We then conclude that p_i has received a WRITE($< val, sn >$) message and accordingly updated $register_i$ line 04, Figure 2. As (1) the write operation lasts δ time units (line 02 of the write operation, Figure 2), (2) the join operation lasts at least δ time units, and (3) the message WRITE($< val, sn >$) - sent at the beginning of the write - takes at most δ time units, it follows from $register_i \neq \perp$ that the join() and the write() operations overlap, i.e., they are concurrent, which proves the lemma for that case. (The worst case scenario is depicted on the right.)



- $register_i = \perp$. In that case, p_i broadcasts an INQUIRY(i) message and waits for 2δ time units (lines 05-06 of the join() operation, Figure 1). Let τ be the time at with p_i broadcasts the INQUIRY(i) message. At the end of the 2δ round trip upper bound delay, p_i updates $register_i$ with the value associated with the highest sequence number it has received (lines 07-08). We consider two sub-cases.

  – No write is concurrent with the join operation. As $|A[\tau, \tau+3\delta]| \geq n(1-3\delta c) > 0$ (Lemma 2), $A[\tau, \tau+3\delta]$ is not empty. Consequently, at least one process that has a copy of the last written value answers the inquiry of p_i and consequently p_i sets $register_i$ to that value by 2δ time units after the broadcast, which proves the lemma.

– There is (at least) one write issued by a process p_j concurrent with the join operation. In that case, p_i can receive both WRITE($< val, sn >$) messages and REPLY($< j, val, sn >$) messages. According the values received at time $\tau + 2\delta$, p_i will update $register_i$ to the value written by a concurrent update, or the value written before the concurrent writes.

$\Box_{Lemma\ 3}$

Lemma 4 Safety. *Let $c < 1/3\delta$. A* read() *operation returns the last value written before the read invocation, or a value written by a write operation concurrent with it.*

Proof Let us observe that a read by any process can be invoked only after that process has terminated its join operation. It follows from that observation, Lemma 3, and the lines 03-04 associated with the write operation (Figure 2) that a read always returns a value that has been written.

Let us observe that a write that starts at time τ, terminates at time $\tau+\delta$, and all the processes in $A[\tau, \tau+\delta]$ have delivered the value it has written by $\tau + \delta$. Considering that observation, the proof that a read operation returns the last written value or a value written by a concurrent write is similar to the proof of the previous lemma. It is left to the reader. $\Box_{Lemma\ 4}$

Theorem 1 *Let $c < 1/3\delta$. The protocol described is the Figures 1 and 2 implements a regular register in a synchronous dynamic system.*

Proof The proof follows directly from Lemmas 1 and 4. $\Box_{Theorem\ 1}$

4 Regular register in an asynchronous system

This section shows that (not surprisingly) it is not possible to implement a regular register in a fully asynchronous dynamic system. Such a system is similar to the synchronous system defined in Section 3 with the following difference: there are no bound on message transfer delays such as δ and δ'.

Theorem 2 *It is not possible to implement a regular register in a fully asynchronous dynamic system.*

Proof The proof is a consequence of the impossibility to implement a register in a fully asynchronous static message-passing system prone to any number of process crashes [3][5]. More specifically, if processes enter and leave the system, due to the absence of an upper bound on message transfer delays, it is easy possible a run in which the value obtained by a process is always older than the last value whose write is terminated.

$\Box_{Theorem\ 2}$

5 Regular register in an eventually synchronous system

5.1 System model

In static systems, an *eventually synchronous* distributed system[6] is a system that after an unknown but finite time behaves synchronously [6, 9]. We adopt the same definition for dynamic systems. More precisely:

[5]This paper shows also that such a construction is possible as soon as a majority of processes do not crash.

[6]Sometime also called *partially synchronous* system.

- As in an asynchronous system, there is a time notion (whose domain is the set of integers), but this time notion is inaccessible to the processes. The definition of the churn rate c is the same as in Section 2.1, i.e., the rate of the number of processes that join/leave the system per time unit.
- Eventual timley delivery: There is a time τ and a bound δ such that any message sent (broadcast) at time $\tau' \geq \tau$, is received (delivered) by time $\tau' + \delta$ to the processes that are in the system during the interval $[\tau', \tau' + \delta]$.

It is important to observe that the date τ and the bound δ do exist but can never be explicitly known by the processes.

5.2 Assumptions

The proposed protocol is based on the following assumptions.

- $\forall\ \tau :\ |A(\tau)| > \frac{n}{2}$. This assumption states that, at any time τ, a majority of the n processes that currently compose the system are active. Let us recall that a process becomes *active* as soon as it has obtained a copy of the regular register. This assumption is similar to the "majority of non-faulty processes" assumption used in the design of fault-tolerant protocols suited to asynchronous systems prone to process crashes. The "spirit" of these assumptions is the same: the majority of active (resp., non-faulty) processes is needed to preserve the consistency of the regular register (resp., system state).

- $c < 1/(3\delta n)$. This assumption restricts the churn rate that defines the dynamicity of the system. Differently from synchronous systems, it involves both the eventual bound δ and the size n of the system.

5.3 A protocol for eventually synchronous dynamic system

As it cannot rely on the passage of time combined with a safe known bound on message transfer delay, the protocol is based on acknowledgment messages. Moreover, it allows any process to write under the assumption that no two processes write concurrently (this assumption could be implemented with appropriate quorums, but we do not consider its implementation in this paper).

Local variables at a process p_i Each process p_i has to manage local variables, where (as before) $register_i$ denotes the local copy of the regular register. In order to ease the understanding and the presentation of the protocol, the control variables that have the same meaning as in the synchronous protocol are denoted with the same identifiers (but their management can differ). Those are the variables denoted sn_i, $active_i$, $replies_i$ and $reply\_to_i$ in Section 3. In addition to these control variables, the protocol uses the following ones.

- $read\_sn_i$ is a sequence number used by p_i to distinguish its successive read requests. The particular value $read\_sn_i = 0$ is used by the join operation.
- $reading_i$ is boolean whose value is true when p_i is reading.
- $write\_ack_i$ is a set used by p_i (when it writes a new value) to remember the processes that have acknowledged its last write.
- 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) or while they are reading. When it terminates its join operation, p_i has to send them a reply to prevent them to be blocked forever.

Remark In the following, the processing associated with a message reception (if the message has been sent) or a message delivery (if the message has been broadcast) appears in the description of one of the operations join(), read(), or write(). But the sending (or broadcast) of the message that triggers this processing is not necessarily restricted to that operation.

The join() operation The algorithm implementing this operation is described in Figure 4. (The read algorithm -Figure 5- can be seen as a simplified version of it.)

After having initialized its local variables, the process p_i broadcasts an INQUIRY$(i, read_sn_i)$ message to inform the other processes that it enters the system and wants to obtain the value of the regular register (line 03, as indicated $read\_sn_i$ is then equal to 0). Then, after it has received "enough" replies (line 04), p_i proceeds as in the synchronous protocol: it updates its local pair $(register_i, sn_i)$ (lines 05-06), becomes active (line 07), and sends a reply to the processes in the set $reply\_to_i$ (line 08-10). It sends such a reply message also to the processes in its set dl_prev_i to prevents them from waiting forever. In addition to the tern $< i, register_i, sn_i >$, a reply message sent to a process p_j, from a process p_i, has now to carry also the read sequence number r_sn that identifies the corresponding request issued by p_j.

operation join(i):
(01) $register_i \leftarrow \perp$; $sn_i \leftarrow -1$; $active_i \leftarrow false$; $reading_i \leftarrow false$; $replies_i \leftarrow \emptyset$; $reply\_to_i \leftarrow \emptyset$;
(02) $write\_ack_i \leftarrow \emptyset$; $dl\_prev_i \leftarrow \emptyset$; $read\_sn_i \leftarrow 0$;
(03) broadcast INQUIRY$(i, read_sn_i)$;
(04) wait until$\left(|replies_i| > \frac{n}{2}\right)$;
(05) **let** $< id, val, sn > \in replies_i$ **such that** $(\forall < -, -, sn' > \in replies\_i : \ sn \geq sn')$;
(06) **if** $(sn > sn_i)$ **then** $sn_i \leftarrow sn$; $register_i \leftarrow val$ **end if**
(07) $active_i \leftarrow true$;
(08) **for each** $< j, r\_sn > \in reply\_to_i \cup dl\_prev_i$ **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) **if** $(reading_i)$ **then** send DL_PREV (i, r_sn) to p_j **end if**;
(15) **else** $reply\_to_i \leftarrow reply\_to_i \cup \{< j, r\_sn >\}$;
(16) send DL_PREV (i, r_sn) to p_j
(17) **end if**.

(18) **when** REPLY$(< j, value, sn >, r_sn)$ is received:
(19) **if** $((r\_sn = read\_sn_i)$ **then**
(20) $replies_i \leftarrow replies_i \cup \{< j, value, sn >\}$; send ACK (i, r_sn) to p_j
(21) **end if**.

(22) **when** DL_PREV(j, r_sn) is received: $dl\_prev_i \leftarrow dl\_prev_i \cup \{< j, r\_sn >\}$.

Figure 4: The join() protocol for an eventually synchronous system (code for p_i)

The behavior of p_i when it receives an INQUIRY(j, r_sn) message is similar to one of the synchronous protocol, with two differences. The first is that 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 16). The second difference is that, if p_i is reading, it also sends a DL_PREV() message to p_j (line 14); this is required to have p_j send to p_i the value it has obtained when it terminates its join operation.

When p_i receives a REPLY$(< j, value, sn >, r_sn)$ message from a process p_j, if the reply message is an answer to its INQUIRY$(i, read_sn)$ message (line19), p_i adds $< j, value, sn >$ to the set of replies it has received so far and sends back an ACK(i, r_sn) message to p_j (line 20).

Finally, when p_i receives a message DL_PREV(j, r_sn), it adds its content to the set dl_prev_i (line 22), in order to remember that it has to send a reply to p_j when it will become active (lines 08-09).

The $\mathsf{read}()$ **operation** The $\mathsf{read}()$ and $\mathsf{join}()$ operations are strongly related. More specifically, a read is a simplified version of the join operation[7]. Hence, the code of the $\mathsf{read}()$ operation, described in Figure 5, is a simplified version of the code of the $\mathsf{join}()$ operation.

Each read invocation is identified by a pair made up of the process index i and a read sequence number $read\_sn_i$ (line 03). So, p_i first broadcasts a read request READ$(i, read_sn_i)$. Then, after it has received "enough" replies, p_i selects the one with the greatest sequence number, updates (if needed) its local pair $(register_i, sn_i)$, and returns the value of $register_i$.

When it receives a message READ(j, r_sn), p_i sends back a reply to p_j if it is active (line 09). If it is joining the system, it remembers that it will have to send back a reply to p_j when it will terminate its join operation (line 10).

operation $\mathsf{read}(i)$:

(01) $read\_sn_i \leftarrow read\_sn_i + 1$;

(02) $replies_i \leftarrow \emptyset$; $reading_i \leftarrow true$;

(03) broadcast READ$(i, read_sn_i)$;

(04) wait until$(|replies_i| > \frac{n}{2})$;

(05) **let** $< id, val, sn > \in replies_i$ **such that** $(\forall < -, -, sn' > \in replies_i : \ sn \geq sn')$;

(06) **if** $(sn > sn_i)$ **then** $sn_i \leftarrow sn$; $register_i \leftarrow val$ **end if**;

(07) $reading_i \leftarrow false$; return$(register_i)$.

---

(08) **when** READ(j, r_sn) is delivered:

(09) **if** $(active_i)$ **then** send REPLY $(< i, register_i, sn_i >, r\_sn)$ to p_j

(10) **else** $reply\_to_i \leftarrow reply\_to_i \cup \{< j, r\_sn >\}$

(11) **end if**.

Figure 5: The $\mathsf{read}()$ protocol for an eventually synchronous system (code for p_i)

The $\mathsf{write}()$ **operation** The code of the write operation is described in Figure 6. Let us recall that it is assumed that a single process at a time issues a write.

When a process p_i wants to write, it issues first a read operation in order to obtain the sequence number associated with the last value written (line 01)[8]. Then, after it has broadcast the message WRITE$(i, < v, sn_i >)$ to disseminate the new value and its sequence number to the other processes (line 04), p_i waits until it has received "enough" acknowledgments. When this happens, it terminates the write operation by returning the control value ok (line 05).

When it is delivered a message WRITE$(j, < val, sn >)$, p_i takes into account the pair (val, sn) if it is more uptodate than its current pair (line 07). In all cases, it sends back to the sender p_j a message ACK (i, sn) to allow it to terminate its write operation (line 08).

[7] As indicated before, the "read" identified $(i, 0)$ is the $\mathsf{join}()$ operation issued by p_i.

[8] As the write operations are not concurrent, this read obtains the greatest sequence number. The same strategy to obtain the last sequence number is used in protocols implementing an atomic registers (e.g., [3, 10]).

When it receives a message ACK (j, sn), p_i adds it to its set $write\_ack_i$ if this message is an answer to its last write (line 10).

operation write(v):
(01) read(i);
(02) $sn_i \leftarrow sn_i + 1$; $register_i \leftarrow v$;
(03) $write\_ack_i \leftarrow \emptyset$;
(04) broadcast WRITE$(i, < v, sn_i >)$;
(05) wait until$(|write_ack_w| > \frac{n}{2})$; return$(ok)$.

---

(06) **when** WRITE$(j, < val, sn >)$ is delivered:
(07) **if** $(sn > sn_i)$ **then** $register_i \leftarrow val$; $sn_i \leftarrow sn$ **end if**;
(08) send ACK (i, sn) to p_j.

(09) **when** ACK(j, sn) is received:
(10) **if** $(sn = sn_i)$ **then** $write\_ack_i \leftarrow write\_ack_i \cup \{j\}$ **end if**.

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

5.4 Proof of the eventually synchronous protocol

Lemma 5 *Let us assume that (1) $\forall\ \tau\ :\ |A(\tau)| > \frac{n}{2}$, and (2) a process that invokes the* join() *operation remains in the system for at least 3δ time units. If a process p_i invokes the* join() *operation and does not leave the system, this join operation terminates.*

Proof Let us first observe that, in order to terminate its join() operation, a process p_i has to wait until its set $replies_i$ contains $\frac{n}{2}$ elements (line 04, Figure 4). Empty at the beginning of the join operation (line 01, Figure 4), this set is filled in by p_i when it receives the corresponding REPLY() messages (line 20 of Figure 4).

A process p_j sends a REPLY() message to p_i if (i) either it is active and has received an INQUIRY message from p_i, (line 13, Figure 4), or (ii) it terminates its join() join operation and $< i, - > \in reply\_to_j \cup dl\_prev_j$ (lines 08-09, Figure 4).

Let us suppose by contradiction that $|replies_i|$ remains smaller than $\frac{n}{2}$. This means that p_i does not receive enough REPLY() carrying the appropriate sequence number. Let τ be the time at which the system becomes synchronous and let us consider a time $\tau' > \tau$ at which a new process p_j invokes the join operation. At time τ', p_j broadcasts an INQUIRY message (line 03, Figure 4). As the system is synchronous from time τ, every process present in the system during $[\tau', \tau' + \delta]$ receives such INQUIRY message by time $\tau' + \delta$.

As it is not active yet when it receives p_j's INQUIRY message, the process p_i executes line 16 of Figure 4 and sends back a DL_PREV message to p_j. Due to the assumption that every process that joins the system remains inside for at least 3δ time units, p_j receives p_i's DL_PREV and executes consequently line 22 (Figure 4) adding $< i, - >$ to dl_prev_j.

Due to the assumption that there are always at least $\frac{n}{2}$ active processes in the system, we have that at time $\tau' + \delta$ at least $\frac{n}{2}$ processes receive the INQUIRY message of p_j, and each of them will execute line 13 (Figure 4) and send a REPLY message to p_j. Due to the synchrony of the system, p_j receives these messages by time $\tau' + 2\delta$ and then stops waiting and becomes active (line 07, Figure 4). Consequently (lines 08-09) p_j sends a REPLY to p_i as $i \in reply\_to_j \cup dl\_prev_j$. By δ time units, p_i receives that REPLY message and executes line 20, Figure 4. Due to churn rate, there are an infinity of processes invoking the join after time τ

and p_i will receive a reply from any of them so p_i will fill in its set $replies_i$ and terminate its join operation. $\Box_{Lemma\ 5}$

Lemma 6 *Let us assume that (1) $\forall\ \tau\ :\ |A(\tau)| > \frac{n}{2}$, and (2) a process that invokes the* join() *operation remains in the system for at least 3δ time units. If a process p_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 5. In fact the read is a simplified case of the join algorithm in which the process p_i that issues the operation is already active. Due top page limitation, the proof is left to the reader. $\Box_{Lemma\ 6}$

Lemma 7 *Let us assume that (1) $\forall\ \tau\ :\ |A(\tau)| > \frac{n}{2}$, and (2) a process that invokes the* join() *operation remains in the system for at least 3δ time units. If a process invokes* write() *and does not leave, this write operation terminates.*

Proof Let us first assume that the read() operation invoked at line 01 terminates (this is proved in Lemma 6). Before terminating the write of a value v with a sequence number snb a process p_i has to wait until its set $write\_ack_i$ contains at least $\frac{n}{2}$ elements (line 05, Figure 6). Empty at the beginning of the write operation (line 03), this set is filled in when the ACK$(-, snb)$) messages are delivered to p_i (line 10). Such an ack message is sent by every process p_j such that (i) p_j receives the corresponding WRITE message from p_i (line 08) or (ii) p_j receives a REPLY message for its join from p_i (line 20, Figure 4).

Suppose by contradiction that p_i never fills in $write\_ack_i$. This means that p_i misses ACK() messages carrying the sequence number snb. Let us consider the time τ at which the system becomes synchronous, i.e., every message sent by any process p_j at time $\tau' > \tau$ is delivered by time $\tau' + \delta$. Due to the assumption that the writer does not leave before the termination of its write, it follows that p_i will receive all the INQUIRY messages sent by processes joining after time τ. When it receives an INQUIRY() message from some joining process p_j, p_i executes line 13 of Figure 4[9] and sends a REPLY message to p_j with the sequence number snb. Since, after τ, the system is synchronous, p_j receives this REPLY message in at most δ time units and executes line 20 of Figure 4 sending back an ACK$(-, snb)$ message to p_i. As (1) by assumption a process that joins the system does not leave for at least 3δ time units and (2) the system is now synchronous, such an ACK$(-, snb)$ message is received by p_i in at most δ time units and consequently p_i executes line 10 and adds p_j to the set $write\_ack_i$. Due to the dynamicity of the system (captured by the churn rate c), processes continuously join the system. Due to the chain of messages INQUIRY(), REPLY(), ACK(), the reception of each message triggering the sending of the next one, it follows that p_i eventually receives $\frac{n}{2}$ ACK$(-, snb)$ messages and terminates its write operation. $\Box_{Lemma\ 7}$

Theorem 3 Termination. *Let us assume that $\forall\ \tau\ :\ |A(\tau)| > \frac{n}{2}$. If a process invokes* join(), read() *or* write (), *and does not leave the system, it terminates its operation.*

Proof It follows from Lemma 5, Lemma 6 and Lemma 7. $\Box_{Theorem\ 3}$

Theorem 4 Safety. *Let us assume that $\forall\ \tau\ :\ |A(\tau)| > \frac{n}{2}$. A read operation returns the last value written before the read invocation, or a value written by a write operation concurrent with it.*

[9]The writer process p_i executes line 13 of Figure 4 because a writer is always in the active mode.

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

Let τ_0 be the starting time. It follows from the initialization statement, that the n processes that initially define the system are active and contain the initial value of the regular register (say v_0). Consequently, we have $|A(\tau_0)| = |W_0(\tau_0)| > \frac{n}{2}$.

Let $\tau_y = \tau_0 + y$ (the time instant that is y time units after τ_0). By time τ_1, nc processes leave the system and nc processes invoke the join operation. All the processes that leave were active at time τ_0 and their local copy of the register contained v_0. Hence, $|A(\tau_1)| > \frac{n}{2}$ and $|W_0(\tau_1)| > \frac{n}{2}$. As $\forall\ \tau : \ |A(\tau)| > \frac{n}{2}$.(assumption), at most one process in $A(\tau_1)$ (and then also in $W_0(\tau_1)$) can leave before any process entering the system terminates its join operation.

Let p_i be the first process that terminates its join operation. If no write operation is concurrent with that join, then all of the replies received by p_i come from processes in $W_0(\tau_1)$. Each of these processes stores the last value written (namely, the initial value v_0) in its local copy of the register together with the sequence number 0. When p_i executes the lines 05-06 of the $\mathsf{join}()$ operation (Figure 4), it updates its local variable with the value v_0.

If there is a concurrent write operation ($\mathsf{write}_1(v_1)$) issued by a process, it is possible that, before the end of this write, (1) active processes receive the WRITE message and consequently update their local copy of the register with v_1 (line 05, Figure 6), while (2) other active processes still keep the value v_0. At any τ between the invocation and the end of $\mathsf{write}_1(v_1)$, we have $|W_i(\tau)| = x$ and $|W_0(\tau)| = |A(\tau)| - x$, where x is the number of processes that, at time τ, have updated to v_1 their local copy of the register. Due to the asynchrony of the system we cannot know when an active process replies an INQUIRY message. It can reply before receiving the WRITE message or after. If they all reply before, p_i will return the last value written before the concurrent write $\mathsf{write}_1(v)$. Otherwise, p_i will return a concurrently written value. Note that, in order to terminate (say at time τ_E), the $\mathsf{write}_1(v_1)$ operation needs to receive $n(1-c)$ ACK() messages (line 03, Figure 6). Hence, we have $|W_1(\tau_E)| \geq \frac{n}{2}$. Then, by iterating the above reasoning, we have that any join concurrent with $\mathsf{write}_1(v_1)$ will return either the last written value or the value concurrently written, and any subsequent join will return the last written value.

Then, the reasoning is the same as before for the subsequent write operations. It follows that a read obtains the last value written it there is no concurrent write operations, or the last value written before the read invocation, or a value written by a write operation concurrent with it. $\Box_{Theorem\ 4}$

6 Related Work

Assumption of correct processes in the system [24] and [28] describe protocols implementing distributed computing abstractions on top of a dynamic distributed system (leader election and connectivity, respectively) together with their correctness proofs. These protocols assume that a certain number of processes remain in the system for ”long enough” periods (as an example, [28] requires the perpetual presence of at least one process to maintain the system connectivity). The regular register protocols that have been presented do not require such an assumption (any process can be replaced at any time as allowed by the constant churn).

Registers implementation in mobile ad-hoc networks To the best of our knowledge the proposed regular register protocols are the firsts for distributed systems subject to churn. Other register protocols have been designed for mobile ad-hoc networks, e.g., [8]. Interestingly this protocol relies on a a form of deterministic broadcast, namely, geocast which is similar to our broadcast primitive in the sense that it ensures delivery to the processes that stay long enough in the system without leaving. Register protocols for MANET have been provided for synchronous systems.

7 Conclusion

This paper has addressed the construction of a regular register in synchronous and eventually synchronous message-passing distributed systems where processes can join and leave the system. Two protocols have been presented and proved correct, one for each type of system.

Several questions remain unanswered. One concerns the churn rate c. If there is no constraint on c, there is no protocol implementing a shared register (intuitively, this is because it is possible that no process remains long enough in the system to be able to implement the permanence/durability associated with a register). So, a fundamental question is the following: Is it possible to characterize the greatest value of c for a synchronous system (defined as a function involving the delay upper bound δ)? Moreover, has c to depend on n (the system size) in an asynchronous system? Another question concerns the implementation of quorums in dynamic systems[10] (quorums are required if one wants to permit any process: to write at any time). Finally, another important question is how to cope with the additional adversary that are process failures in a dynamic system?

References

[1] Abraham, I. and Malkhi, D., Probabilistic Quorum Systems for Dynamic Systems. *Distributed Computing*, 18(2):113-124, 2005.

[2] Aguilera M.K., A Pleasant Stroll Through the Land of Infinitely Many Creatures. *ACM SIGACT News, Distributed Computing Column*, 35(2):36-59, 2004.

[3] Attiya H., Bar-Noy A. and Dolev D., Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM*, 42(1):129-142, 1995.

[4] Baldoni R., Bertier M., Raynal M. and Tucci S., Looking for a Definition of Dynamic Distributed Systems. *Proc. 9th Int'l Conference on Parallel Computing Technologies (PaCT'07)*, Springer Verlag LNCS #4671, pp. 1-14, 2007.

[5] Bloom B., Constructing Two-writer Atomic Registers. *IEEE Transactions on Computers*, 37:1506-1514, 1988.

[6] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.

[7] Chaudhuri S., Kosa M.J. and Welch J., One-write Olgorithms for Multivalued Regular and Atomic Registers. *Acta Informatica*, 37:161-192, 2000.

[8] Dolev S., Gilbert S., Lynch N., Shvartsman A., and Welch J., Geoquorum: Implementing Atomic Memory in Ad hoc Networks. *Proc. 17th Int'l Symposium on Distributed Computing (DISC'03)*, Springer-Verlag LNCS #2848, pp. 306-320, 2003.

[9] Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288-323, April 1988.

[10] Dynamic quorums are addressed in a few papers, e.g., [1, 13, 17, 23, 26].

[10] Friedman R., Raynal M. and Travers C., Abstractions for Implementing Atomic Objects in Distributed Systems. *9th Int'l Conference on Principles of Distributed Systems (OPODIS'05)*, Springer Verlag LNCS #3974, pp. 73-87, 2005.

[11] Gafni E. and Lamport L., Disk Paxos. *Distributed Computing*, 16(1):1-20, 2003.

[12] Gafni E., Merritt M., and Taubenfeld G., The Concurrency Hierarchy, and Algorithms for Unbounded Concurrency. *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, pp. 161-16, 2001.

[13] Gramoli V. and Raynal M., Timed Quorum Systems for Large-Scale and Dynamic Environments. *Proc. 11th Int'l Conference on Principles of Distributed Systems (OPODIS'07)*, Springer-Verlag, LNCS # 4878, pp. 429-442, 2007.

[14] Guerraoui R. and Raynal M., The Alpha of Indulgent Consensus. *The Computer Journal*, 50(1):53-67, 2007.

[15] Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press, New-York, pp. 97-145, 1993.

[16] Haldar S. and Vidyasankar K., Constructing 1-writer Multireader Multivalued Atomic Variables from Regular Variables. *Journal of the ACM*, 42(1):186-203, 1995.

[17] Herlihy, M., Dynamic Quorum Adjustment for Partitioned Data. *ACM Transactions on Database Systems (TODS)*, 12(2):170-194, 1987.

[18] Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[19] Ko S., Hoque I. and Gupta I., Using Tractable and Realistic Churn Models to Analyze Quiescence Behavior of Distributed Protocols. *Proc. 27th IEEE Int'l Symposium on Reliable Distributed Systems (SRDS'08)*, IEEE Computer Press, pp. 259-268, 2008.

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

[21] Li M., Tromp J. and Vityani P., How to Share Concurrent Wait-free Variables. *Journal of the ACM*, 43(4):723-746, 1996.

[22] Liben-Nowell D., Balakrishnan H., and Karger D.R., Analysis of the Evolution of Peer-to-peer Systems. *Proc. 21th ACM Symposium on Principles of Distributed Computing (PODC'02)*, ACM press, pp 233-242, 2002.

[23] Lynch, N. and Shvartsman A., RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. *Proc. 16th Int'l Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 173-190, 2002.

[24] Mostefaoui A., Raynal M., Travers C., Peterson S., El Abbadi, Agrawal D., ¿From Static Distributed Systems to Dynamic Systems. *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, IEEE Computer Society Press, pp. 109-119, 2005.

[25] Merritt M. and Taubenfeld G., Computing with Infinitely Many Processes. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, Springer-Verlag LNCS #1914, pp. 164-178, 2000.

[26] Nadav U. and Naor M., The Dynamic And-or Quorum System. *Proc. 19th Int'l Symposium on Distributed Computing (DISC'05)*, Springer-Verlag LNCS #3724, pp. 472-486, 2005.

[27] Singh A.K., Anderson J.H. and Gouda M., The Elusive Atomic Register. *Journal of the ACM*, 41(2):331-334, 1994.

[28] Tucci Piergiovanni S. and Baldoni R., Connectivity in Eventually Quiescent Dynamic Distributed Systems, *Third Latin-American Symposium on Dependable Computing (LADC07)*, IEEE Press, pp. 38-56, 2007.

[29] Vidyasankar K., Converting Lamport's Regular Register to Atomic Register. *Information Processing Letters*, 28(6):287-290, 1988

[30] Vityani P. and Awerbuch B., Atomic Shared Register Access by Asynchronous Hardware. *Proc. 27th IEEE Symposium on Foundations of Computer Science (FOCS'87)*, IEEE Computer Press, 223-243, 1986.