Joining a Distributed Shared Memory Computation in a Dynamic Distributed System

Roberto Baldoni[1], Silvia Bonomi[1], and Michel Raynal[2]

[1] Universitá La Sapienza, Via Ariosto 25, I-00185 Roma, Italy,
[2] IRISA, Université de Rennes, Campus de Beaulieu, F-35042 Rennes, France
{baldoni, bonomi}@dis.uniroma1.it raynal@irisa.fr

Abstract. This paper is on the implementation of high level communication abstractions in dynamic systems (i.e., systems where the entities can enter and leave arbitrarily). Two abstractions are investigated, namely the read/write register and add/remove/get set data structure. The paper studies the join protocol that a process has to execute when it enters the system, in order to obtain a consistent copy of the (register or set) object despite the uncertainty created by the net effect of concurrency and dynamicity. It presents two join protocols, one for each abstraction, with provable guarantees.

Keywords: Churn, Dynamic system, Provable guarantee, Regular register, Set object, Synchronous system.

1 Introduction

Dynamic systems The passage from statically structured distributed systems to unstructured ones is now a reality. Smart environments, P2P systems and networked systems are examples of modern systems where the application processes are not aware of the current system composition. Because they are run on top of a dynamic distributed system, these applications have to accommodate a constantly change of the their membership (i.e., churn) as a natural ingredient of their life. As an extreme, an application can cease to run when no entity belongs to the membership, and can later have a membership formed by thousands of entities.

Considering the family of state-based applications, the main issue consists in maintaining their state despite membership changes. This means that a newcomer has to obtain a valid state of the application before joining it (state transfer operation). This is a critical operation as a too high churn may prevent the newcomer from obtain such a valid state. The shorter the time taken by the join procedure to transfer a state, the highest the churn rate the join protocol is able to cope with.

Join protocol with provable guarantees This paper studies the problem of joining a computation that implements a distributed shared memory on the top of a message-passing dynamic distributed system. The memory we consider is made up of the noteworthy object abstractions that are the regular registers and the sets. For each of them, a

notion of admissible value is defined. The aim of that notion is to give a precise meaning to the object value a process can obtain in presence of concurrency and dynamicity.

The paper proposes two join protocols (one for each object type) that provide the newcomer with an admissible value. To that end, the paper considers an underlying synchronous system where, while processes can enter and leave the application, their number remains always constant.

While the regular register object is a well-known shared memory abstraction introduced by Lamport [10], the notion of a set object in a distributed context is less familiar. The corresponding specification given in this paper extends the notion of weak set introduced by Delporte-Gallet and Fauconnier in [5].

Roadmap The paper is made up of 5 sections. First, Section 2 introduces the register and set objects (high level communication abstractions), and Section 3 presents the underlying computation model. Then, Sections 4 and 5 presents two join protocols, each suited to a specific object.

2 Distributed Shared Memory Paradigm

A distributed shared memory is a programming abstraction, built on top of a message passing system, that allows processes to communicate and exchange informations by invoking operations that return or modify the content of shared objects, thereby hiding the complexity of the message exchange needed to maintain it.

One of the simplest shared objects that can be considered is a register. Such an object provides the processes with two operations called read and write. Objects such as queues, stacks are more sophisticated objects.

It is assumed that every process it sequential: it invokes a new operation on an object only after receiving an answer from its previous object invocation. Moreover, we assume a global clock that is not accessible to the processes. This clock can be seen as measuring the real time as perceived by an external observer that would not part of the system.

2.1 Base Definitions

An operation issued on a shared object is not instantaneous: it takes time. Hence, two operations executed by two different processes, may overlap in time. Two events (denoted $invocation$ and the $response$) are associated with each operation. They occur at the beginning (invocation time) and at the end of the operation (return time).

Given two operation op and op' having respectively invocation times $t_B(op)$ and $t_B(op')$, and return times $t_E(op)$ and $t_E(op')$, respectively, we say that *op precedes* op' ($op \prec op'$) iff $t_E(op) < t_B(op')$. If op does not precede op' and op' does not precede op then they are *concurrent* ($op||op'$).

Definition 1 (Execution History). *Let H be the set of all the operations issued on a shared object $\mathcal{O}$. An execution history $\widehat{H} = (H, \prec)$ is a partial order on H satisfying the relation $\prec$.*

Definition 2 (Sub-history $\widehat{H_t}$ of $\widehat{H}$ at time t). *Given an execution history $\widehat{H} = (H, \prec)$ and a time t, the sub-history $\widehat{H_t} = (H_t, \prec_t)$ of $\widehat{H}$ at time t is the sub-set of $\widehat{H}$ such that: (i) $H_t \subseteq H$, and (ii) $\forall op \in H$ such that $t_B(op) \leq t$ then $op \in H_t$.*

2.2 Regular register: definition

A register object $\mathcal{R}$ has two operations. The operation $\mathsf{write}(v)$ defines the new value v of the register, while the operation $\mathsf{read}()$ returns a value from the register. The semantic of a register is given by specifying which are the values returned by its read operations. Without loss of generality, we consider that no two write operations write the same value.

This paper consider a variant of the regular register abstraction as defined by Lamport [10]. In our case, a *regular* register can have any number of writers and any number of readers [13]. The writes appear as if they were executed sequentially, this sequence complying with their real time occurrence 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 last value written in the register. Otherwise, the read operation returns any value written by a concurrent write operation or the last value of the register before these concurrent writes.

Definition 3 (Admissible value for a $\mathsf{read}()$ operation). *Given a $\mathsf{read}()$ operation op, a value v is* admissible *for op if:*

- $\exists\, \mathsf{write}(v) : \mathsf{write}(v) \prec op \ \vee \mathsf{write}(v) \mid\mid op$, and
- $\nexists\, \mathsf{write}(v')\ (with\ v \neq v') : \mathsf{write}(v) \prec \mathsf{write}(v') \prec op$.

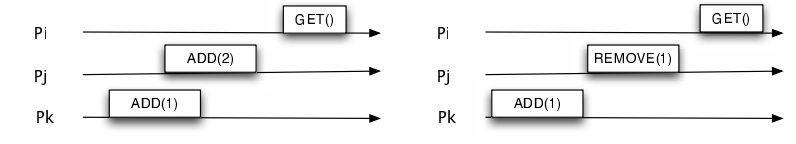
Definition 4 (Admissible value for a regular register at time t). *Given an execution history $\widehat{H} = (H, \prec)$ of a regular register $\mathcal{R}$ and a time t, let $\widehat{H_t} = (H_t, \prec_t)$ be the sub-history of $\widehat{H}$ at time t. An* admissible value at time t for $\mathcal{R}$ *is any possible admissible value v for an instantaneous read operation op executed at time t.*

2.3 Set data structure: definition

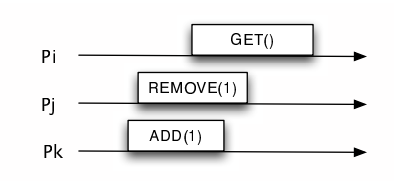
A set object $\mathcal{S}$ can be accessed by processes by means of three operations: $\mathsf{add}()$ and $\mathsf{remove}()$ that modify the content of the set and $\mathsf{get}()$ that returns the current content of the set.

The $\mathsf{add}(v)$ operation takes an input a parameter v and returns the value ok when it terminates. Its aim is to add the element v to $\mathcal{S}$. Hence, if $\{x_1, x_2, \ldots, x_k\}$ are the values belonging to $\mathcal{S}$ before the invocation of $\mathsf{add}(v)$, and if no $\mathsf{remove}(v)$ operation is executed concurrently, the value of the set will be $\{x_1, x_2, \ldots, x_k, v\}$ after its invocation.

The $\mathsf{remove}(v)$ operation takes an input a parameter v and returns the value ok. Its aim is to delete the element v from $\mathcal{S}$ if v belongs to $\mathcal{S}$, otherwise the remove operation has no effect.



(a) $V_{seq} = \{1, 2\}$, $V_{conc} = \emptyset$ (b) $V_{seq} = \emptyset$, $V_{conc} = \emptyset$



(c) $V_{seq} = \emptyset$, $V_{conc} = \{1\}$

Fig. 1. V_{seq} and V_{conc} in distinct executions

The get() operation takes no input parameter. It returns a set containing the current content of $\mathcal{S}$, without modifying the content of the object. In a concurrency-free context, every get() operation returns the current content of the set. The content of the set is well-defined when the operations occur sequentially. In order to state without ambiguity the value returned by get() operation in a concurrency context, let us introduce the notion of *admissible values* for a get() operation op (i.e. $V_{ad}(op)$) by defining two sets, denoted *sequential set* ($V_{seq}(op)$) and *concurrent set* ($V_{conc}(op)$).

Definition 5 (Sequential set for a get() **operation).** *Given a* get() *operation* op *executed on* $\mathcal{S}$*, the set of* sequential values *for* op *is a set (denoted* $V_{seq}(op)$*) that contains all the values* v *such that:*

1. $\exists\, \mathsf{add}(v) : \mathsf{add}(v) \prec op$, and
2. *If* $\mathsf{remove}(v)$ *exists, then* $\mathsf{add}(v) \prec op \prec \mathsf{remove}(v)$.

As an example, let consider Figure 1(a). The sequential set $V_{seq}(op) = \{1, 2\}$ because there exist two operations adding the values 1 and 2, respectively, that terminate before the get() operation starts, and there is neither $\mathsf{remove}(1)$, nor $\mathsf{remove}(2)$, before get() termionates. Differently, $V_{seq}(op) = \emptyset$ in Figure 1(b).

Definition 6 (Concurrent set for a get() **operation).** *Given a* get() *operation* op *executed on* $\mathcal{S}$*, the set of* concurrent values *for the* get() *operation is a set (denoted* $V_{conc}(op)$*) that contains all the value* v *such that:*

1. $\exists\, \mathsf{add}(v) : \mathsf{add}(v) \mid\mid op$, or
2. $\exists\, \mathsf{add}(v),\ \mathsf{remove}(v) : (\mathsf{add}(v) \prec op) \wedge (\mathsf{remove}(v) \mid\mid op)$, or
3. $\exists\, \mathsf{add}(v),\ \mathsf{remove}(v) : \mathsf{add}(v) \mid\mid \mathsf{remove}(v) \wedge \mathsf{add}(v) \prec op \wedge \mathsf{remove}(v) \prec op$.

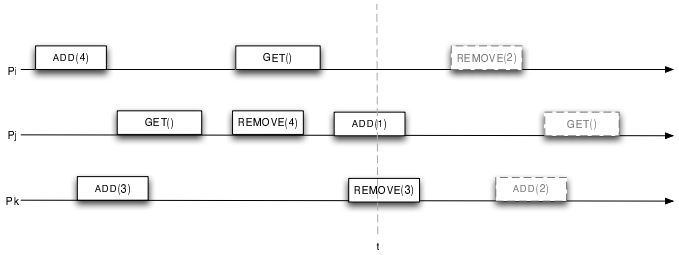


Fig. 2. Sub-History $\widehat{H}_t$ at time t

When considering the execution of Figure 1(c), $V_{conc}(op) = \{1\}$ due to the first item 1 of the previous definition.

Definition 7 (Admissible set for a get() **operation).** *Given a* get() *operation* op*, a sequential set* $V_{seq}(op)$ *and a concurrent set* $V_{conc}(op)$*, a set* $V_{ad}(op)$ *is an* admissible set *of values for* op *if* $\big(V_{seq}(op) \subseteq V_{ad}(op)\big) \wedge \big(\big(\big(V_{ad}(op) \setminus V_{seq}(op)\big) \subseteq V_{conc}(op)\big)\big)$.

As an example, let consider the executions depicted in Figure 1. For Figure 1(a) and Figure 1(b), there exists only one admissible set V_{ad} for the get operation and it is respectively $V_{ad}(op) = \{1, 2\}$ and $V_{ad}(op) = \emptyset$. Differently, for Figure 1(c) there exist two different admissible sets for the get operation, namely, $V_{ad}(op) = \emptyset$ and $V_{ad}(op) = \{1\}$. Note that, in the executions depicted in Figure 1(c), if there was another get() operation after the add() and the remove() operations, these two get() could return different admissible sets.

In order to take into consideration such point, consistency criteria have to be defined.

Definition 8 (Admissible Sets of values at time t**).** *An* admissible set of values at time t for $\mathcal{S}$ *(denoted* $\mathcal{V}_{ad}(t)$*) is any possible admissible set* $V_{ad}(op)$ *for any* get() *operation* op *that would occur instantaneously at time* t*.*

As an example, consider the scenario depicted in Figure 2. The sub-history at the time t is the partial order of all the operations started before t (i.e. the operations belonging to the set H_t are add(4) and get() executed by p_i, get(), remove(4) and add(1) executed by p_j, and add(3) and remove(3) executed by p_k.

The instantaneous get operation op is concurrent with add(1) executed by p_j and remove(3) executed by p_k. The sequential set for op $V_{seq}(op)$ is $\emptyset$ because for both the add operations preceding op there exists a remove not following op while the concurrent set for op $V_{conc}(op)$ is $\{1, 3\}$. The possible admissible sets for op (and then the possible admissible sets at time t) could be then (i) $\emptyset$, (ii) $\{1\}$, (iii) $\{3\}$ and (iv) $\{1, 3\}$.

3 Joining a Computation in Dynamic Distributed System

3.1 System Model

The distributed system is composed, at each time, by a fixed number (n) of processes that communicate by exchanging messages. Processes are uniquely identified with their indexes and they may join and leave the system at any point in time.

The system is synchronous in the following sense. The processing times of local computations 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, but their transmission time is upper bounded. Moreover, we assume that processes can access a global clock (this is for ease of presentation; as we are in a synchronous system, such a global clock could be implemented by synchronized local clocks). We assume that there exists an underling protocol (implemented at the connectivity layer) that keeps processes connected.

3.2 The Problem

Given a shared object $\mathcal{O}$ (e.g. a register or a set), it is possible to associate with it, at each time t, a set of admissible values. Processes continuously join the system along time and every process p_i that enters the computation has no information about the current state of the object with the consequence of being unable to perform any operation. Therefore every process p_i that wishes to enter into the computation needs to retrieve an admissible value for the object $\mathcal{O}$ from the other processes.

This problem is captured by adding a $\mathsf{join}()$ operation that has to be invoked by every joining process. This operation is implemented by a distributed protocol that builds an admissible value for the object.

3.3 Distributed Computation

A distributed computation is defined, at each time, by a subset of processes. A process p, belonging to the system, that wants to participate to the distributed computation has to execute the $\mathsf{join}()$ operation. Such an operation, invoked at some time t, is not instantaneous. But, from time t, the process p can receive and process messages sent by any other process that belongs to the system and that participate to the computation. Processes participating to the computation implements a shared object. A process leaves the computation in an implicit way. When it does, it leaves the computation forever and does not longer send 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.)

We assume that no process crashes during the computation (i.e., it does not crash from the time it joins the system until it leaves).

In order to formalize the set of processes that participate actively to the computation we give the following definition.

Definition 9. *A process is* active *from the time it returns from the* $\mathsf{join}()$ *operation until the time it leaves the system.* $A(t)$ *denotes the set of processes that are active at time* t*, while* $A([t_1, t_2])$ *denotes the set of processes that are active during the interval* $[t_1, t_2]$*.*

3.4 Communication Primitives

Two communication primitives are used by processes belonging to the distributed computation to communicate: point-to-point and broadcast communication.

Point-to-point communication This primitive allows a process p_i to send a message to another process p_j as soon as p_i knows that p_j has joined the computation. 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 t, then p_j receives that message by time $t + \delta'$ (if it has not left the system by that time). In that case, the message is said to be "sent" and "received".

Broadcast Processes participating to the distributed computation are equipped with an appropriate broadcast communication sub-system 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 distributed 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 t be the time at which a process p belonging to the computation 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 $t + \delta$, then all the processes that are in the system at time t and do not leave by time $t + \delta$, deliver m by time $t + \delta$.

Such a pair of broadcast operations has first been formalized in [8] in the context of systems where process can commit crash failures. It has been extended to the context of dynamic systems in [7].

3.5 Churn Model

The phenomenon of continuous arrival and departure of nodes in the system is usually referred as *churn*. In this paper, the churn of the system is modeled by means of the join distribution $\lambda(t)$, the leave distribution $\mu(t)$ and the node distribution $N(t)$ [3]. The join and the leave distribution are discrete functions of the time that returns, for any time t, respectively the number of processes that have invoked the join operation at time t and the number of processes that have left the system at time t. The node distribution returns, for every time t, the number of processes inside the system. We assume, at the beginning, n_0 processes inside the system and we assume to have $\lambda(t) = \mu(t) = cn_0$ (where $c \in [0, 1]$ is a percentage of node of the system) meaning that at each time unit, the number of precess that joins the system is the same as the number of process that leave, i.e. the number of processes inside the system $N(t)$ is always equal to n_0.

4 Joining a Register Computation

4.1 The Protocol

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).

The local variables of each process p_k (of the n processes that compose the initial set of processes) are such that $register_k$ contains the initial value of the regular register (say the value 0), $sn_k = 0$, $active_k = true$, and $replies_k = reply\_to_k = \emptyset$.

```
operation join(i):
(01) register_i ← ⊥; sn_i ← −1; active_i ← false; replies_i ← ∅; reply_to_i ← ∅;
(02) wait(δ);
(03) if (register_i = ⊥) then
(04)   replies_i ← ∅; broadcast INQUIRY(i); wait(2δ);
(05)   let < id, val, sn > ∈ replies_i such that (∀ < −, −, sn' > ∈ replies_i : sn ≥ sn');
(06)   if (sn > sn_i) then sn_i ← sn; register_i ← val end if
(07) end if;
(08) active_i ← true;
(09) for each j ∈ reply_to_i do send REPLY (< i, register_i, sn_i >) to p_j end for;
(10) return(ok).
```

```
(11) when INQUIRY(j) is delivered:
(12)       if (active_i) then send REPLY (< i, register_i, sn_i >) to p_j
(13)                     else reply_to_i ← reply_to_i ∪ {j}
(14)       end if.

(15) when REPLY(< j, value, sn >) is received:
                                  replies_i ← replies_i ∪ {< j, value, sn >}.
```

Fig. 3. The join() protocol for a register object in a synchronous system (code for p_i)

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 3, involves all the processes that are currently present (be them active or not). The interested reader will find a proof in [3].

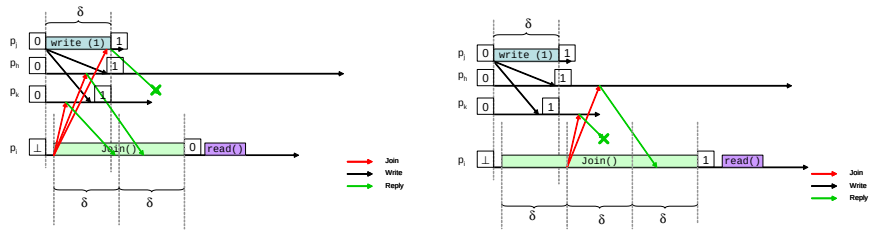
First p_i initializes its local variables (line 01), and waits for a period of δ time units (line 02); This waiting period is explained later. If $register_i$ has not been updated during this waiting period (line 03), p_i broadcasts (with the broadcast() operation) an INQUIRY(i) message to the processes that are in the system (line 04) and waits for 2δ time units, i.e., the maximum round trip delay (line 04)[3]. When this period terminates,

[3] 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

p_i updates its local variables $register_i$ and sn_i to the most uptodate values it has received (lines 05-06). Then, p_i becomes active (line 08), which means that it can answer the inquiries it has received from other processes, and does it if $reply\_to \neq \emptyset$ (line 09). Finally, p_i returns ok to indicate the end of the join() operation (line 10).

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 12). Otherwise, p_i postpones its answer until it becomes active (line 13 and lines 08-09). 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 15).

Why the wait(δ) *statement at line 02 of the* join() *operation?* To motivate the wait(δ) statement at line 02, let us consider the execution of the join() operation depicted in Figure 4(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 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



(a) Without wait(δ) (b) With wait(δ)

Fig. 4. Why wait(δ) is required

broadcast invoked 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 wait(δ) statement at line 02, its execution of the lines 03-07 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 4(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.

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.

5 Joining a Set Computation

5.1 The Protocol

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

- Two variables denoted set_i and sn_i; set_i is a set variable and contains the local copy of the set, while sn_i is an integer variable that count how many update operations have been executed by process p_i on the local copy of the set.
- A FIFO set variable $last_{ops_i}$ used to maintain an history of the update operations executed by p_i. Such variable contains all the 3-uples $< val, op\_type, id >$ each one characterizing an operation of type $op\_type = \{\mathsf{add}$ or $\mathsf{remove}\}$ of the value val issued by a process with identity id.
- A boolean $active_i$, initialized to $false$, that is switched to $true$ just after p_i has joined the system.
- Three set variables, denoted $replies_i$, $reply\_to_i$ and $pending_i$,that are used in the period during which p_i joins the system. The local variable $replies_i$ contains the 3-uples $< set, sn, ops >$ 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). The set $pending_i$ contains the 3-uples $< val, op\_type, id >$ each one characterizes an update operation executed concurrently with the join.

Initially, n processes compose the system. The local variables of each of these processes p_k are such that set_k contains the initial value of the regular register (without loss of generality, we assume that, at the beginning, every process p_k has nothing in its variable set_k), $sn_k = 0$, $active_k = true$, and $pending_k = replies_k = reply\_to_k = \emptyset$.

The $\mathsf{join}()$ *operation* The algorithm implementing the join operation for a set object, is described in Figure 5, and 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); the motivations for such waiting period are basically the same described for the regular register and it is needed to avoid that p_i looses some update. After this waiting period, p_i broadcasts (with the $\mathsf{broadcast}()$ operation) an INQUIRY(i) message to the processes that are in the system and waits for 2δ time units, i.e., the maximum round trip delay (line 02). When this period terminates, p_i first updates its local variables set_i, sn_i and $last_{ops_i}$ to the most uptodate values it has received (lines 03-04) and then executes all the operations concurrent with the join contained in $pending_i$ and not yet executed (lines 05-13). Then, p_i becomes active (line 14), which means that it can answer the inquiries it has received from other processes, and does it if $reply\_to \neq \emptyset$ (line 15). Finally, p_i returns ok to indicate the end of the $\mathsf{join}()$ operation (line 16).
When a process p_i receives a message INQUIRY(j), it answers p_j by sending back a REPLY$(< set_i, sn_i, last_{ops_i} >)$ message containing its local variables if it is active (line 18). Otherwise, p_i postpones its answer until it becomes active (line 19 and line 15). Finally, when p_i receives a message REPLY$(< set, sn, ops >)$ from a process p_j it adds the corresponding 3-uple to its set $replies_i$ (line 21).

```
operation join(i):
(01) sn_i ← 0; last_ops_i ← ∅ set_i ← ∅; active_i ← false;
     pending_i ← ∅; replies_i ← ∅; reply_to_i ← ∅;
(02) wait(δ); broadcast INQUIRY(i); wait(2δ);
(03) let < set, sn, ls >∈ replies_i such that (∀ < −, sn', − >∈ replies_i : sn ≥ sn');
(04) set_i ← set; sn_i ← sn; last_op_i ← ls;
(05) for each < val, op_type, id >∈ pending_i do
(06)     if (< val, op_type, id >∉ last_op_i) then
(07)                                   sn_i ← sn_i + 1;
(08)                                   last_op_i ← last_op_i ∪ {< val, op_type, id >};
(09)                                   if (op_type = add) then set_i ← set_i ∪ {val};
(10)                                                      else set_i ← set_i/{val};
(11)                                   end if
(12)     end if
(13) end for;
(14) active_i ← true;
(15) for each j ∈ reply_to_i do send REPLY (< set_i, sn_i, last_op_i >) to p_j end for;
(16) return(ok).
-------------------------------------------------------------------------------
(17) when INQUIRY(j) is delivered:
(18)       if (active_i) then send REPLY (< set_i, sn_i, last_op_i >) to p_j
(19)                     else reply_to_i ← reply_to_i ∪ {j}
(20)       end if.

(21) when REPLY(< set, sn, ops >) is received:
                                   replies_i ← replies_i ∪ {< set, sn, ops >}.
```

Fig. 5. The join() protocol for a set object in a synchronous system (code for p_i)

5.2 add() and remove() protocols

These protocols are trivially executed by sending an update message using the broadcast primitives (i.e. their execution time is bounded by δ). At the receipt of the update message, every process p_i checks its state. If p_i is active, it simply adds or removes the value from its local copy of the set. If p_i is not active (i.e. it is still executing the join() protocol), it buffers the operation in the local set $pending_i$ set by adding the 3-uple $< val, op\_type, id >$. Such a tuple is made up of (i) the value val to be updated, (ii) the type op_type of the operation (add or remove), and (iii) the id of the process that issued the update. Every operation in the set $pending_i$ will be then executed by p_i at the end of the join() protocol (lines 05-13 of Figure 5).

5.3 Correctness proof

Due to page limitation, this section only states two lemmas and the main theorem. Their proofs can be found in [4].

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

Lemma 2. *Let t_0 be the time at which the computation of a set object $\mathcal{S}$ starts, $\widehat{H} = (H, \prec)$ an execution history of $\mathcal{S}$, and $\widehat{H}_{t_1+3\delta} = (H_{t_1+3\delta}, \prec)$ the sub-history of $\widehat{H}$ at time $t_1+3\delta$. Let p_i be a process that invokes* join() *on $\mathcal{S}$ at time $t_1 = t_0+1$, if $c < 1/3\delta$ then at time $t_1 + 3\delta$ the local copy set_i of $\mathcal{S}$ maintained by p_i will be an admissible set at time $t_1 + 3\delta$.*

Theorem 1. *Let $\widehat{H} = (H, \prec)$ be the execution history of a set object $\mathcal{S}$, and p_i a process that invokes* join() *on the set $\mathcal{S}$ at time t. If $c < 1/3\delta$ then at time $t + 3\delta$ the local copy set_i of $\mathcal{S}$ maintained by p_i will be an admissible set at time $t + 3\delta$.*

References

1. Aguilera M.K., A Pleasant Stroll Through the Land of Infinitely Many Creatures. *ACM SIGACT News, Distributed Computing Column*, 35(2):36-59, 2004.
2. Baldoni R., Bonomi S., Kermarrec A.M., Raynal M., Implementing a Register in a Dynamic Distributed System. In *Proc. 29th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'09)*, IEEE Computer Society Press, pp. 639-647 2009.
3. Baldoni R., Bonomi S., Raynal M. Regular Register: an Implementation in a Churn Prone Environment. In *16th International Colloquium on Structural Information and Communication Complexity (SIROCCO'09)* Springer-Verlag, LNCS, 2009.
4. Baldoni R., Bonomi S., Raynal M. Joining a Distributed Shared Memory Computation in a Dynamic Distributed System. *Tech Report 5/09*, MIDLAB, Universitá di Roma, La Sapienza, (Italy), July 2009. http://www.dis.uniroma1.it/ midlab/publications.
5. Delporte-Gallet C., Fauconnier H. Two Consensus Algorithms with Atomic Registers and Failure Detector Ω. *In 10th International Conference on Distributed Computing and Networking (ICDCN'09)* Springer-Verlag, LNCS #5408, pp 251-262, 2009.
6. 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 (DISC03)*, Springer-Verlag LNCS #2848, pp. 306-320, 2003.
7. 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)*, LNCS #3974, pp. 73-87, 2005.
8. Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems*, ACM Press, New-York, pp. 97-145, 1993.
9. 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)*, 2008.
10. Lamport. L., On Interprocess Communication, Part 1: Models, Part 2: Algorithms. *Distributed Computing*, 1(2):77-101, 1986.
11. Leonard D., Yao Z., Rai V. and Loguinov D., On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks. *IEEE/ACM Transaction on Networking* 15(3), 644-656 (2007)
12. Merritt M. and Taubenfeld G., Computing with Infinitely Many Processes. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, LNCS #1914, pp. 164-178, 2000.
13. Shao C., Pierce E. and Welch J., Multi-writer consistency conditions for shared memory objects. *Proc. 17th Int'l Symposium on Distributed Computing (DISC'03)*, Springer-Verlag, LNCS #2848, pp.106-120, 2003.