Value-based Sequential Consistency for Set Objects in Dynamic Distributed Systems

Roberto BALDONI[†] Silvia BONOMI[†] Michel RAYNAL[‡]

[†] Universitá La Sapienza, Roma, Italy
[‡] IRISA, Université de Rennes, Campus de Beaulieu, Rennes, France

Abstract

This paper introduces a set object, namely a shared object that allows processes to add and remove values as well as take a snapshot of its content. A new consistency condition suited to such an object is introduced. This condition, named value-based sequential consistency, is weaker than linearizability. The paper addresses also the construction of a set object in a synchronous anonymous distributed system where participants can continuously join and leave the system. Interestingly, the protocol is proved correct under the assumption that some constraint on the churn is satisfied. This shows that the notion of "provably correct software" can be applied to dynamic systems.

Keywords: *Churn, Consistency condition, Dynamic system, Infinite arrival model, Set object, Synchronous system.*

1 Introduction

A set $\mathcal{S}$ is a shared object that stores a (possibly empty) finite set of values. A process can acquire the content of $\mathcal{S}$ through a *get* operation while it can add (remove) an element to $\mathcal{S}$ through an *add* (*remove*) operation. A restricted form of set, namely *weak set*, has been introduced for the first time by Delporte-Gallet and Fauconnier in [9]. A weak set is a set without the *remove* operation. Delporte-Gallet and Fauconnier point out that (due to the semantic of the object itself) a weak set object is not linearizable [14]. More precisely, a *get* operation does not care about the execution order of two concurrent *add* operations that happened before the *get* because the important issue for a *get* operation is the fact that a value is (or is not) in the weak set (and not the order in which values have been inserted into the set). The authors show that a weak set is a useful abstraction to solve consensus in anonymous shared memory systems.

Contribution of the paper The paper presents a set object that extends the notion of weak set proposed in [9]. A set has operations for joining the computation (i.e. *join* operation), remove a value from the set (i.e. *remove* operation) as well as *get* and *add* operations defined in the weak set. The paper has two main contributions.

- It first introduces a consistency condition for a set object. This new condition is named *value-based sequential consistency*. While it allows concurrent *get* operations to return the same output in the absence of concurrent *add* and/or *remove* operation, this condition is weaker than linearizability [14].

This is because processes are required to see the same order only of concurrent *add* and *remove* operations that are on a same value. Concurrent operations executed on distinct values can be perceived in any order by a process.

- The second contribution is a protocol that implmeentsd a set $\mathcal{S}$ on the top of a dynamic anonymous message-passing synchronous message-passing distributed system. The implementation uses a copy of $\mathcal{S}$ at any process. An important part of that contribution is the proof that the implementation is correct when the churn reamains below a given threshold. For the churn we use the characterization given in [5] in which the number of processes in the system is always constant (this means at any time the same number of processes join and leave the system).

Roadmap Section 2 presents the set objects, details the set operations and introduces the value-based sequential consistency. The distributed system model and the model of churn is presented in Section 3. A protocol implementing the set objects is introduced in Section 4. In the same section we prove that the protocol satisfies the value-based sequential consistency. Section 5 presents the related work and section 6 concludes the paper.

2 The Set Object

A set object $\mathcal{S}$ is a shared object used to store values. Without loss of generality, we assume that (i) $\mathcal{S}$ contains only integer values and (ii) at the beginning of the computation $\mathcal{S}$ is empty. A set $\mathcal{S}$ can be accessed by three operations: *add* and *remove* that modify the content of the set and *get* that returns the current content of the set. More precisely:

- The *add* operation, denoted $\mathsf{add}(v)$, takes an input parameter v and returns a confirmation that the operation has been executed (i.e. the value OK). It adds v to $\mathcal{S}$. If v is already in the set, the *add* operation has no effect.

- The *remove* operation, denoted $\mathsf{remove}(v)$, takes an input parameter v and returns a confirmation that the operation has been executed (i.e. the value OK). If v belongs to $\mathcal{S}$, it suppresses it from $\mathcal{S}$. Otherwise it has no effect.

- The *get* operation, denoted $\mathsf{get}()$, takes no input parameter. It returns a set containing the current value of $\mathcal{S}$. It does not modifies the content of the object.

Generally, each of these operation is not instantaneous and takes time to be executed; we assume that every process executes operations sequentially (i;e., a process does not invoke any operation before it got a response from the previous one). Hence, given two operations executed by two different processes, they may overlap and the current content of the set may be not univocally well defined. Consider, for example, a $\mathsf{get}()$ operation overlapping with an $\mathsf{add}(v)$ operation while v is not present in the set: has v to be contained in the result of the $\mathsf{get}()$? Moreover, if two processes modify concurrently the set by adding and removing the same value v, has v yo be returned by a successive $\mathsf{get}()$ operation or not?

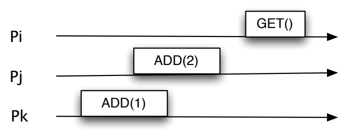
In the following section the notion of concurrency between operations is defined and the behavior of the get operation in case of concurrency is specified.

2.1 Basic Definitions

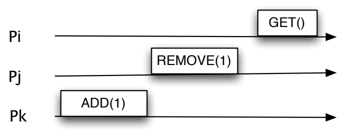
Every operation can be characterized by two events occurring at its boundary: an $invocation$ event and a $reply$ event. These events occur at two time instants (invocation time and reply time). According to these time instants, it is possible to state when two operations are concurrent with respect to the real time execution. For ease of presentation we assume the existence of a fictional global clock (not accessible by processes). The invocation time and response time of every operation are defined with respect to that clock.

Given two operation op and op' having respectively invocation times $t_B(op)$ and $t_B(op')$ and return times $t_E(op)$ and $t_E(op')$, 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* ($op || op'$).

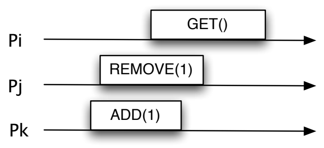
By definition, every $\mathsf{get}()$ operation, issued on a set object $\mathcal{S}$, should return the current content of $\mathcal{S}$. If operations occur sequentially, such a content is represented by all the values added by an $\mathsf{add}()$ operation preceding the $\mathsf{get}()$ and for which there does not exist a $\mathsf{remove}()$ that precedes $\mathsf{get}()$. Conversely, if there is concurrency between operations, the values added or removed concurrently may belong to the set. In order to formalize such a behavior, we introduce the notion of *admissible sets* for an $op = \mathsf{get}()$ operation (denoted $V_{ad}(op)$). To that end we first define two sets, namely a *sequential set* ($V_{seq}(op)$) and a *concurrent set* ($V_{conc}(op)$) for an $op = \mathsf{get}()$ operation. These sets define admissible values, with respect to a $\mathsf{get}()$ operation, in case of sequential access and concurrent access to $\mathcal{S}$.



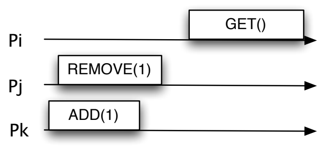
(a) $V_{seq} = \{1, 2\}$ and $V_{conc} = \emptyset$



(b) $V_{seq} = \emptyset$ and $V_{conc} = \{\emptyset\}$



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



(d) $V_{seq} = \emptyset$ and $V_{conc} = \{1\}$

Figure 1: V_{seq} and V_{conc} in distinct executions.

Definition 1 (Sequential set for a $\mathsf{get}()$ operation) *Given an $op = \mathsf{get}()$ operation executed on a set object $\mathcal{S}$, the set $V_{seq}(op)$ of sequential values for op contains all the values v such that:*

1. $\exists\ \mathsf{add}(v) : \mathsf{add}(v) \prec op$ *and,*
2. *(a)* $\nexists\ \mathsf{remove}(v)$ or
   (b) $\exists \mathsf{remove}(v) : (\mathsf{remove}(v) \prec \mathsf{add}(v) \prec op) \vee (\mathsf{add}(v) \prec op \prec \mathsf{remove}(v))$.

Informally, given a get() operation, a sequential set will include all the values that have to be returned by such a get (i.e. all the values for which there exists a terminated add operation and there not exists any remove between the add and the get). As an example, let consider the execution of Figure 1(a) and let *op* be the $\mathsf{get}()$ operation represented in Figure 1(a), the sequential set $V_{seq}(op)$ is equal to $\{1, 2\}$ because there exist two $\mathsf{add}()$ operations, adding values 1 and 2 respectively, that terminate before the $\mathsf{get}()$ operation is issued and does not exist any $\mathsf{remove}()$ operation starting before the $\mathsf{get}()$. Conversely, in the execution of Figure 1(b) $V_{seq}(op) = \emptyset$ because the only value added to the set is subsequently removed before the $\mathsf{get}()$ operation *op* is issued.

Definition 2 (Concurrent set for a $\mathsf{get}()$ operation) *Given an $op = \mathsf{get}()$ operation executed on a set object $\mathcal{S}$, the set $V_{conc}(op)$ of concurrent values for op contains all the values 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. $\exists\ \mathsf{add}(v),\ \mathsf{remove}(v) : \mathsf{add}(v) \mid\mid \mathsf{remove(v)} \wedge \mathsf{add}(v) \prec op \wedge \mathsf{remove}(v) \prec op.$

Informally, given a get() operation, a concurrent set will include all the values that may be returned by such a get (i.e. all the values for which the add or the remove operation is executed concurrently with the get or by themself). As an example, let consider the execution of Figure 1(c) and let op be the $\mathsf{get}()$ operation represented in that figure, the concurrent set $V_{conc}(op)$ is equal to $\{1\}$ since item 1 of the definition is satisfied, while, in the execution of Figure 1(d), $V_{conc}(op) = \{1\}$ due to item 3.

Now, it is possible to define an admissible set of values for a $\mathsf{get}()$ operation.

Definition 3 (Admissible set for a $\mathsf{get}()$ operation) *Given an $op = \mathsf{get}()$ operation issued on a set object $\mathcal{S}$, its sequential set $V_{seq}(op)$ and its concurrent set op $V_{conc}(op)$, its* admissible set *$V_{ad}(op)$ is such that*

1. *$V_{ad}(op)$ contains at least the values in op $V_{seq}(op)$,*
2. *and, $\forall v \in V_{ad}(op)/V_{seq}(op)$, we have $v \in V_{conc}(op)$.*

As an example, let consider the four executions depicted in Figure 1. In Figure 1(a) and Figure 1(b), there exists only one admissible set $V_{ad}(op)$ for each of the $\mathsf{get}()$ operations op and is respectively $V_{ad}(op) = \{1, 2\}$ for the execution of Figure 1(a) and $V_{ad}(op) = \emptyset$ for the execution of Figure 1(b). Contrarily, in the executions of Figure 1(c) and Figure 1(d) there exist two different admissible sets for each of the $\mathsf{get}()$ operations. In particular these admissible sets (equal for both the executions) are $V_{ad}(op) = \emptyset$ and $V_{ad}(op) = \{1\}$; the first one contains only the element contained in $V_{seq}(op)$ while the second one contains also the elements of $V_{conc}(op)$.

Note that in the executions depicted in Figure 1(c) and Figure 1(d) if another $\mathsf{get}()$ operation is issued after the $\mathsf{add}()$ and $\mathsf{remove}()$ operations, the $\mathsf{get}()$ may return different admissible sets.

2.2 Value-based Sequential Consistency Condition

A consistency condition defines which are the values that a $\mathsf{get}()$ operation is allowed to return. In a shared memory context, a set of formal consistency conditions has been defined [21] as constraints on the partial order of $\mathsf{read}()$ and $\mathsf{write}()$ operations issued on the shared memory. In order to specify a condition for a set object, we introduce the concepts of $execution\ history$, $legal\ get$ and $linear\ extension\ of\ an\ history$.

Definition 4 (Execution History) *Let H be the set of all the operations issued on the set object $\mathcal{S}$. An execution history $\hat{H} = (H, \prec)$ is a partial order on H satisfying the relation $\prec$.*

Definition 5 (Legal get()) *Let V be set returned by an $op = \mathsf{get}()$ operation. This operation is legal if V is an admissible set for op.*

Definition 6 (Linear extension of an history) *A linear extension $\hat{S} = (S, \rightarrow_s)$ of an execution history $\hat{H}$ is a topological sort of its partial order where (i) $S = H$, (ii) $op_1 \prec op_2 \Rightarrow op_1 \rightarrow_s op_2$ and (iii) $\rightarrow_s$ is a total order.*

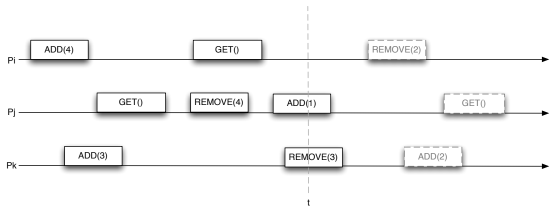


Figure 2: Sub-History $\hat{H}_t$ at time t of the History $\hat{H}$.

Let us introduce now the notion of *value-based sequential consistency* for a set object. Informally, this consistency condition requires that any group of concurrent $\mathsf{get}()$ operations that do not overlap with any other operation, return the same set. Moreover, due to the semantic of the set when considering *concurrent operations involving different values* (e.g. $\mathsf{add}(v)$ and $\mathsf{add}(v')$), these operations can be perceived in different order by different processes. More formally we have th following.

Definition 7 (Value-based sequential consistency) *A history $\hat{H} = (H, \prec)$ is* value-based sequentially consistent *iff for each process p_i there exists a linear extension $\hat{S}_i = (S, \rightarrow_{s_i})$ such that for any pair of concurrent operations op=*add*(v) and op'=*remove*(v'), with $v = v'$, if op $\rightarrow_{s_i}$ op' for some p_i then op $\rightarrow_{s_j}$ op' for any other process p_j.*

Note that, if the domain of values that can be written in the set is composed of a single value, then value-based sequential consistency is equivalent to sequential consistency[1] [16]. In fact, in this case, any pair of concurrent operations occurs on the same value and thus have to be ordered the same way by all the processes. Since the non-concurrent operations are totally ordered, the result is a unique total order on which all the processes agree.

Let now consider the following case: each process can add and/or remove only one specific value (e.g., its identifier). Value-based sequential consistency boils down to to causal consistency[2] [2] . Since each value is associated with a only one one process and each process executes operations sequentially, it follows that the concurrent operations are issued on different values and each process can perceive them in a different order, exactly as in causal consistency.

2.3 Admissible set at time t

Definition 8 (Sub-history $\hat{H}_t$ of $\hat{H}$ at time t) *Given an execution history $\hat{H} = (H, \prec)$ and a time t, the sub-history $\hat{H}_t = (H_t, \prec)$ of $\hat{H}$ at time t is the sub-set of $\hat{H}$ such that:*

- *$H_t \subseteq H$,*
- *$\forall op \in H$ such that $t_B(op) \leq t$ then $op \in H_t$.*

As an example, consider the history $\hat{H}$ depicted in Figure 2. The sub-history $\hat{H}_t$ at the time t is the partial order of all the operations started before t (i.e. H_t contains $\mathsf{add}(4)$ and $\mathsf{get}()$ invoked by p_i, $\mathsf{get}()$, $\mathsf{remove}(4)$ and $\mathsf{add}(1)$ invoked by p_j and $\mathsf{add}(3)$ and $\mathsf{remove}(3)$ invoked by p_k).

[1]A history $\hat{H} = (H, \prec)$ is sequential consistent if it admits a linear extension in which all the $\mathsf{get}()$ operations are legal.

[2]Let $\hat{H}^i$ be the sub-history of $\hat{H}$ from which all $\mathsf{get}()$ operations not issued by p_i have been removed. An history $\hat{H} = (H, \prec)$ is causal consistent if, for every p_i, all the $\mathsf{get}()$ operations of $\hat{H}^i$ are legal.

Definition 9 (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 an instantaneous get operation op that would be executed at time t.*

As an example, consider the execution of Figure 2. The possible admissible sets at time t are, by definition, all the admissible sets for a "virtual" $\mathsf{get}()$ operation *op* executed instantaneously at time t (i.e. $t_B(op) = t_E(op) = t$). Such a $\mathsf{get}()$ operation is concurrent with $\mathsf{add}(1)$ issued by p_j and $\mathsf{remove}(3)$ issued by p_k; it follows $\mathsf{add}(4)$ and $\mathsf{get}()$ issued by p_i, $\mathsf{get}()$ and $\mathsf{remove}(4)$ issued by p_j and $\mathsf{add}(3)$ issues by p_j. Hence, the corresponding sequential set and concurrent set for *op* (the instantaneous $\mathsf{get}()$ operation executed at time t) are respectively $V_{seq}(op) = \emptyset$ (because for both the $\mathsf{add}()$ operations preceding *op* exists a $\mathsf{remove}()$ not following *op*) and $V_{conc}(op) = \{1, 3\}$. Combining these two sets, we obtain four possible admissible sets for *op* and the possible admissible sets at time t are respectively (*i*) $V_{ad}(t) = \emptyset$, (*ii*) $V_{ad}(t) = \{1\}$, (*iii*) $V_{ad}(t) = \{3\}$ and (*iv*) $V_{ad}(t) = \{1, 3\}$.

3 System Model

The distributed system is composed, at each time, by a bounded number 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. Moreover we assume that processes can access a global clock[3].

We assume that there exists an underling protocol, that keeps processes connected each other. This protocol is implemented at the connectivity layer (the layer at the bottom of Figure 3).

3.1 Distributed Computation

A distributed computation is formed, at each instant of time, by a subset of processes of the distributed system. A process p, belonging to the system, that wants to participate to the distributed computation has to execute a $\mathsf{join}()$ operation. Such an operation, invoked at some time t, is not instantaneous: it consumes time. 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 distributed computation implements a set object.

A process leaves the computation in an implicit way. When it does, it leaves the computation forever and does no 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 a process does not crash during the execution of add() and remove() operations.

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

Definition 10 *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]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.

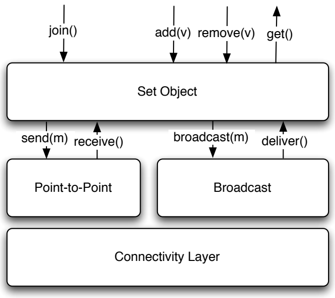


Figure 3: System architecture

3.2 Communication Primitives

Two communication primitives are used by processes belonging to the distributed computation to communicate: point-to-point and broadcast communication as shown in Figure 3.

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 the processes currently present 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 t be the time at which a process p belonging to the distributed 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 [13] in the context of systems where process can commit crash failures. It has been extended to the context of dynamic systems in [11].

Assuming that the processing times are negligible, the bound δ and δ' makes the system synchronous.

3.3 Churn Model

The continuous arrival and departure of nodes in the system is usually referred as *churn* phenomenon. 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 $N(t)$ [5]. The join and the leave distribution are discrete functions of the time that return, 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 are inside the system and we assume that, for each time t, $\lambda(t) = \mu(t) = c \times n_0$ (where $c \in [0, 1]$ is a percentage of node of the system) meaning that, at each time unit, the number of processes that join the system is the same as the number of process that leave it, i.e. the number of processes inside the system $N(t)$ is always equal to n_0.

4 Set Implementation in a Synchronous Dynamic Distributed System

This section presents a value-based sequentially consistent protocol implementing a set in a dynamic distributed systems. Its correctness proof shows show that (liveness) every operation eventually terminates and (safety) the execution history generated by the proposed protocol is value-based sequentially consistency.

4.1 Value-based Seq. Consistent 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 variable that contains the local copy of the set; sn_i is an integer variable that count update operations 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 recent update operations executed by p_i. Such variable contains 4-uples $< type, val, sn, id >$ each one characterizing an operation of type $type = \{A \ or \ R\}$ (respectively for add() and remove()) of the value val, with a sequence number sn, 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 4-uples $< type, val, sn, id >$ 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 set (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 join() operation The algorithm implementing the join operation for a set object, is described in Figure 4, 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 is explained later. After this waiting period, p_i broadcasts (with the 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 03). 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 04-05) and then executes all the operations concurrent with the join contained in $pending_i$ and not yet executed as if the UPDATE message is just received (lines 06-11). Then, p_i becomes active (line 12), which means that it can answer the inquiries it has received from other processes, and does it if $reply\_to \neq \emptyset$ (line 13). Finally, p_i returns ok to indicate the end of the 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 13). 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).

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 5(a). At time t, the processes p_i,

```
operation join($i$):
(01) $sn_i \leftarrow 0$; $last\_ops_i \leftarrow \emptyset$ $set_i \leftarrow \emptyset$; $active\_i \leftarrow false$;
     $pending\_i \leftarrow \emptyset$; $replies_i \leftarrow \emptyset$; $reply\_to_i \leftarrow \emptyset$;
(02) wait($\delta$);
(03) broadcast INQUIRY($i$); wait($2\delta$);
(04) let $< set, sn, ls > \in replies_i$
     such that $(\forall < -, sn', - > \in replies_i : \ sn \geq sn')$;
(05) $set_i \leftarrow set$; $sn_i \leftarrow sn$; $last\_ops_i \leftarrow ls$;
(06) for each $< type, val, sn, id > \in pending_i$ do
(07)     $< tupe, val, sn, id > \leftarrow$ first_element($pending$);
(08)     if ($< type, val, sn, id > \notin last_{op_i}$)
(09)     then execute UPDATE($< type, val, sn, id >$);
(10)     end if
(11) end for;
(12) $active_i \leftarrow true$;
(13) for each $j \in reply\_to_i$ do
(14)     send REPLY ($< set_i, sn_i, last_{op_i} >$) to $p_j$;
(15) 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 $reply\_to_i \leftarrow reply\_to_i \cup \{j\}$
(20)         end if.

(21) when REPLY($< set, sn, ops >$) is received:
             $replies_i \leftarrow replies_i \cup \{< set, sn, ops >\}$.
```

Figure 4: The join() protocol for a set object in a synchronous system (code for p_i)

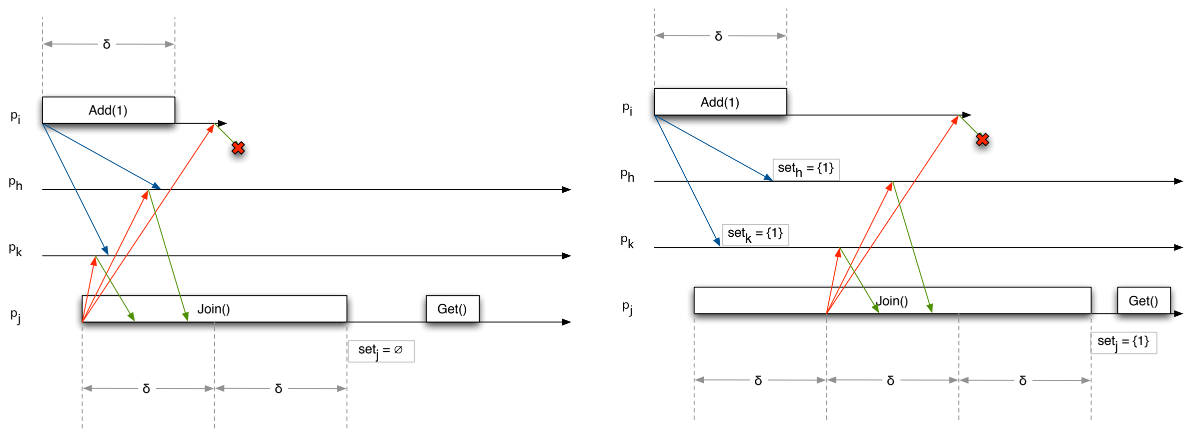
p_h and p_k are the three processes composing the system. Moreover, the process p_j executes join() just after t. The set is initially empty. Due to the 'timely delivery" property of the broadcast invoked by p_i to send the UPDATE message, p_h and p_k deliver the value to be added (i.e. 1) by $t + \delta$. But, since p_j entered the system after t, there is no such a guarantee for it. Hence, if p_j does not execute the wait(δ) statement at line 02, its execution of the lines 03-13 can provide it with the previous value of the set, namely $\emptyset$. If after obtaining $\emptyset$, p_j issues a get() operation it obtains again $\emptyset$, while it should obtain the new set $\{1\}$ (because 1 is the value added and there is no remove() concurrent with this get() issued by p_j).

The execution depicted in Figure 5(b) shows that this incorrect scenario cannot occur if p_j is forced to wait for δ time units before inquiring to obtain the last value of the set.

The get() operation The algorithms for the get() operation is described in Figure 6. The get is purely local (i.e., fast): it consists in returning the current value of the local variable set_i.

The add(v) and the remove(v) operations The add() and the remove() algorithms are shown in Figure 6. Both the add() operation and the remove() operation have the aim to modify the content of the set object by adding and removing respectively an element. Hence, the structure of the two protocols implementing the two operations is the same. In order to assure value-based sequential consistency, all the processes that execute the update operations on the set, have to execute such updates in the same order by applying some deterministic rules. In the proposed algorithm, such deterministic rule is given by the total order of the pairs $< sn, id >$ where sn is the sequence number of the operation and id is the identifier of the process issuing the operation.

When p_i wants to add/remove an element v to/from the set, it increments its sequence number sn_i (line 02 and line 08 of Figure 6), it broadcasts an UPDATE($type, val, sn, id$) message (line 03 and line 09 of



(a) Without wait(δ) (b) With wait(δ)

Figure 5: Why wait(δ) is required

Figure 6) where $type$ is a flag that identify the type of the update (i.e. A for an add() operation or R for a remove() operation), val is the value that has to be added or removed, sn is the sequence number of the operation and id is the identifier of the process that issues the operation. After, it executes the operation on its local copy of the set (line 04 and line 10 of Figure 6) and it stores locally in its $last\_ops_i$ variable the tuple $< type, val, sn, id >$ that identifies the last operation executed on the set (line 05 and line 11 of Figure 6). Then p_i waits for δ time units (line 06 and line 12 of Figure 6) to be sure that all the active processes have received the UPDATE message and finally it returns by the operation (line 07 and line 13 of Figure 6).

```
operation get():   % issued by any process $p_i$ %
(01)  return($set_i$).
____________________________________________________
operation add($v$):   % issued by any process $p_i$%
(02)  $sn_i \leftarrow sn_i + 1$;
(03)  broadcast UPDATE($A, v, sn_i, i$);
(04)  $set_i \leftarrow set_i \cup \{v\}$;
(05)  $last\_ops_i \leftarrow last\_ops_i \cup \{< A, v, sn_i, i >\}$;
(06)  wait($\delta$);
(07)  return($ok$).
____________________________________________________
operation remove($v$):   % issued by any process $p_i$%
(08)  $sn_i \leftarrow sn_i + 1$;
(09)  broadcast UPDATE($R, v, sn_i, i$);
(10)  $set_i \leftarrow set_i / \{v\}$;
(11)  $last\_ops_i \leftarrow last\_ops_i \cup \{< R, v, sn_i, i >\}$;
(12)  wait($\delta$);
(13)  return($ok$).

(14)  when UPDATE($type, val, sn_j, j$) is delivered: % at any process $p_i$ %
(15)  if($\neg active_i$) then $pending_i \leftarrow pending_i \cup \{< type, val, sn_j, j >\}$
(16)                      else execute UPDATE($type, val, sn_j, j$)
(17)  endif.
```

Figure 6: The get(), add() and remove() protocol for a synchronous system (code for p_i)

When p_i receives the UPDATE($type, val, sn_j, j$) from a process p_j, if it is not active, it puts the current UPDATE message in its $pending_i$ buffer and will process it as soon as it will be active, otherwise it executes the UPDATE() procedure shown in Figure 7. In the UPDATE() procedure p_i checks if the sequence number sn_j, corresponding to the current operation, is greater than the one stored by p_i and if it is so, then p_i execute the operation (lines 01-04 of Figure 7). Contrary, p_i checks if in the set of the last executed operation $last\_ops_i$ there is some operation occurred on the same value val; if there is not such an operation, p_i executes the current one (lines 05 - 12 of Figure 7) otherwise, it checks, according to the type of the operation to be executed, if the two operations are in the right order and in positive case, p_i executes the operation (lines 13 - 21 of Figure 7). Finally p_i updates it sequence number (line 22 of Figure 7)).

Garbage Collection Let us remark that the $last\_ops_i$ set variable collects the information related to operations executed on the set. In order to make the protocol working correctly, only the information related to recent operations are needed. Moreover, if the rate of operation is high, each process becomes immediately overloaded of information. To avoid this problem it is possible to define a garbage collection procedure that periodically removes from the $last\_ops_i$ variable the information related to "old" operation. The thread managing the garbage collection is very easy; it is always running and each δ time unit, operations stored more that δ time before are removed. Due to lack of space we omit here the pseudocode of the procedure.

```
procedure UPDATE($type, val, sn_j, j$) % at any process $p_i$ %
(01) if ($sn_j > sn_i$) then $last\_ops_i \leftarrow last\_ops_i \cup \{< type, val, sn_j, j >\}$;
(02)                     if ($type$ = $A$) then $set_i \leftarrow set_i \cup \{val\}$;
(03)                                    else $set_i \leftarrow set_i / \{val\}$;
(04)                     endif
(05) else
(06)      $temp \leftarrow \{X \in last\_ops_i | X =< -, val, -, - >\}$
(07)      if ($temp = \emptyset$)
(08)      then $last\_ops_i \leftarrow last\_ops_i \cup \{< type, val, sn_j, j >$;
(09)            if ($type$ = $A$) then $set_i \leftarrow set_i \cup \{val\}$;
(10)                          else $set_i \leftarrow set_i / \{val\}$;
(11)            endif
(12)      else if (($type = A$)$\wedge$
                ($\nexists < R, -, sn, id >\in temp \mid (sn, id) > (sn_j, j)$))
(13)            then $set_i \leftarrow set_i \cup \{val\}$;
(14)                 $last\_ops_i \leftarrow last\_ops_i \cup \{< type, val, sn_j, j >\}$;
(15)            endif
(16)            if (($type = R$)$\wedge$
                ($\nexists < A, -, sn, id >\in temp \mid (sn, id) > (sn_j, j)$))
(17)            then $set_i \leftarrow set_i / \{val\}$;
(18)                 $last\_ops_i \leftarrow last\_ops_i \cup \{< type, val, sn_j, j >\}$;
(19)            endif
(20)      endif
(21) endif
(22) $sn_i \leftarrow max(sn_i, sn_j)$.
```

Figure 7: The UPDATE() protocol for a synchronous system (code for p_i)

4.2 Correctness proof

In this section we first show that every protocol's operation terminates (Theorem 1). In order to show that the protocol generates only value-based sequential consistent histories if the churn is below a certain bound (Theorem 2) we pass through the following main steps: (*i*) if every active process maintains at any time an admissible set, the execution history is always value based sequential consistent (Lemma 5); (*ii*)if the churn is below a certain bound, every get operation returns an admissible set (Lemma 4). To get this result we also prove that if the churn is below a certain bound, a process that issued a join operation becomes active endowing an admissible set (Lemma 3).

We omit the proof of Lemma 1, Lemma 2 and Lemma 3 that can be found in [6].

Theorem 1 *If a process invokes* join() *and does not leave the system for at least 3δ time units or invokes a* get() *operation or invokes an* add() *operation or a* remove() *operation and does not leave the computation for at least δ time units, then it terminates the invoked operation.*

Proof The get() operation trivially terminates. The termination of the join(), add() and remove() operations follows from the fact that the wait() statements at line 02 of Figure 4 and at line 06 and line 12 of Figure6 terminate s. $\Box_{Theorem\ 1}$

Lemma 1 *Let $c < 1/3\delta$. $\forall t:\ |A[t, t+3\delta]| \geq n_0(1-3\delta c) > 0$.*

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 $t_1 + 3\delta$.*

Lemma 3 *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 $t + 3\delta$.*

Lemma 4 *Let $\mathcal{S}$ be a set object and let op be a* get() *operation issued on $\mathcal{S}$ by some process p_i. If $c < 1/3\delta$, the set of values V returned by op is always an admissible set (i.e. $V = V_{ad}(op)$).*

Proof Let us suppose by contradiction that there exists a process p_i that issues a get() operation *op* on the set object $\mathcal{S}$ that returns a set of value V not admissible for *op*. If V is not an admissible set then one of the following case is verified:

1. There exists a value v contained in $V_{seq}(op)$ that is not contained in the returned set V (i.e. $\exists\, v |\ v \in V_{seq}(op) \wedge v \notin V$);
2. There exists a value v that is returned in the set V but is not contained in $V_{seq}(op)$ nor in $V_{conc}(op)$ (i.e. $\exists\, v |\ v \in V \wedge v \notin V_{sec}(op) \cup V_{conc}(op)$).

Case 1. If $v \in V_{seq}(op)$ then, by definition, there exists an add(v) that precedes *op* and does not exists any remove(v) operation starting before the end of *op*. Since there exists the add(v) operation then there exists also a process p_j issuing such operation that executes the algorithm of Figure 6.

- If $p_i = p_j$, then p_i has executed lines 04-05 of Figure 6 and has added v to its local copy set_i of the set object. Since there not exists any remove(v) operation starting before the end of *op* and since the get() operation returns the content of the local copy of the set without modifying it (line 01 Figure 6) then $v \in V$ and we have a contradiction.
- If $p_i \neq p_j$, then p_j has executed, at some time t, line 03 of Figure 6 by sending an UPDATE (A, v, sn_j, j) message by using the broadcast primitive. Due to the broadcast property, every process that is active at time t will receive the update up to time $t + \delta$.

  If $\mathbf{p_i} \in \mathbf{A(t)}$, then p_i has received p_j's update and has executed the update procedure. If the sequence number attached to the message was greater than the one maintained locally by p_i, then it executes immediately the update by adding v to its local copy of the set (lines 01-04 Figure 6); otherwise it checks if it has in its $last\_ops_i$ an operation already executed that "collides" with the current one (i.e. if there is a $\mathsf{remove}(v)$ of the same element issued by a process with a lower identifier). Since there not exist any $\mathsf{remove}(v)$ operation starting before the end of the $\mathsf{get}()$, when process p_i evaluate the condition at line 07 it finds an empty set and the executes lines 09 - 10 by adding v to its local copy of the set. Since there not exists any $\mathsf{remove}(v)$ operation starting before the end of op and since the $\mathsf{get}()$ operation returns the content of the local copy of the set without modifying it (line 01 Figure 6) then $v \in V$ and we have a contradiction.

$\mathbf{p_i} \notin \mathbf{A(t)}$, it means that it is executing or it will execute the join protocol. If it is executing the join protocol then it will buffer the update message of p_j and will execute the update just before become active (lines 06-11 Figure 4); p_i will add v to its local copy of the set. Since there not exists any $\mathsf{remove}(v)$ operation starting before the end of op and since the $\mathsf{get}()$ operation returns the content of the local copy of the set without modifying it (line 01 Figure 6) then $v \in V$ and we have a contradiction. Contrary, if p_i is not joining the system at time t when it will join, it asks the local copy of the set to other active processes. Due to Theorem 3, at the end of the join, p_i will have in its local copy of the set a set that is admissible and will include v. Even in this case, since there not exist any $\mathsf{remove}(v)$ operation starting before the end of op and since the $\mathsf{get}()$ operation returns the content of the local copy of the set without modifying it (line 01 Figure 6) then $v \in V$ and we have a contradiction.

Case 2. Since at the beginning of the computation p_i has in its local copy of the set an empty set and since $v \in V$ then p_i has added v to the local copy set_i (i) during the join operation or (ii) managing an update message.

- If p_i has added v to set_i as consequence of the join, it means that v is a value of an admissible set at time t of the end of the join (cfr. Theorem 3). If v belongs to an admissible set at time t, it means that v belongs to $V_{seq}(o)$ or to $V_{conc}(o)$, where o is an instantaneous $\mathsf{get}()$ operation issued at time t. If $v \in V_{seq}(o) \cup V_{conc}(o)$ then there exists an $\mathsf{add}()$ operation that terminates or is running at time t. Since $t < t_E(op)$ and since there not exist any $\mathsf{remove}(v)$ operation starting before the end of op, $v \in V_{seq}(op)$ and we have a contradiction.

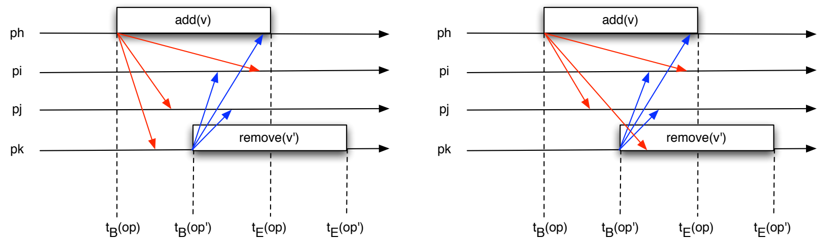
- If p_i has added v to set_i as consequence of an UPDATE message it means that there exists a process p_j that has sent it. An UPDATE message is generated by a process p_j when the $\mathsf{add}()$ operation is issued; this means that there exist an $\mathsf{add}()$ operation that precedes or is concurrent with op. Since there not exist any $\mathsf{remove}(v)$ operation starting before the end of op, $v \in V_{seq}(op) \cup V_{conc}(op)$ and we have a contradiction.

$\Box_{Lemma\ 4}$

Lemma 5 *Let $\mathcal{S}$ be a set object and let $\hat{H} = (H, \prec)$ be an execution history of $\mathcal{S}$ generated by the algorithm in Figure 4 and Figure 6. If every active process p_i maintains an admissible set, $\hat{H}$ is always value-based sequential consistent.*

Proof Let us suppose by contradiction that there exists an history $\hat{H} = (H, \prec)$ generated by the algorithms in Figure 4 and Figure 6 such that $\hat{H}$ is not value-based sequentially consistent. If $\hat{H}$ is not value-based sequential consistent then there exist two processes p_i and p_j that admit two linear extensions, respectively $\hat{S}_i$ and $\hat{S}_j$, where at least two concurrent operation $op = \mathsf{add}(v)$ and $op' = \mathsf{remove}(v')$ (with $v = v'$) appear in a different order. Without loss of generality, let us suppose that op and op' have no other concurrent operation, op is issued by a process p_h, op' is issued by a process p_k and $t_B(op) < t_B(op')$.

At time $t < t_B(op)$ all the processes has the same value for their sequence numbers[4] and let us suppose that $sn_w = x$ for every p_w. When process p_h issues the $\mathsf{add}(v)$ operation, it increments its sequence number $sn_i = x+1$ by executing line 02 of Figure 6 and attaches such value to the $\text{UPDATE}(A, v, x+1, h)$ message. We can have two cases: (i) process p_k receives the UPDATE message of p_h before issuing the $\mathsf{remove}(v)$ operation (Figure 8(a)) or (ii) process p_k receives the UPDATE message of p_h just after issuing the $\mathsf{remove}(v)$ operation (Figure 8(b)).



(a) (b)

Figure 8: Concurrent execution of $\mathsf{add}(v)$ and $\mathsf{remove}(v)$.

Let us show what happen to p_i and p_j in both cases.

Case 1. When p_k receives the $\text{UPDATE}(A, v, x+1, h)$, the received sequence number is greater than the one maintained by p_k then p_k executes the update (lines 02-04) and sets its sequence number to the one received by executing line 22 (i.e. $sn_k = x+1$). Just after, p_k issues a $\mathsf{remove}(v)$ operation, it increments its sequence number $sn_k = x+2$ by executing line 02 of Figure 6 and attaches such value to the $\text{UPDATE}(R, v, x+2, k)$ message. Let us consider the scenario depicted in Figure 8(a) where p_i receives first the update message of the $\mathsf{remove}(v)$ sent by p_k and then the update message of the $\mathsf{add}(v)$ sent by p_h while p_j receives first the update message of the $\mathsf{add}(v)$ sent by p_h and then the update message of the $\mathsf{remove}(v)$ sent by p_k.

behavior of process p_i. When p_i receives the $\text{UPDATE}(R, v, x+2, k)$ message, its sequence number is smaller than the one received (i.e. $sn_i = x$) then it executes lines 02-04 removing the value v (if it is already contained in the set) and storing the tuple $< R, v, x+2, k >$ in the $last\_ops_i$ set, and then it sets its sequence number to the one received by executing line 22 (i.e. $sn_i = x+2$). Later, p_i receives the $\text{UPDATE}(A, v, x+1, h)$ message and, since its sequence number is now smaller that the received one, it executes lines 06-21. In particular, p_i examines the $last\_ops_i$ buffer and founds an entry (i.e. $< R, v, x+2, k >$) with the value greater than to the one received (line 06) then, it checks if the received operation can be executed or it is "overwritten" by the one already processes by applying the deterministic ordering based on the pair $< sn, id >$. Since the condition at line 12 is false, then p_i does not execute the update. Note that, not executing the $\mathsf{add}(v)$ is equal to execute $\mathsf{add}(v)$ followed by a $\mathsf{remove}(v)$ operation. Hence, in the linear extension $\hat{S}_i$ of p_i we have that $op \rightarrow_{S_i} op'$.

behavior of process p_j. When p_j receives the $\text{UPDATE}(A, v, x+1, h)$ message, its sequence number is smaller than the one received (i.e. $sn_j = x$) then it executes lines 02-04 adding the value v and storing the tuple $< A, v, x+1, h >$ in the $last_{ops_j}$ set, and then it sets its sequence number to the one received by executing line 22 (i.e. $sn_j = x+1$). Later, p_j receives the $\text{UPDATE}(R, v, x+2, k)$ message and another time its sequence number is smaller than the one received (i.e. $sn_j = x+1$); hence it executes lines 02-04

[4]Let us recall that the sequence number maintained by every process counts the number of updates issued on the local copy of the set.

removing the value v. Hence, in the linear extension $\hat{S}_j$ of p_j we have again that $op \rightarrow_{S_j} op'$. Since the two operations appears in the same order both in $\hat{S}_i$ and $\hat{S}_j$ we have a contradiction.
Case 2. When p_k issues the remove(v) operation, it has not yet received the UPDATE messaged sent by p_h and the the two sequence number of both the operation are the same (i.e. $sn_h = sn_k = x + 1$). Let us consider the scenario depicted in Figure 8(b) where p_i receives first the update message of the remove(v) sent by p_k and then the update message of the add(v) sent by p_h while p_j receives first the update message of the add(v) sent by p_h and then the update message of the remove(v) sent by p_k.
behavior of process p_i. When p_i receives the UPDATE$(R, v, x + 1, k)$ message, its sequence number is smaller than the one received (i.e. $sn_i = x$) then it executes lines 02-04 removing the value v (if it is already contained in the set) and storing the tuple $< R, v, x + 1, k >$ in the $last\_ops_i$ set, and then it sets its sequence number to the one received by executing line 22 (i.e. $sn_i = x + 1$). Later, p_i receives the UPDATE$(A, v, x + 1, h)$ message and, since its sequence number is equal to the received one, it executes lines 06-21. In particular, p_i examines the $last\_ops_i$ buffer and founds an entry (i.e. $< R, v, x + 1, k >$) with the value equals to the one received (line 06) then, it checks if the received operation can be executed or it is "overwritten" by the one already processes by applying the deterministic ordering based on the pair $< sn, id >$. Since the condition at line 12 is false, then p_i does not execute the update. Note that, even in this case not executing the add(v) is equal to execute the add(v) followed by the remove(v) operation. Hence, in the linear extension $\hat{S}_i$ of p_i we have that $op \rightarrow_{S_i} op'$.
behavior of process p_j. When p_j receives the UPDATE$(A, v, x + 1, h)$ message, its sequence number is smaller than the one received (i.e. $sn_j = x$) then it executes lines 02-04 adding the value v and storing the tuple $< A, v, x + 1, h >$ in the $last_{ops_j}$ set, and then it sets its sequence number to the one received by executing line 22 (i.e. $sn_j = x + 1$). Later, p_j receives the UPDATE$(R, v, x + 1, k)$ message and, since its sequence number is now equal to the received one, it executes lines 06-21. In particular, p_j examines the $last_{ops_j}$ buffer and founds an entry (i.e. $< A, v, x + 1, h >$) with the value equals to the one received (06) then, it checks if the received operation can be executed or it is "overwritten" by the one already processes by applying the deterministic ordering based on the pair $< sn, id >$. Since the condition at line 12 is true, then p_j executes the update removing v. Hence, in the linear extension $\hat{S}_j$ of p_j we have again that $op \rightarrow_{S_j} op'$.

Since the two operations appears in the same order both in $\hat{S}_i$ and $\hat{S}_j$ we have a contradiction.

$\square_{Lemma\ 5}$

¿From Lemma 4 and Lemma 5 follow the theorem that concludes our proof.

Theorem 2 *Let $\mathcal{S}$ be a set object and let $\hat{H} = (H, \prec)$ be an execution history of $\mathcal{S}$ generated by the algorithm in Figure 4 and Figure 6. If $c < 1/3\delta$, $\hat{H}$ is always value-based sequential consistent.*

4.3 Discussion

Let us point out that the bound on the churn strictly depends on the structure of the protocol. The implementation proposed here has lightweight set access operations (in terms of messages exchanged and latency to be completed) and an heavyweight operation to join the computation. This is based on the assumption that set access operations, namely get() add(), remove(), are more frequent than join(). Other implementations can be considered h aving lightweight operations to join the computation and heavyweight operations that access the set based on a request-reply paradigm instead of a simple push based procedure. These implementations can tolerate higher churn at the cost of more expensive set accesses.

5 Related Work

Dynamicity Model Dynamic systems are nowadays an open field of research and new models able to capture all the aspects of such dynamicity are going to be defined. In [1, 20] are presented models, namely *infinite arrival models*, able to capture the evolution of the network removing the constraint of having a predefined and constant size n. These models do not address the way the processes join or leave the system. More recently, other models have been proposed that take into account the process behavior. This is done by considering both probabilistic distribution [18], or deterministic distribution [15], on the join and leave of nodes (but in both cases the value of the system size is constant).

Registers and Weak-Set Among shared objects, registers are certainly one of the basic one. A register is a shared variable that can be accessed by processes by means of two operations, namely $\mathsf{write}()$ and $\mathsf{read}()$, used to store a value in the register and to retrieve the value from the object. According to the set of values that can be returned by a $\mathsf{read}()$ operation, Lamport has defined different type of registers [17] as *regular* or *atomic*. In [4] an implementation of a regular register in a dynamic distributed system subject to churn is provided while in [10] an atomic register is implemented in a mobile ad-hoc network. In [9], the authors show how it is possible to implement a weak-set in a static system, by using a finite number of atomic registers, in two particular cases: (i) when the number of processes is finite and known and (ii) when the set of possible values that can be added to the set is finite and show that a weak-set is stronger than a regular register. Unfortunately, in the model considered in this paper, it is not possible to implement a set object by using a finite number of registers. The intuition besides such impossibility is that (i) the domain of the set is possibly infinite and (ii) it is not possible to rely on the number of processes as in the solutions proposed in [9] without using an infinite number of registers. Even if, at each time unit there are always n processes in the system, they change along time and possibly infinite processes may participate to the computation.

Tuple Space A tuple space [12] is a shared memory object where generic data structures, called tuples, are stored and retrieved. A tuple space is defined by three operations: $\mathsf{out}(t)$ which outputs the entry t in the tuple space (i.e. write), $\mathsf{in}(\bar{t})$ which removes the tuple that matches with $\bar{t}$ from the tuple space (i.e. destructive read) and $\mathsf{rd}(\bar{t})$ that is similar to the previous one b ut without removing the tuple. A set object, as presented in this paper, is something different from a tuple space. In fact, even if $\mathsf{add}(v)$ and $\mathsf{remove}(v)$ can be seen as a particular case of $\mathsf{in}(\bar{t})$ and $\mathsf{out}(t)$ (i.e. where the tuple is composed only from one value), the set object differs from a tuple space for the possibility to return the complete content of the object without any parameter (while in a tuple space the $\mathsf{rd}(\bar{t})$ operation needs a tuple $\bar{t}$ as parameter to be matched).

6 Conclusion

Shared objects provide programmers with a powerful way to design distributed applications on top of complex distributed systems. This paper has introduced a set object suited to dynamic systems. The paper has presented a consistency condition for set objects that is weaker than sequential consistency (by exploiting the semantic of the set object and allowing, at the same time, concurrent readings to return the same set in absence of other operations). The paper also presented a value-based sequentially consistent implementation of the set object in a dynamic, synchronous and anonymous distributed systems. A proof has been given that shows that the proposed protocol is correct when the churn remains below a given threshold. This shows that "provably correct software" can be extended to dynamic systems.

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] Ahamad M., Burns J. E., Hutto P. W., Neiger G. and Kohli P., Causal memory: definitions, implementations and programming. *Distributed Computing*, 9:37-49, 1995.

[3] Attiya H., Welch J. L. Sequential Consistency versus Linearizability ACM Transaction on Computer Systems 12(2): 91-122 (1994)

[4] 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 Press, pp. 639-647, 2009.

[5] Baldoni R., Bonomi S., Raynal M. Regular Register: an Implementation in a Churn Prone Environment. *16th International Colloquium on Structural Information and Communication Complexity* (SIROCCO) 2009.

[6] Baldoni R., Bonomi S., Raynal M. Joining a Distributed Shared Memory Computation in a Dynamic Distributed System. *Proc. 7th Workshop on Software Technologies for Future Embedded and Ubiquitous Computing Systems (SEUS'09)*, Springer-Verlag LNCS #5860, pp. 91-102, 2009.

[7] CoMiFin - Communication Middleware for Monitoring Financial Critical Infrastructure. http://www.comifin.eu/

[8] Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. *6th Symposium on Operating System Design and Implementation (OSDI 2004)*, 2009.

[9] Delporte-Gallet C., Fauconnier H. Two Consensus Algorithms with Atomic Registers and Failure Detector Ω. *10th Int'l Conf. on Distr. Computing and Networking (ICDCN 2009)* Springer-Verlag, LNCS 5408, pp 251-262, 2009.

[10] 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.

[11] 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.

[12] Gelernter D., Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), 80-112, 1985.

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

[14] 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.

[15] 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.

[16] Lamport L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs IEEE Transaction on Computers (TC) 28(9):690-691, 1979.

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

[18] 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.

[19] Lipton R. J., Sandberg J. S. PRAM: a scalable shared memory Tech. Report CS-TR-180-88, Princeton Univ., 1988.

[20] 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.

[21] Raynal M. and Schiper A. A suite of formal definitions for consistency criteria in distributed shared memories. *Proc. 9-th Int'l IEEE Conference on Parallel and Distributed Computing Systems (PDCS)*, IEEE Computer Press, pp. 125-131, 1996.