Implementing Distributed Computing Abstractions in the presence of Churn

Silvia Bonomi
Università degli Studi di Roma “La Sapienza”
Dottorato di Ricerca in Ingegneria Informatica
XXII Ciclo - 2010

Silvia Bonomi

Implementing Distributed Computing Abstractions in the presence of Churn

Thesis Committee
Prof. Roberto Baldoni (Advisor)
Prof. Michel Raynal (Advisor)
Prof. Massimo Mecella

Reviewers
Prof. Jennifer L. Welch
Prof. Alexander A. Shvarstman
Author’s address:
Silvia Bonomi
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma “La Sapienza”
Via Ariosto 25, I-00185 Roma, Italy
E-mail: bonomi@dis.uniromal.it
www: http://www.dis.uniromal.it/~bonomi/
# Contents

## Introduction

1. Modeling Dynamic Distributed Systems 9
   1.1 Distributed System .............................................. 9
   1.2 Distributed Computation ......................................... 10
   1.3 Communication Primitives and Timing Assumptions .............. 11
      1.3.1 Synchronous System ......................................... 12
      1.3.2 Asynchronous System ......................................... 12
      1.3.3 Eventually Synchronous System ............................... 13
   1.4 Churn Model .................................................. 13
      1.4.1 Definitions ................................................ 13
      1.4.2 Churn constraint on the upper and lower bounds on the net-
      work size ................................................... 16
   1.5 Related Work ................................................ 17

2. Implementing a Regular Register in the Presence of Churn 21
   2.1 Related Work ................................................ 22
   2.2 Single-writer/Multi-reader Regular Register Specification .... 25
   2.3 Synchronous Systems ........................................... 25
      2.3.1 Protocol ................................................ 25
      2.3.2 Correctness ............................................... 28
   2.4 Asynchronous System ........................................... 31
      2.4.1 Impossibility Result .................................... 31
   2.5 Eventually Synchronous Systems ................................ 32
      2.5.1 Protocol ................................................ 32
      2.5.2 Correctness ............................................... 36
   2.6 Multi-writer/Multi-reader Regular Register Specification .... 42
      2.6.1 Discussion ............................................... 44

3. Implementing a Set Object in the Presence of Churn 47
   3.1 Related Work ................................................ 48
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Set Object S</td>
<td>49</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Basic Definitions</td>
<td>50</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Value-based Sequential Consistency Condition</td>
<td>53</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Admissible set at time $t$</td>
<td>55</td>
</tr>
<tr>
<td>3.3</td>
<td>Synchronous System</td>
<td>57</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Value-based Sequential Consistent Protocol</td>
<td>57</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Correctness proof</td>
<td>62</td>
</tr>
<tr>
<td>3.4</td>
<td>Eventually Synchronous System: Impossibility Result</td>
<td>73</td>
</tr>
<tr>
<td>3.5</td>
<td>$k$-bounded Set Object</td>
<td>75</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Protocol for an Eventually Synchronous System</td>
<td>77</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Correctness proof</td>
<td>83</td>
</tr>
<tr>
<td>3.6</td>
<td>$k$-bounded Set Applications</td>
<td>90</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Eventual Participant Detector</td>
<td>90</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Shared List</td>
<td>91</td>
</tr>
<tr>
<td>Conclusion</td>
<td></td>
<td>93</td>
</tr>
</tbody>
</table>
Introduction

Due to the evolution of technologies and communication paradigms, modern distributed systems are evolving towards novel computational models. Last few years have been characterized by an always increasing growth of computational capabilities of devices, bandwidth availability and diffusion of wireless communications, that let distributed systems moving from classical models to new classes of systems and applications. Vehicular Area NETworks (VANETs), peer-to-peer networks (P2P), Wireless Sensor Networks (WSNs), Smart Environments, Cloud Computing are just few examples of emerging systems and applications.

Such systems have as common point the impossibility to be modeled as done in common distributed systems due to the uncertainty on the set of processes participating to the application computation. As a consequence, all the problems that have been proved correct under a well defined set of assumptions are newly under discussion when instantiated in these new settings.

As an example, let consider the Skype context; Skype provides a messaging and VoIP service, built on top of a peer-to-peer overlay network where users cooperate by sharing their resources (CPU, bandwidth, memory space) to implement the service [13]. Applications developed in such context are characterized by the complete autonomy of processes; in fact, they can decide autonomously and independently when enter the computation and when leave it. Therefore, to take into account such emerging behavior, it becomes mandatory to define procedures able to manage unexpected and possibly large changes, due to the arrival and departure of processes (a phenomenon also called churn).

Note that, to guarantee applications availability, such procedures have to be implemented in a distributed fashion; on the contrary, the responsible for the application management can become quickly overloaded. In other words, the system has to be self-managed to quickly react to changes and to provide availability and quality of service in spite of continuous changes.

Another example is given by Google’s MapReduce; it is one of the most remarkable ways to structure huge computations in order to process and aggregate data in a parallel fashion [17]. These parallel programming models use shared data structures, or objects, to communicate among processes. In such large scale settings the environment cannot be fully managed, but rather it requires a high degree of self-
Introduction

management.

Considering such dynamic behavior, modern applications have several noteworthy implications. First, the design and the implementation of distributed computing abstractions have to deal not only with asynchrony and failures, but also with the system dynamicity dimension. Hence, abstractions and protocols implementing them have to be reconsidered to take into account this new challenging setting. Dynamicity makes abstractions more difficult to understand and master than in classical static distributed systems where the set of processes is fixed once for all. Although numerous protocols have been designed for dynamic distributed message-passing systems, few papers (such as [1], [5], [54]) strive to present models suited to such systems, and extremely few dynamic protocols have been proved correct. While up to now the most common approach used to address dynamic systems is mainly experimental, a precise model is needed to find the limits that make implementations working correctly despite arrival and departures of nodes. Hence, providing theoretical bounds on the maximum churn rate tolerated by applications, still working correctly, is currently an issue.

Moreover, many of the protocols dealing with dynamicity assume that the churn is quiescent [3], [33], [44], [54], i.e. the arrivals and departures of processes eventually end and the system converges to its target. However, such assumption could be not completely realistic for some scenarios, like Skype, where processes continuously change, although with different arrival and departure rates [28]. To this aim, models considering continuous churn represent an interesting perspective to evaluate the robustness of applications in very challenging scenarios.

Another interesting point is that processes have to be capable to join on-the-fly a computation. Hence, there is the need to define data structures and procedures that support concurrent writes and reads providing, at the same time, a degree of consistency which is adequate to solve a given problem despite the presence of churn. In distributed systems affected by failures, there are very precise bounds on the number of failures that makes possible to implement shared object abstractions [2], [43]. Contrary, bounds on the arrivals and departures of processes do not exist considering the same abstractions in a dynamic system with continuous churn. In particular, given an abstraction implementable in a static distributed system under some timing assumption, it is not possible to say if it is still implementable (under the same timing assumptions) in a dynamic distributed system.

The focus of this thesis is on the analysis of issues arising from considering distributed shared object abstractions in a distributed system where processes continuously join and leave the computation. The interests of shared objects lies in the fact that they are very powerful abstractions, provided to distributed system developers, due to their ability to allow inter-process communication by hiding all the complexity following from the message-exchange behind their implementation. In static dis-
tributed systems, the weaker conditions that make possible the implementation of a shared object are well known. The same is not true, moving towards a dynamic system prone to continuous churn. In this thesis, we try to take a step towards the definition of algorithms, implementing shared objects, and bounds on the dynamicity ensuring protocols to remain correct. Two shared object abstractions have been considered (i) a regular register and (ii) a set object.

A register is one of the most basic concurrent objects (i.e. object that can be accessed concurrently by several processes) that can be implemented on top of a message passing system; the selection of a regular register is due to the fact that regular registers have the same computability power as atomic registers, but are easier to implement in a traditional distributed system [2]. Moreover, interestingly enough, regular registers allow to solve some basic coordination problems such as the consensus problem in asynchronous systems prone to crash and where processes are equipped with an appropriate leader election service [22, 26].

A set object, is a complex shared object able to store values. Each process can acquire the content of the set through a get operation while it can add (remove) an element to the object through an add (remove) operation.

In static distributed systems, where processes do not join and leave, a set object can be emulated by using atomic registers. In particular, an implementation is possible by using a finite number of registers, one associated to each process of the system. Moving to a dynamic system, it is not possible to implement a set object by using a finite number of registers; the intuition follows from the consideration that the number of processes participating to the computation is unbounded and then also the number of the registers can grow to infinite. To this aim, an implementation based on message exchanges is investigated.

The interest of the set object lies in the fact that it is a very useful abstraction in many modern applicative contexts. As an example, let us consider a collaborative intrusion detection system composed by several processes each one belonging to a distinct organization. A process adds an IP address to the set if it suspects that IP address (for instance, to do port scanning on some edge host of the organization). A process can remove an IP address if the suspicion reveals to be wrong. A set represents a blacklist where a get is invoked returning the list of all suspected IP addresses.

Note that, for both the shared object abstractions, a consistency criteria weaker than linearizability [32] has been considered. The general motivation is that linearizable implementations are quite expensive (in terms of number of messages required) and, in many applications, weaker consistency conditions are enough to let the objects usable as basic building blocks.
Contributions. In the following are reported the main contributions of this thesis.

The first contribution is the definition of a general churn model able to characterize the evolution of the computation both qualitatively (i.e. by describing the computation from an external perspective) and quantitatively (i.e. by describing, also numerically, how processes are evolving at each time unit).

The second contribution is the implementation of a regular register abstraction in a distributed system with continuous churn. An interesting point is that, the continuous churn introduces in the system a form of uncertainty that, combined with the asynchrony, makes impossible to provide an implementation that ensure both safety and liveness properties of the object. To master such uncertainty, a given degree of synchrony is required. As soon as the system is eventually synchronous, the regular register becomes implementable. Note that, using the generic churn model introduced so far, it is possible to provide deterministic lower bounds on the required number of processes that have to participate in the computation to ensure a correct execution.

The third contribution is a formal specification of a set as a shared object, together with a new consistency criteria, namely value-based sequential consistency, based on the semantics of the object. The notion of set as data structure, is well know but there not exist (to the best of my knowledge) any formal specification of its behavior as concurrent object. A set object can be used as basic building block for interesting applications (i.e. participant detector, black list for intrusion detection system etc...) and depending on the requirements of such applications, several consistency levels, in spite of concurrency, are required.

The value-based sequential consistency property allows concurrent get operations to return the same output in the absence of concurrent add and/or remove operation. This condition is actually weaker than linearizability [32] due to the semantics of the set object. In fact, given two concurrent add or remove operations, it is important that both the values added (or removed) concurrently belong, or not, to the object and it does not matter in which order they have been inserted (or removed).

The fourth contribution is the implementation of a set object in a distributed system with continuous accesses and churn, and with finite memory for local computation. It is interesting to see that the complexity of the object, combined with the asynchrony and continuous accesses and churn, makes no possible any implementation based on message-exchange, using finite memory space for local computation. In fact, a totally synchronous system is required to implement a set. Under the synchrony assumption, it is also provided an upper bound on the churn rate that the protocol can tolerate remaining correct.

The fifth contribution is the definition of a weak, but rather interesting, object, namely
The contributions listed above and basic ideas from which this work has started are partially contained in the following papers: [6], [10], [11], [12]. Moreover, this work leverages ideas coming from the following papers done during the Ph.D. studies: [7], [8], [9], [48].

Thesis Organization. In Chapter 1 is presented the model of the environment; particular attention has been given to the modeling of the distributed computations considered in this thesis, by decoupling the environment (i.e. connectivity and communication aspects) from shared object related computation. In Section 1.4 the dynamicity, due to continuous arrival and departure of processes from the distributed computation, is modeled (i.e. the churn).

In Chapter 2 is addressed the problem of building a regular register in presence of continuous churn. In particular, Section 2.3 presents an implementation in a dynamic distributed system, assuming that the underline system is synchronous. Such a protocol exploits the synchrony of the communication to implement fast and lightweight read() operations. Section 2.5 presents an implementation, assuming the underline system eventually synchronous and assuming that, at each time unit, there is a majority of processes in the computation, that are participating actively (i.e. processes that have terminated the join() operation). Section 2.4 shows that it is not possible to implement a regular register in a distributed system affected by continuous churn if the underlying system is fully asynchronous.

In Chapter 3 is addressed the problem of implementing a complex shared object, namely a set object, in presence of continuous churn and continuous accesses to the object. In particular, Section 3.2 provide the specification of a set object, Section 3.3 provide an implementation using finite memory for local computation and assuming that the underline system is synchronous. In Section 3.4 it is shown that, surprisingly, it is not possible to implement a set object in an eventually synchronous message-passing system subject to continuous churn, if processes have limited memory for local computation and the accesses to the object are continuous. To overcome this impossibility, in Section 3.5 the specification of the set is weakened, and a k-bounded set is defined. Section 3.5.1 shows that such object is implementable in an eventually synchronous system with continuous accesses and churn, using finite memory.

Each chapter is provided with a related work section, and finally Conclusion and Future work are presented.
Chapter 1

Modeling Dynamic Distributed Systems

In this chapter is described the model of the environment where a regular register abstraction and a set object abstraction will be implemented. First, it is described the general distributed system and then it is specified how the distributed computation, implementing shared object abstractions, is built on top of such distributed system. The distributed computation is based on message exchanges and then the communication primitives, able to support such computation, are specified with respect to the timing assumption of the system. Finally, it is introduced a new churn model that describes, numerically, the churn on the distributed computation caused by the continuous arrival and departure of processes.

1.1 Distributed System

The distributed system is composed by an infinite set of process $\Pi = \{p_1, p_2, \ldots, p_i, \ldots\}$ (also called nodes), each having a unique identifier (i.e. the index) and a finite memory space for local computation.

The passage of time in the system is measured by a common global clock, represented by a set of integer $\{t_0, t_1, \ldots, t_i, \ldots\}$, not accessible by the processes.

At each time unit, the number of processes inside the distributed system is finite. In Figure 1.1 it is represented the distributed system architecture. Processes of the distributed system are connected with each other through an Overlay Network, i.e. a logical network built on top of the physical one. In this thesis, the overlay network is not considered directly but rather it is abstracted by assuming the existence of an underlying overlay management protocol, that keeps processes connected. Some examples of topologies and protocols keeping the system connected.
in a dynamic environment are [7], [9], [35], [36], [39], [53].

On top of the overlay network, processes belonging to the distributed system are allowed to communicate by exchanging messages and each message is uniquely identified (i.e. by a sequence number or by the sender’s identifier). Messages are exchanged by using two communication primitives (detailed in Section 1.3) built on top of the connectivity layer:

- point-to-point communication;
- broadcast communication.

Inside the distributed system, processes may participate to a distributed computation and in this thesis only shared object computations are considered. Without loss of generality, it is assumed that a process, not involved in any computation, discards all the computation-related messages. This is equivalent to say that a process is not aware of any computation detail (i.e. the state of the shared object) before it decides to participate and after it has decided to leave.

1.2 Distributed Computation

Processes belonging to the distributed system may decide autonomously to participate or not in a distributed computation running on top of the distributed system. Hence, a distributed computation is formed, at each instant of time, by a subset of processes of the distributed system. Processes decide autonomously when join the computation and when leave the computation: in this sense the computation is dynamic.

A process $p_i$, belonging to the system, that wants to participate to the distributed computation has to execute the \texttt{join()} operation. Such operation, invoked at some
time \( t \), is not instantaneous: it consumes time. But, from time \( t \), the process \( p_i \) can receive and process messages related to the computation sent by any other process that belongs to the system and that participate to the computation (i.e. \( p_i \) stops to filter computation-related messages).

When a process \( p_j \), belonging to the distributed computation, does no more want to participate, it leaves the computation. The leave operation is performed in an implicit way and \( p_j \) leaves the computation forever and not longer sends messages. From a practical point of view, if a process that has left the computation wants to re-enter, it has to do it as a new process (i.e., with a new name).

In order to formalize the set of processes that participate actively to the computation, the following definition is given:

**Definition 1** A process is active from the time it returns from the \texttt{join()} operation until the time it leaves the system.

The set \( A(t) \) denotes the set of processes that are active at time \( t \).

The set \( A([t_i, t_j]) \) denotes the set of processes that are active for the whole period \([t_i, t_j] \) (i.e. \( A([t_i, t_j]) = \{ p \mid \forall \tau \in [t_i, t_j] \ p \in A(\tau) \} \)).

Processes participating to the distributed computation implements a shared object abstraction by doing local computation and communication steps. The processing times of local computation are negligible with respect to communication delays, so they are assumed to be equal to 0. Contrary, communication steps take time to be executed.

### 1.3 Communication Primitives and Timing Assumptions

Processes use the following communication primitives to exchange messages:

- broadcast primitive;
- point-to-point primitive.

**Broadcast Primitive.** It is assumed that the system is equipped with an appropriate broadcast communication sub-system that provides the processes with two operations, denoted \texttt{broadcast()} and \texttt{deliver()}. The former allows a process to send a message to all the processes in the system, while the latter allows a process to deliver a message. Consequently, we say that such a message is “broadcast” and “delivered”.

Such a pair of broadcast operations has first been formalized in [29] in the context of systems where process can commit crash failures. It has been extended to the context of dynamic systems in [21]. The properties of such primitives are defined in the following depending by the timing assumptions.
Note that the implementation of the broadcast primitive, on top of the overlay network, depends from the overlay structures and in the worse case it can be obtained by flooding the entire overlay.

**Point-to-Point Primitive.** The network allows a process \( p_i \) to send a message to another process \( p_j \) as soon as \( p_i \) knows that \( p_j \) has entered the system. The network is reliable in the sense that it does not lose, create or modify messages.

### 1.3.1 Synchronous System

A synchronous distributed system can be characterized by the following properties [27]:

- **Synchronous computation**: there is a known upper bound on processing delays;
- **Synchronous communication**: there is a known upper bound on message transmission delays;
- **Synchronous physical clocks**: processes are equipped with a local physical clock.

By exploiting the properties of a synchronous system, the two communication primitives can be defined as follows:

- **Point-to-point Timely Delivery**: there exists an integer \( \delta' \), known by processes, such that if \( p_i \) invokes “\text{send } m \text{ to } p_j \text{” at time } t \), then \( p_j \) receives that message by time \( t + \delta' \) if it has not left the system by that time.

- **Broadcast Timely Delivery**: Let \( t \) be the time at which a process \( p \) invokes \text{broadcast}(m). There is a constant \( \delta \) (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 \( t + \delta \).

### 1.3.2 Asynchronous System

An asynchronous distributed system is characterized by the absence of any timing assumption about processes and channels. Processes have no access to any sort of physical clock and any bounds on processing or communication delays can be assumed.

- **Point-to-point Delivery**: let \( p_i \) be a process invoking a “\text{send } m \text{ to } p_j \text{”}. If both \( p_i \) and \( p_j \) do not leave the system, then \( p_j \) eventually receives that message.

- **Broadcast Delivery**: Let \( t \) be the time at which a process \( p \) invokes \text{broadcast}(m). If \( p \) does not leave the system, then all the processes that are in the system at time \( t \) and do not leave, eventually deliver \( m \).
1.4. CHURN MODEL

Note that, such communication primitives are quite weak in a distributed system prone to continuous churn and they can be considered working in a “best-effort” mode.

1.3.3 Eventually Synchronous System

Generally, distributed systems appear to be synchronous. More precisely, for most systems, it is relatively easy to define physical time bounds that are respected most of the time. However, periods where the timing assumptions do not hold are possible. In order to deal with such an issue, in static systems, an eventually synchronous distributed system\(^1\) is modeled as a system that after an unknown but finite time behaves synchronously [15, 20].

Under such assumptions, the two communication primitives used in this thesis can be formalized as follows:

- **Point-to-Point Eventual Timely Delivery**: There is a time \(t\) and a bound \(\delta\) such that any message sent at time \(t' \geq t\), is received by time \(t' + \delta\) to the processes that are in the system during the interval \([t', t' + \delta]\).

- **Broadcast Eventual Timely Delivery**: There is a time \(t\) and a bound \(\delta\) such that any message broadcast at time \(t' \geq t\), is delivered by time \(t' + \delta\) to the processes that are in the system during the interval \([t', t' + \delta]\).

It is important to observe that the time \(t\) and the bound \(\delta\) do exist but can never be explicitly known by the processes.

1.4 Churn Model

1.4.1 Definitions

The dynamicity due to the continuous join and leave of nodes in the computation is a phenomenon identified under the name **churn**. This section introduces a general model able to characterize such dynamicity.

The model proposed here is based mainly on the definition of two functions (i) the join function \(\lambda(t)\) that defines the join of new processes to the system with respect to time and (ii) the leave function \(\mu(t)\) that defines the leave of processes from the system with respect to time. Such functions are deterministic discrete functions of time:

**Definition 2** (Join function) *The join function \(\lambda(t)\) is a deterministic discrete time function that returns the number of processes that invoke the join operation at time \(t\).*

\(^1\)Sometime also called **partially synchronous system**.
Definition 3 (Leave function) The leave function $\mu(t)$ is a deterministic discrete time function that returns the number of processes that have left the system at time $t$.

Note that the join and the leave functions cannot return a value different from 0 in the time units before $t_0$ (i.e. there is no churn before the initial time $t_0$). Therefore $\lambda(t) = \mu(t) = 0$ for each time $t$ smaller equal than $t_0$.

![Figure 1.2: Examples of Join and Leave functions](image)

(a) Time $t_0$: $\lambda(t_0) = 0, \mu(t_0) = 0$  (b) Time $t_1$: $\lambda(t_1) = 2, \mu(t_1) = 2$  (c) Time $t_2$: $\lambda(t_2) = 0, \mu(t_2) = 3$

In Figure 1.2 is shown an example of how it is possible to characterize the churn of the computation by using the join function and the leave function. Let $t_0$ be the starting time of the computation. At time $t_0$, no process joins or leaves the computation then $\lambda(t_0) = 0$ and $\mu(t_0) = 0$ (Figure 1.2(a)); therefore, it is possible to say that in $t_0$ the computation is composed by a set $\Pi_0$ of processes and the size of the computation is $n_0$ (i.e. $|\Pi_0| = n_0$). Moreover, for any time $t < t_0$ we say that $\lambda(t) = \mu(t) = 0$.

In Figure 1.2(b) it is possible to see an example where $\lambda(t_1) = 2$, meaning that two processes enter the computation, namely $p_{12}$ and $p_{13}$, and $\mu(t_1) = 2$ meaning that two processes depart from the computation, namely $p_4$ and $p_7$. Note that, in this case the number of processes inside the system is still $n_0$.

In Figure 1.2(c) it is possible to see another example where $\lambda(t_2) = 0$, meaning that no new process join the computation and $\mu(t_2) = 3$ meaning that three processes depart from the computation, namely $p_1$, $p_{10}$ and $p_{11}$. In this case, the number of processes inside the system is decreased. From this example, it is possible to see how the variation of the computation members can be easily described quantitatively by using the two function.

Note that, this way of modeling is able also to represent a “static system”: a system without failures is a system characterized by a join function and a leave function that are always equal to 0 for any time $t$ (i.e. $\lambda(t) = \mu(t) = 0 \forall t$) while a system with crash failures is a system characterized by a null join function and a leave function having the constraint that the total number of failure is smaller than $n_0$ (i.e. $\lambda(t) = 0 \forall t$ and $\sum_{\tau=t_0}^{\infty} \mu(\tau) < n_0$).

As soon as the dynamicity introduced by the joins and the leaves appears in the computation, it is possible to observe that the size of computation and its composition can change.
1.4. CHURN MODEL

The number of participants to the computation can be calculated depending on the values of $\lambda(t)$ and $\mu(t)$ as follows:

**Definition 4** (Node function) Let $n_0$ be the number of processes in the system at start time $t_0$. $N(t)$ is the number of processes within the system at time $t$ for every $t \geq t_0$ (i.e. $N(t) = N(t-1) + \lambda(t) - \mu(t)$, with $N(t_0) = n_0$).

With respect to the scenario shown in Figure 1.2, the corresponding node function is:

- $N(t_0) = n_0 = 11$
- $N(t_1) = N(t_0) + \lambda(t_1) - \mu(t_1) = n_0 + 2 - 2 = n_0 = 11$
- $N(t_2) = N(t_1) + \lambda(t_2) - \mu(t_2) = n_0 + 0 - 3 = n_0 - 3 = 8$.

**Composition of the System.** The variation of the computation size is not the only effect generated by the dynamicity introduced by the join and leave of nodes. The composition of participants is also affected. As an example consider a computation of size $n$, and a join function and a leave function that are constant and equal (i.e $\lambda(t_i) = \mu(t_i) = c$, $\forall t$ with $c \leq n \in \mathbb{N}$). In such a system, the dynamicity does not affect the network size, it affect only its composition. In order to capture this dynamicity effect, we define the *leave fraction* of the network in a given interval.

**Definition 5** (Leave Fraction) Let $n_0$ be the number of processes in the system at start time $t_0$, let $\lambda(t)$ the join function and $\mu(t)$ the leave function of the system. Let $\Delta t = [t, t + \Delta]$ be a time interval (with $\Delta \geq 0$). $L(\Delta t)$ is the fraction of processes that have been refreshed (i.e., that have left the system) during the interval $\Delta t$, i.e. $L(\Delta t) = \frac{\sum_{\tau=t}^{t+\Delta} \mu(\tau)}{N(t-1)}$ (with $N(\tau) = n_0$ for each $\tau \leq t_0$).

Informally, the leave fraction, measures how many processes do not belong to the computation for all the considered period. As an example, consider the scenario shown in Figure 1.2. The corresponding leave fraction, for each interval starting in $t_0$ is:

- $L([t_0, t_1]) = \frac{\mu(t_0) + \mu(t_1)}{N(t_0-1)} = \frac{2}{11}$
- $L([t_0, t_2]) = \frac{\mu(t_0) + \mu(t_1) + \mu(t_2)}{N(t_0-1)} = \frac{5}{11}$

This metric allows to state a simple lemma on the composition of the system

**Lemma 1** Let $\Delta t = [t, t + \Delta]$ be a time interval. If there exists a process $p$ that is in the system at time $t$ and does not leave for all the interval $\Delta t$ then $L(\Delta t) < 1$. 
1.4.2 Churn constraint on the upper and lower bounds on the network size

Based on the previous definitions, this section derives the constraint that a join function and a leave function have to satisfy in order the network size remains in a given interval. Note that, such kind of behavior is typical of real applications like peer-to-peer systems, VoIP based application etc... [25], [28].

Let $n_0$ be the number of processes inside the computation at the start time $t_0$ and $k_1, k_2$ be two positive integers such that $0 \leq k_1 < n_0$ and $k_2 \geq 0$. The aim is to model a computation affected by the dynamicity whose effect is the variation of the system size in an interval $\Delta N$ that is defined by the upper bound $n_0 + k_2$ and the lower bound $n_0 - k_1$ (i.e. $\Delta N = [n_0 - k_1, n_0 + k_2]$). An example of such computation is shown in Figure 1.3.

**Lemma 2** Let $k_1$ and $k_2$ be two integers such that $k_1, k_2 \geq 0$ and let $n_0$ be the number of processes in the system at the starting time $t_0$. Given the join and the leave function $\lambda(t)$ and $\mu(t)$, the node function $N(t)$ is always comprised inside the interval $\Delta N = [n_0 - k_1, n_0 + k_2]$ if and only if for each $t$ greater than $t_0$:

\[
\begin{align*}
(c1) & \sum_{\tau=t_0}^{t} \mu(\tau) \leq \sum_{\tau=t_0}^{t} \lambda(\tau) + k_1, \\
(c2) & \sum_{\tau=t_0}^{t} \mu(\tau) \geq \sum_{\tau=t_0}^{t} \lambda(\tau) - k_2.
\end{align*}
\]

**Proof** Due to Definition 4, we have that for each time $t$, $N(t) = N(t-1) + \lambda(t) - \mu(t)$ and iterating, we can express the node function as function of $n_0$ as $N(t) = n_0 + \sum_{\tau=t_0}^{t} \lambda(\tau) - \sum_{\tau=t_0}^{t} \mu(\tau)$. Constraining $N(t)$ to be in the interval $\Delta N$ we obtain, for
every $t$, for the lower bound

\begin{align*}
N(t) & \geq n_0 - k_1 \\
n_0 + \sum_{r=t_0}^{t} \lambda(\tau) - \sum_{r=t_0}^{t} \mu(\tau) & \geq n_0 - k_1 \\
\sum_{r=t_0}^{t} \mu(\tau) & \leq \sum_{r=t_0}^{t} \lambda(\tau) + k_1,
\end{align*}

and we obtain, for every $t$, for the upper bound

\begin{align*}
N(t) & \leq n_0 + k_2 \\
n_0 + \sum_{r=t_0}^{t} \lambda(\tau) - \sum_{r=t_0}^{t} \mu(\tau) & \leq n_0 + k_2 \\
\sum_{r=t_0}^{t} \mu(\tau) & \geq \sum_{r=t_0}^{t} \lambda(\tau) - k_2.
\end{align*}

□ Lemma 2

1.5 Related Work

In this section several approaches to model dynamic system are presented.

[1] and [46] are two of the first works where the assumption of having a finite and known number of process is relaxed and in particular, several classic problems in concurrent computing (i.e. election, mutual exclusion, and consensus) are considered by assuming that the number of processes which may participate is infinite. To solve such problems, the authors present a new model based on the concepts of concurrency level and required participation.

Considering that in each infinite run the number of processes taking part to the computation is unknown, the concurrency level defines the maximum number of processes that may participate to the algorithm at the same time. In particular, the authors define three types of concurrency level:

- **finite**: there is a finite and known bound on the maximum number of processes that participate simultaneously in the algorithm, over all the runs;

- **bounded**: (also called infinite arrival model) in each run, there is a finite bound on the number of processes that participate simultaneously to the algorithm (note that such bound can change between two different runs);

- **unbounded**: in each run, the number of processes that participate simultaneously to the algorithm can grow without bound.

Note that, the concept of process failure/leave is implicit. Moreover, the authors observe that, depending form the problem specification, all the processes or only a subset are required to participate to the algorithm (i.e. in the mutual exclusion problem is not required to all the processes to participate and a process can remain forever in the not critical section state). The participation is defined by providing a pair $[\ell, u]$ where $\ell$ is the minimum number of processes required from the algorithm and $u$ is
the maximum. Using such a model, the authors show the weakest combinations of concurrency level and required participation that make possible to implement (i) election, (ii) consensus, (iii) dead-lock free mutual exclusion, (iv) starvation free mutual exclusion and (v) test & set, by using atomic registers in a fault free model.

In [1], two variant of the bounded concurrency model are underlined:

- **infinite arrival model with b-bounded concurrency**: every run has a maximum concurrency bounded by a constant $b$ known by the algorithms;
- **infinite arrival model with bounded concurrency**: each run has a maximum finite concurrency.

Note that, in this two variants the departure of nodes is involved; in fact, having a maximum degree of concurrency implies that when such maximum concurrency is reached, arrival and departure rate become strictly related.

Comparing the model presented in Section 1.4 with the one described in [1, 46], it is possible to say that the model presented here is able to represent all the the three concurrency level; in particular:

- the finite concurrency level is obtained by having $\lambda(t) = \mu(t) = 0 \forall t$;
- the bounded concurrency level is obtained by having $k_2 \neq 0$ and $\lambda(t)$ and $\mu(t)$ satisfying the constraints of Lemma 2;
- the unbounded concurrency level is obtained by having $k_2$ unbounded and $\lambda(t)$ and $\mu(t)$ arbitrary.

With respect to the participation, the model assumed in this thesis consider $[0, \infty]$-participation with respect to the distributed computation and $[1, \infty]$-participation with respect to the distributed system. In fact, processes of the distributed system are not forced to join the distributed computation but if they decide to join, they are involved even if they do not execute any operations on the object.

Note that, the model presented here, is also able to provide quantitative information on the dynamicity useful to provide correctness bounds (e.g. providing bounds on the maximum number of leave operations tolerated by an algorithm to work safely).

In [5], (to the best of my knowledge) appears the first definition of what a dynamic system is. Trying to collect all the common aspects identified so far in literature, the authors provide the following definition:

*A dynamic system is a continually running system in which an arbitrarily large number of processes are part of the system during each interval of time and, at any*
time, any process can directly interact with only an arbitrary small part of the system.

Such definition is quite general and is applicable also to large scale system, i.e. peer-to-peer system. To this aim, the authors extend the models presented in [1, 46], by considering a second dynamicity dimension, i.e. the diameter of the network, to take into account the fact that in large scale networks, at any time each process has only a partial view of the system (i.e. it can directly interact only with a subset of processes called neighborhood). To model such behavior, the network is represented by a graph where each process is a vertex and given two processes $p_i$ and $p_j$ belonging to the system, there exists an edge between $p_i$ and $p_j$ if they are neighbors. The dynamicity is captured by adding a new vertex (and the related edges) each time that a process arrives and removing a vertex (and the related edges) each time that a process departs and the diameter of such a graph is considered as a dynamicity metric.

- **bounded and known diameter**: every run has a maximum diameter bounded by a constant $b$ known by the algorithms;

- **bounded and unknown diameter**: in each run the diameter is finite but the union of all the run may be unbounded;

- **unbounded diameter**: the diameter is unbounded and can grow indefinitely in the run.

Using this model, the authors show, through an illustrative example, a classes of applications (i.e. aggregation problems) where also the diameter of the network plays an important role.

Note that all the models presented above do not consider explicitly the distribution of the arrival and departure of processes but give only a qualitative measure of the dynamicity.

Recently a quantitative approach to model the churn (similar to the one proposed in this thesis) has been presented in [34]. Such framework is suited to describe systems where the number of processes participating is constant along time and is composed of two models (i) **train model** and (ii) **crowd model**.

1. In the train model, processes join and leave the computation at periodic time and in groups of the same size $c$ that is the same for all the computation (i.e. at each time $t = t_0 + i\Delta t$, $c$ processes join and $c$ processes leave, where $t_0$ is the time at which the computation starts, $\Delta$ is the period between two join/leave burst and $i \in \mathbb{N}$).
2. The crowd model is a generalization of the train model and it allows processes to join and leave the computation at arbitrary time instant in groups of the same size \( c_i \) that can change along the computation.

Note that, the churn model presented in Section 1.4 is able to model both the behavior described by the train model and the crowd model. Moreover, it is a further generalization of the crowd model. In fact, it makes possible to characterize not only systems with a constant dimension but also systems whose dimension change within a give range.
Chapter 2

Implementing a Regular Register in the Presence of Churn

A register is an object shared by a set of processes. Such an object provides processes with two operations, namely read() and write(), that allow them to read the value contained in the object or to modify such a value.

According to the value domain of the register, the set of processes that are allowed to read it, the ones that are allowed to write it, and the specification of which value is the value returned by a read operation, a family of types of registers can be defined. As far as the last point (the value returned by a read operation) is concerned, Lamport has defined three types of register [38]:

- A safe register can be written by one writer only. Moreover, a read operation on such a register returns its current value if no write operation is concurrent with that read. In case of concurrency, the read can return any value of the value domain of the register (which means that a read concurrent with a write can return a value that has never been written). This type of register is very weak.

- A regular register can have any number of writers and any number of readers [50]. The writes appear as if they were executed sequentially, complying with their real time order (i.e., if two writes \(w_1\) and \(w_2\) are concurrent they can appear in any order, but if \(w_1\) terminates before \(w_2\) starts, \(w_1\) has to appear as being executed before \(w_2\)). As far as a read operation is concerned we have the following. If no write operation is concurrent with a read operation, that read operation returns the current value kept in the register. Otherwise, the read operation returns any value written by a concurrent write operation or the last value of the register before these concurrent writes. A regular register is stronger than a safe register, as the value returned in presence of concurrent write operations is no longer arbitrary.

Nevertheless, a regular register can exhibit what is called a new/old inversion.
In Figure 2.1 are depicted two write operations $w_1$ and $w_2$ and two read operations $r_1$ and $r_2$ that are concurrent ($r_1$ is concurrent with $w_1$ and $w_2$, while $r_2$ is concurrent with $w_2$ only).

According to the definition of register regularity, it is possible that $r_1$ returns the value written by $w_2$ while $r_2$ returns the value written by $w_1$.

- An atomic register is a regular register without new/old inversion. This means that an atomic register is such that all its read and write operations appear as if they have been executed sequentially, this sequential total order respecting the real time order of the operations. (Linearizability [32] is nothing else than atomicity when we consider objects defined by a sequential specification).

Interestingly enough, safe, regular and atomic registers have the same computational power. This means that it is possible to implement a multi-writer/multi-reader atomic register from single-writer/single-reader safe registers. There is a huge number of papers in the literature discussing such transformations (e.g., [14, 30, 49, 51, 52] to cite a few).

In this chapter, a regular register is considered because its implementation is less expensive than atomic register although it has the same computational power.

First, two algorithms are presented implementing a single-writer/multi reader register; the first works under synchrony assumption while the second works in an eventually synchronous dynamic distributed system. Then the impossibility to build a regular register in presence of continuous churn is proved and finally, it is discussed which multi-writer specification is supported by the proposed solutions according to the ones provided in [50].

### 2.1 Related Work

Several works have been done recently with the aim to address the implementation of concurrent data structures on wired message passing dynamic systems (eg., [3], [6], [16], [24], [44]).

In [44], a Reconfigurable Atomic Memory for Basic Object (RAMBO) is presented. RAMBO works in a distributed system where processes can join and fail.
2.1. RELATED WORK

during the execution of the algorithm. To guarantee reliability of data, in spite of network changes, RAMBO replicates data at several network location and defines configuration to manage small and transient changes. Each configuration is composed by a set of members, a set of read-quorum and a set of write-quorums. In order to manage large changes to the set of participant process, RAMBO defines a reconfiguration procedure whose aim is to move from an existing configuration to a new one where the set of members, read-quorum or write-quorum are modified. In order to ensure atomicity, the reconfiguration procedure is implemented by a distributed consensus algorithm that makes all the processes agree on the same successive configurations. Therefore, RAMBO cannot be implemented in a fully asynchronous system. It is important to note that in RAMBO the notion of churn is abstracted by defining a sequence of configurations. Note that, RAMBO poses some constraints on the removal of old configurations and in particular, a certain configuration $C$ cannot be removed until each operation, executed by processes belonging $C$, is not ended; as a consequence, many old configurations may take long time to be removed.

In [24] and [16] are presented improvements to the original RAMBO protocol and in particular to the reconfiguration mechanism. In [24] the reconfiguration protocol has been changed by parallelize new configuration installation and the removal of an arbitrary number of old configurations. In [16], the authors present a mechanism that combine the features of RAMBO and the underlying consensus algorithm to speed up the reconfiguration and reduce the time during which old configurations are accessible.

In [3] Aguilera et al. show that an atomic R/W object can be realized without consensus and, thus, on a fully asynchronous distributed system provided that the number of reconfiguration operations is finite and thus the churn is quiescent (i.e. there exists a time $t$ after which no join and no failure occur). Configurations are managed by taking into account all the changes (i.e. join and failure of processes) suggested by the participant and the quorums are represented by any majority of processes. However, to ensure liveness of read and write operations, the authors assume that the number of reconfigurations is finite and that there is a majority of correct processes in each reconfiguration.

Note that, the assumption of quiescent churn is common to all the approaches described so far.

The model presented in this thesis considers, as in [3], the possibility to add and remove processes during the computation but differs from it because no assumption is done about the number of join and leave (i.e. reconfigurations) invoked, that can be possibly infinite. The main consequence is that, in this environment, it is not possible to implement a register in a fully asynchronous dynamic system (cfr. Section 2.4).
but only in an eventually synchronous one (and of course in a synchronous dynamic system - cfr. Section 2.5 and Section 2.3).

<table>
<thead>
<tr>
<th></th>
<th>Synchronous</th>
<th>Eventually Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quiescent Churn</td>
<td></td>
<td>e.g. [16], [24], [44] Usage of Consensus</td>
<td>e.g. [3] Majority of Correct ( \forall t )</td>
</tr>
<tr>
<td>Continuous Churn</td>
<td>[6] (cfr. Section 2.3) At least one active in every interval ([t, t + \delta])</td>
<td>[6] (cfr. Section 2.5) Majority of Active ( \forall t )</td>
<td>Impossibility [6] (cfr. Section 2.4)</td>
</tr>
</tbody>
</table>

Figure 2.2: Register Implementation in Dynamic Systems prone to Churn

In Figure 2.1 it is presented a summary of the main works done on the implementation of register objects in dynamic systems, with respect to the assumption made on the churn, and is highlighted the contribution of this thesis with respect to the state of the art.

Register abstractions have been considered also taking into account another type of dynamicity: the mobility.

In [19], an atomic memory implementation for mobile ah-hoc networks is presented. This implementation is a quorum based one and implements an atomic register in a region of the space where nodes can move. The authors abstract the mobility of nodes by introducing a static concept: the focal point. A focal point is a region where nodes can move and can have a state, depending on the number of nodes inside the region. The authors point out that, in order to implement this shared object, some additional constraints must be put on these focal points, e.g. only a fraction of focal points are allowed to fail. As a result, this model introduces some limitations on node motions and density over time.

In [55] these constraints are reduced and only a small set of mobility constraints are considered to implement the atomic register. In particular, the main novelty with respect to Dolev’s work is the possibility of tolerating unlimited number of focal point failures during the entire system lifetime. In fact, Tulone’s mobility constraints only require a minimum coverage of the mobile nodes without additional constraint on movements.

In this thesis, mobility aspects are not directly considered.
2.2 Single-writer/Multi-reader Regular Register Specification

The notion of a regular register, as specified by Lamport in [38], is not directly applicable in a dynamic distributed system because it does not consider the possibility that processes join or leave the computation and then it has to be adapted to dynamic systems. A protocol implements a regular register in a dynamic system if the following properties are satisfied.

- **Termination**: If a process invokes a read or a write operation and does not leave the system, it eventually returns from that operation.

- **Validity**: A read operation returns the last value written before the read invocation, or a value written by a write operation concurrent with it.

Moreover, it is assumed that a process invokes the read or write operation only after it has returned from its `join()` invocation.

2.3 Synchronous Systems

In this section it is assumed that the `join` function and the `leave` function satisfy the two constraints of Lemma 2 (i.e. the node distribution is always upper-bounded by a constant $n_0 + k_2$ and lower-bounded by a constant $n_0 - k_1$).

2.3.1 Protocol

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

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

- Two variables denoted `register_i` and `sn_i`; `register_i` contains the local copy of the regular register, while `sn_i` is the associated sequence number.

- A boolean `active_i`, initialized to `false`, that is switched to `true` just after $p_i$ has joined the system.

- Two set variables, denoted `replies_i` and `reply_to_i`, that are used in 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
Figure 2.3: The join() protocol for a synchronous system (code for $p_i$)

join period, while $\text{reply}_i$ contains the processes that are joining the system concurrently with $p_i$ (as far as $p_i$ knows).

Initially, $n_0$ processes compose the system. The local variables of each of these processes $p_k$ are such that $\text{register}_k$ contains the initial value of the regular register\(^1\), $sn_k = 0$, $active_k = false$, and $\text{replies}_k = \text{reply}_k = \emptyset$.

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 2.3, 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 $\delta$ time units (line 02); this waiting period is explained later. If $\text{register}_i$ has not been updated during this waiting period (line 03), $p_i$ broadcasts (with the $\text{broadcast}(\cdot)$ operation) an $\text{inquiry}(\cdot)$ message to the processes that are in the system (line 05) and waits for $2\delta$ time units, i.e., the maximum round trip delay (line 06). When this period

\(^1\)Without loss of generality, we assume that at the beginning every process $p_k$ has in its variable $\text{register}_k$ the value 0.

\(^2\)The statement $\text{wait}(2\delta)$ can be replaced by $\text{wait}(\delta + \delta')$, which provides a more efficient join operation; $\delta$ is the upper bound for the dissemination of the message sent by the reliable broadcast that is a one-to-many communication primitive, while $\delta'$ is the upper bound for a response that is sent to a process whose id is known, using a one-to-one communication primitive. So, $\text{wait}(\delta)$ is related to the broadcast, while $\text{wait}(\delta')$ is related to point-to-point communication. We use the $\text{wait}(2\delta)$ statement to make easier the presentation.
terminates, \( p \) updates its local variables \( \text{register}_i \) and \( \text{sn}_i \) to the most up-to-date values it has received (lines 07-08). Then, \( p \) becomes active (line 10), which means that it can answer the inquiries it has received from other processes, and does it if \( \text{reply}_j \neq \emptyset \) (line 11). Finally, \( p \) returns \( \text{ok} \) to indicate the end of the \( \text{join}() \) operation (line 12).

When a process \( p \) receives a message \( \text{inquiry}(j) \), it answers \( p \) by return sending back a \( \text{reply}(i, \text{register}_i, \text{sn}_i) \) message containing its local variable if it is active (line 14). Otherwise, \( p \) postpones its answer until it becomes active (line 15 and lines 10-11). Finally, when \( p \) receives a message \( \text{reply}(j, \text{value}, \text{sn}) \) from a process \( p_j \) it adds the corresponding 3-uple to its set \( \text{replies}_i \) (line 17).

The read() and write() operations. The algorithms for the read and write operations associated with the regular register are described in Figure 2.4. The read is purely local (i.e., fast): it consists in returning the current value of the local variable \( \text{register}_i \).

The write consists in disseminating the new value \( v \) (together with its sequence number) to all the processes that are currently in the system (line 02). In order to guarantee the correct delivery of that value, the writer is required to wait for \( \delta \) time units before terminating the write operation (line 03).

\[\begin{align*}
\text{operation read}(): & \quad \text{return} (\text{register}_i). \quad \% \text{issued by any process } p_i, \% \\
\text{operation write}(v): & \quad \% \text{issued only by the writer } p_w, \% \\
(01) & \quad \text{sn}_w \leftarrow \text{sn}_w + 1; \text{register}_w \leftarrow v; \\
(02) & \quad \text{broadcast write}(v, \text{sn}_w); \\
(03) & \quad \text{wait}(\delta); \text{return(\text{ok}).} \\
(04) & \quad \text{when write}(\text{value}, \text{sn}) \text{ is delivered: } \% \text{at any process } p_i, \% \\
(05) & \quad \text{if } (\text{sn} > \text{sn}_i) \text{ then register}_i \leftarrow \text{val}; \text{sn}_i \leftarrow \text{sn} \text{ end if.}
\end{align*}\]

Figure 2.4: The read() and write() protocols for a synchronous system.

Why the wait(\( \delta \)) statement at line 02 of the \( \text{join}() \) operation? To motivate the \text{wait}(\( \delta \)) statement at line 02, let us consider the execution of the \( \text{join}() \) operation depicted in Figure 2.5(a). At time \( t \), the processes \( p_i, p_h \) and \( p_k \) are the three processes composing the system, and \( p_i \) is the writer. Moreover, the process \( p_j \) executes \( \text{join}() \) just after \( t \). The value of the copies of the regular register is 0, while \( \text{register}_j = \perp \) (line 01). The ‘timely delivery” property of the broadcast invoked by the writer \( p_i \) ensures that \( p_h \) and \( p_k \) deliver the new value \( v = 1 \) by \( t + \delta \). But, as it entered the system after \( t \), there is no such a guarantee for \( p_j \). Hence, if \( p_j \) does not execute the \text{wait}(\( \delta \)) statement at line 02, its execution of the lines 03-09 can provide it with the previous value of the regular register, that is 0. If after obtaining 0, \( p_j \) 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 \( p_j \)).
The execution depicted in Figure 2.5(b) shows that this incorrect scenario cannot occur if \( p_j \) is forced to wait for \( \delta \) time units before inquiring to obtain the last value of the regular register.

### 2.3.2 Correctness

Correctness of the register has to be proved by showing liveness and safety of the implementation.

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

**Proof** The read() operation trivially terminates. The termination of the join() and write() operations follows from the fact that the wait() statement terminates. □Lemma 3

**Lemma 4** Let $t_0$ be the starting time of the system and let $n_0$ be the number of processes inside the system at time $t_0$. Let $\sum_{t=-3\delta+1}^{t} \mu(\tau) < N(t - 3\delta) \forall t$, then $|A(t)| > 0 \forall t$.

**Proof** At time $t_0$ the system is composed of $n_0$ processes and each of these process maintains the value of the register so we have that $|A(t_0)| = n_0$. By the definition of the join and leave distribution, we have that at time $t_0 + 1 A(t_0 + 1)$ processes start the join operation and $\mu(t_0 + 1)$ processes leave the system then $|A(t_0 + 1)| = n_0 - \mu(t_0 + 1)$. At time $t_0 + 2$ we have that $\lambda(t_0 + 2)$ more processes invoke the join and $\mu(t_0 + 2)$ more processes leave the system. In the worse case, all the processes leaving the system are active hence $|A(t_0 + 2)| \geq n_0 - \mu(t_0 + 1) - \mu(t_0 + 2)$. To be completed, a join operation needs, in the worse case, $3\delta$ time then until time $t_0 + 3\delta$ the number of the active processes always decrease and $|A(t_0 + 3\delta)| \geq n_0 - \sum_{t_0}^{t_0+3\delta} \mu(\tau)$. At time $t_0 + 3\delta + 1$, the joins invoked at time $t_0 + 1$ terminate and such joining processes become active so $|A(t_0 + 3\delta + 1)| \geq n_0 - \sum_{t_0+3\delta+1}^{t_0+3\delta+4} \mu(\tau) + \lambda(t_0 + 1)$. At time $t_0 + 3\delta + 2$, also the join operations invoked at time $t_0 + 2$ terminate and $\lambda(t_0 + 2)$ more processes become active. By iteration, for a given $t$, $|A(t)| \geq n_0 - \sum_{t_0}^{t} \mu(\tau) + \sum_{t_0+3\delta}^{t-3\delta} \lambda(\tau) = N(t - 3\delta) - \sum_{t=-3\delta+1}^{t} \mu(\tau)$. Moreover, as $\sum_{t=-3\delta+1}^{t} \mu(\tau) < N(t - 3\delta) \forall t$, we have that $|A(t)| > 0$ for any time $t$. □Lemma 4

**Lemma 5** Let $n_0$ be the number of processes inside the system at time $t_0$. Let $t \geq t_0$, let $\Delta t = [t, t + \Delta]$ a time interval and let $\sum_{t=-3\delta+1}^{t+\Delta} \mu(\tau) < N(t + \Delta - 3\delta) \forall t$ then $|A(\Delta t)| > 0 \forall t$.

**Proof** Considering that processes become active, in the worse case, $3\delta$ time after the invocation of their join and that in every time unit there is always at least one active process (due to Lemma 4), it follows that in every time interval there is always at least one active process. □Lemma 5

From this two Lemmas it is possible to derive a relation between the number of active processes and the refresh of the system.

**Corollary 1** Let $t \geq t_0$, let $\Delta t = [t, t + \Delta]$ a time interval.

$$|A(\Delta t)| > 0 \Leftrightarrow L(\Delta t) < 1.$$ 

**Proof** From Lemma 4 and Lemma 5 we know that $|A(\Delta t)| \geq N(t + \Delta - 3\delta) - \sum_{t=-3\delta+1}^{t+\Delta} \mu(\tau)$. In order to have at least one active process inside the system, we have to constraint such quantity to be greater than 0 and then
\[
N(t + \Delta - 3\delta) - \sum_{t=t+\Delta-3\delta+1}^{t+\Delta} \mu(\tau) > 0
\]
\[
N(t + \Delta - 3\delta) > \sum_{t=t+\Delta-3\delta+1}^{t+\Delta} \mu(\tau)
\]
\[
\frac{\sum_{t=t+\Delta-3\delta+1}^{t+\Delta} \mu(\tau)}{N(t+\Delta-3\delta)} < 1.
\]
that is exactly the definition of \( L(\Delta t) \).

\[\square \text{Corollary 1}\]

**Single-writer Validity.** The validity property for the single-writer case is proved in two steps: first it is shown that, at the end of the \( \text{join}() \) operation, every active process stores in its local variable the last value written or a value concurrently written and second it is shown that each \( \text{read}() \) operation returns the last value written or a value whose write is concurrent with the \( \text{read}() \).\

**Lemma 6** Let the join function \( \lambda(t) \), the leave function \( \mu(t) \) such that \( \sum_{t=t+3\delta}^{t+\Delta} \mu(\tau) < N(t) \) \( \forall t \geq t_0 \). When a process \( p_i \) terminates the execution of \( \text{join}() \), its local variable \( \text{register}_i \) contains the last value written in the regular register (i.e., the last value before the \( \text{join}() \) invocation), or a value whose write is concurrent with the \( \text{join}() \) operation.

**Proof** Let \( p_i \) be a process that issues a \( \text{join}() \) operation. It always executes the \( \text{wait}(\delta) \) statement at line 02. Then, there are two cases according to the value of the predicate \( \text{register}_i = \perp \) evaluated at line 03 of the join operation.

- **register\(_i \) \( \neq \perp \).** it is possible to conclude that \( p_i \) has received a \( \text{write}(< \text{val}, \text{sn} >) \) message and accordingly updated \( \text{register}_i \) line 05, Figure 2.4. As (1) the write operation lasts \( \delta \) time units (line 03 of the write operation, Figure 2.4), (2) the join operation lasts at least \( \delta \) time units, and (3) the message \( \text{write}(< \text{val}, \text{sn} >) \) - sent at the beginning of the write - takes at most \( \delta \) time units, it follows from \( \text{register}_i \neq \perp \) that the \( \text{join}() \) and the \( \text{write}() \) operations overlap, i.e., they are concurrent, which proves the lemma for that case.

- **register\(_i \) = \( \perp \).** In that case, \( p_i \) broadcasts an \( \text{inquery}(i) \) message and waits for \( 2\delta \) time units (lines 05-06 of the \( \text{join}() \) operation, Figure 2.3). Let \( t \) be the time at with \( p_i \) broadcasts the \( \text{inquery}(i) \) message. At the end of the \( 2\delta \) round trip upper bound delay, \( p_i \) updates \( \text{register}_i \) with the value associated with the highest sequence number it has received (lines 07-08). We consider two subcases.

- **Case 1:** No write is concurrent with the join operation. As \(|A[t, t+3\delta]| > 0 \) (Lemma 5), \( A[t, t+3\delta] \) is not empty. Consequently, at least one process that has a copy of the last written value answers the inquiry of \( p_i \) and consequently \( p_i \) sets \( \text{register}_i \) to that value by \( 2\delta \) time units after the broadcast, which proves the lemma.
- **Case 2:** There is (at least) one write issued by a process \( p_j \) concurrent with the join operation. In that case, \( p_i \) can receive both \text{wrr}(<\text{val}, \text{sn}>) messages and \text{reply}(<j, \text{val}, \text{sn}>) messages. According the values received at time \( t + 2\delta \), \( p_i \) will update \text{register}_i \) to the value written by a concurrent update, or the value written before the concurrent writes.

\( \Box \text{Lemma 6} \)

Now it is possible to prove the safety of the implementation of the regular register. Informally, the following lemma states that to guarantee safety, during the join of a process, the number of processes departing from the system has to be strictly lesser than the ones joining the system.

\textbf{Lemma 7 Validity.} \textit{Let the join function } \lambda(t) \textit{and the leave function } \mu(t) \textit{be such that } \forall t \geq t_0, \sum_{t+\delta}^{t+3\delta} \mu(\tau) < N(t) \textit{and consider the protocol described in Section 5. The read() operation returns the last value written before the read invocation, or a value written by a write() operation concurrent with it.}

\textbf{Proof} \ Let us observe that a read by any process can be invoked only after that process has terminated its join operation. It follows from that observation, Lemma 6, and the lines 04-05 associated with the write operation (Figure 2.4) that a read always returns a value that has been written.

Let us observe that a write that starts at time \( t \), terminates at time \( t + \delta \), and all the processes in \( A[t, t+\delta] \) have delivered the value it has written by \( t+\delta \). Considering that observation, the proof that a read operation returns the last written value or a value written by a concurrent write is similar to the proof of the previous lemma. \( \Box \text{Lemma 7} \)

\textbf{Theorem 1} \textit{Let the join function } \lambda(t) \textit{and the leave function } \mu(t) \textit{be such that } \forall t \geq t_0, \sum_{t+\delta}^{t+3\delta} \mu(\tau) < N(t) \textit{. The protocol described is the Figures 2.3 and 2.4 implements a regular register in a synchronous dynamic distributed system.}

\textbf{Proof} \ The proof follows directly from Lemmas 3 and 7. \( \Box \text{Theorem 1} \)

\section*{2.4 Asynchronous System}

This section shows that (not surprisingly) it is not possible to implement a regular register in a fully asynchronous dynamic system prone to continuous churn.

\subsection*{2.4.1 Impossibility Result}

\textbf{Theorem 2} \textit{If the churn is continuous then it is not possible to implement a regular register in a fully asynchronous dynamic system.}
Proof The proof is a consequence of the impossibility to implement a register in a fully asynchronous static message-passing system prone to any number of process crashes \([2]\). More specifically, in the churn model considered in this thesis, all the processes can leave at their will. As a consequence, due to the absence of an upper bound on message transfer delays, it is easy possible a run in which operations do not terminate or where the value obtained by a process is always older than the last value whose write is terminated.

Theorem 2

2.5 Eventually Synchronous Systems

2.5.1 Protocol

As consequence of the impossibility to know when the system behaves as synchronous, it is not possible to rely on the passage of time combined with a safe known bound on message transfer delay. Hence the protocol has to be based on acknowledgment messages.

Local variables at a process \(p_i\). Each process \(p_i\) has to manage local variables, where (as before) \(\text{register}_i\) denotes the local copy of the regular register. In order to ease the understanding and the presentation of the protocol, the control variables that have the same meaning as in the synchronous protocol are denoted with the same identifiers (but their management can differ). Those are the variables denoted \(\text{sn}_i\), \(\text{active}_i\), \(\text{replies}_i\) and \(\text{reply}_\text{to}_i\) in Section 2.3. In addition, the protocol uses the following variables.

- \(\text{read}_{\text{sn}}_i\) is a sequence number used by \(p_i\) to distinguish its successive read requests. The particular value \(\text{read}_{\text{sn}}_i = 0\) is used by the join operation.
- \(\text{reading}_i\) is boolean whose value is true when \(p_i\) is reading.
- \(\text{write}_{\text{ack}}_i\) is a set used by \(p_i\) (when it writes a new value) to remember the processes that have acknowledged its last write.
- \(\text{dl}_{\text{prev}}_i\) is a set where (while it is joining the system) \(p_i\) records the processes that have acknowledged its inquiry message while they were not yet active (so, these processes were joining the system too) or while they are reading. When it terminates its join operation, \(p_i\) has to send them a reply to prevent them to be blocked forever.

Remark. In the following, the processing associated with a message reception (if the message has been sent) or a message delivery (if the message has been broadcast)
2.5. EVENTUALLY SYNCHRONOUS SYSTEMS

<table>
<thead>
<tr>
<th>operation join():</th>
</tr>
</thead>
<tbody>
<tr>
<td>(01) register, ← ⊥; sn, ← −1; active, ← false; reading, ← false;</td>
</tr>
<tr>
<td>(02) replies, ← ∅; reply_to, ← ∅; write_ack, ← ∅;</td>
</tr>
<tr>
<td>(03) dl_prev, ← ∅; read_sn, ← 0;</td>
</tr>
<tr>
<td>(04) broadcast inquiry(i, read_sn);</td>
</tr>
<tr>
<td>(05) wait until(</td>
</tr>
<tr>
<td>(06) let &lt; id, val, sn &gt;∈ replies;</td>
</tr>
<tr>
<td>such that (V &lt; −, −, sn' &gt;∈ replies_i : sn ≥ sn');</td>
</tr>
<tr>
<td>(07) if (sn &gt; sn_i) then sn, ← sn; register, ← val end if</td>
</tr>
<tr>
<td>(08) active, ← true;</td>
</tr>
<tr>
<td>(09) for each &lt; j, r_sn &gt;∈ reply_to ∪ dl_prev, do</td>
</tr>
<tr>
<td>(10) do send reply (&lt; i, register, sn_i, r_sn &gt;) to p_j</td>
</tr>
<tr>
<td>(11) end for;</td>
</tr>
<tr>
<td>(12) return(ok).</td>
</tr>
</tbody>
</table>

(13) when inquiry(j, r_sn) is delivered: |
| (14) if (active) then |
| (15) send reply (< i, register, sn_i, r_sn >) to p_j |
| (16) if (reading) then |
| (17) send dl_prev (i, r_sn) to p_j |
| (18) end if; |
| (19) else reply_to, ← reply_to ∪ {< j, r_sn >}; |
| (20) send dl_prev (i, r_sn) to p_j |
| (21) end if. |

(22) when reply(< j, value, sn >, r_sn) is received: |
| (23) if ((r_sn = read_sn) then |
| (24) replies, ← replies, ∪ {< j, value, sn >}; |
| (25) send ack (i, r_sn) to p_j |
| (26) end if. |

(27) when dl_prev(i, r_sn) is received: |
| (28) dl_prev, ← dl_prev, ∪ {< j, r_sn >}. |

Figure 2.6: The join() protocol for an eventually synchronous system (code for p_i)

appears in the description of one of the operations join(), read(), or write(). But the sending (or broadcast) of the message that triggers this processing is not necessarily restricted to that operation.

The join() operation. The algorithm implementing this operation is described in Figure 2.6. (The read algorithm -Figure 2.7- can be seen as a simplified version of it.)

After having initialized its local variables, the process p_i broadcasts an inquiry(i, read_sn) message to inform the other processes that it enters the system and wants to obtain the value of the regular register (line 04, as indicated read_sn_i is then equal to 0). Then, after it has received “enough” replies (line 05), p_i proceeds as in the
synchronous protocol: it updates its local pair \((register_i, sn_i)\) (lines 06-07), becomes active (line 08), and sends a reply to the processes in the set \(reply_to_i\) (line 09-11). It sends such a reply message also to the processes in its set \(dl_prev_i\) to prevents them from waiting forever. In addition to the term \(<i, register_i, sn_i>\), a reply message sent to a process \(p_j\), from a process \(p_i\), has now to carry also the read sequence number \(r_sn\) that identifies the corresponding request issued by \(p_j\).

The behavior of \(p_i\) when it receives an \(inquiry(j, r_sn)\) message is similar to one of the synchronous protocol, with two differences. The first is that it always sends back a message to \(p_j\). It sends a \(reply()\) message if it is active (line 14 - 15), and a \(dl_prev()\) if it not active yet (line 20). The second difference is that, if \(p_i\) is reading, it also sends a \(dl_prev()\) message to \(p_j\) (line 16); this is required to have \(p_j\) sends to \(p_i\) the value it has obtained when it terminates its join operation.

When \(p_i\) receives a \(reply(<j, value, sn>, r_sn)\) message from a process \(p_j\), if the reply message is an answer to its \(inquiry(i, read_sn)\) message (line23), \(p_i\) adds \(<j, value, sn>\) to the set of replies it has received so far and sends back an \(ack(i, r_sn)\) message to \(p_j\) (line 24 - 25).

Finally, when \(p_i\) receives a message \(dl_prev(j, r_sn)\), it adds its content to the set \(dl_prev_i\) (line 28), in order to remember that it has to send a reply to \(p_j\) when it will become active (lines 09-10).

**The read() operation.** The read() and join() operations are strongly related. More specifically, a read is a simplified version of the join operation\(^3\). Hence, the code of the read() operation, described in Figure 2.7, is a simplified version of the code of the join() operation.

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

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

**The write() operation.** The code of the write operation is described in Figure 2.8.

When a process \(p_i\) wants to write, it issues first a read operation in order to obtain the sequence number associated with the last value written (line 01)\(^4\). Then, after it has broadcast the message \(wrrte(i, <v, sn_i>)\) to disseminate the new value and

---

\(^3\)As indicated before, the “read” identified \((i, 0)\) is the join() operation issued by \(p_i\).

\(^4\)As the write operations are not concurrent, this read obtains the greatest sequence number. The same strategy to obtain the last sequence number is used in protocols implementing an atomic registers (e.g., [2, 21]).
2.5. EVENTUALLY SYNCHRONOUS SYSTEMS

operation read(i):
(01) read_sn ← read_sn + 1;
(02) replies ← ∅; reading ← true;
(03) broadcast read(i, read_sn);
(04) wait until(|replies| > C);
(05) let <id, val, sn> ∈ replies,
    such that (∀ <−, −, sn'> ∈ replies : sn ≥ sn');
(06) if (sn > sn_i) then sn_i ← sn; register_i ← val end if;
(07) reading ← false; return(register_i).

(08) when read(j, r_sn) is delivered:
(09) if (active_i) then send reply (<i, register_i, sn_i, r_sn>) to p_j
(10) else reply_to_i ← reply_to_i ∪ {<j, r_sn>}
(11) end if.

Figure 2.7: The read() protocol for an eventually synchronous system (code for p_i)

its sequence number to the other processes (line 04), p_i waits until it has received “enough” acknowledgments. When this happens, it terminates the write operation by returning the control value ok (line 05).

When it is delivered a message write(j, <val, sn>), p_i takes into account the pair (val, sn) if it is more up-to-date than its current pair (line 08). In all cases, it sends back to the sender p_j a message ack (i, sn) to allow it to terminate its write operation (line 09).

When it receives a message ack (j, sn), p_i adds it to its set write_ack_i if this message is an answer to its last write (line 11).

operation write(v):
(01) read(i);
(02) sn_i ← sn_i + 1; register_i ← v;
(03) write_ack_i ← ∅;
(04) broadcast write(i, <v, sn_i>);
(05) wait until(|write_ack_i| > C);
(06) return(ok).

(07) when write(j, <val, sn>) is delivered:
(08) if (sn > sn_i) then register_i ← val; sn_i ← sn end if;
(09) send ack (i, sn) to p_j.

(10) when ack(j, sn) is received:
(11) if (sn = sn_i) then write_ack_i ← write_ack_i ∪ {j} end if.

Figure 2.8: The write() protocol for an eventually synchronous system (code for p_i)
2.5.2 Correctness

The structure of the proof is as follows. We first determine the *maximum* number of reply messages that a process has to wait for terminating a `join()` or `read()` operation (Lemma 8). Then, we compute the *minimum* number of reply messages that are necessary to ensure the validity property of the regular register (Lemma 9). Given these two thresholds, several values for the constant $C$ of the protocol presented in the previous section can be selected.

Lemma 10, Lemma 11 and Lemma 12 show respectively that, if at least $C$ processes are always active then any `join()`, `read()` or `write()` operation terminates. Moreover, Corollary 1 shows that the values of $n_0$, $k_1$ and $k_2$ cannot be chosen arbitrarily but they are linked by the following relation: $n_0 > 2k_1 + k_2$ Finally, Theorem 3 and Theorem 4 are associated with the termination and validity properties of a regular register, respectively.

**Lemma 8** Consider op be ether a `join()` or a `read()` operation issued on the regular register and consider $\lambda(t)$ and $\mu(t)$ be respectively the join and the leave distributions that keep the number of processes of the distributed system at any time in the range $[n_0 - k_1, n_0 + k_2]$ by satisfying the constraints of Lemma 2. The number of reply messages that a process has to wait to terminate the protocol of Figures 2.6 and 2.7 is at most $n_0 - k_1$ (i.e., $|\text{replies}_i| \leq n_0 - k_1$).

**Proof** Let consider the worse case scenario where the computation size decrease to $n_0 - k_1$ processes, and let $t$ be the time at which the computation reaches such minimum size and let $op$ be the first `join()` operation issued by a process $p_i$ after time $t$.

Let us suppose by contradiction that the number of required replies in the protocols of Figures 2.6 and 2.7 is greater than $n_0 - k_1$ and $op$ terminates. Therefore A `join()` operation terminates when $|\text{reply}_i| > n_0 - k_1$ (line 05, Figure 2.6). Initially empty, $\text{reply}_i$ is filled when a reply message is received (line 24, Figure 2.6). Such a message is sent by an active process $p_j$ to $p_i$ when an inquiry message is received (lines 10 and 15, Figure 2.6). The inquiry message is sent by $p_i$ at the beginning of the join operation and, for the broadcast property, it is deterministically delivered only by processes belonging to the system at that time. Considering that (i) at the time $t$ there are at least $n_0 - k_1$ processes in the computation (i.e. $|A(t)| \leq N(t)$) and (ii) $op$ is the first join after $t$, we have that at most $n_0 - k_1$ processes will receive the inquiry and generate the reply.

Hence $p_i$ will receive at most $n_0 - k_1$ replies remaining blocked. The same reasoning is applied to all the join operations issued after time $t$ then no more processes become active and all the join will be blocked waiting for ever.
Since a read is only a simplified version of a join, we can repeat the same reasoning also for read operations: all the read operations issued after time $t$ will wait for ever.

Note that the distributed system size can grow after time $t$ due to the churn but the set of active processes can only decrease (due to leave of processes).

**Lemma 9** Consider $\lambda(t)$ and $\mu(t)$ be respectively the join and the leave distributions that keep the number of processes of the distributed system at any time in the range $[n_0 - k_1, n_0 + k_2]$ by satisfying the constraints of Lemma 2. The number of replies in the protocols of Figures 2.6 - 2.8 necessary to ensure validity is at least $\lceil \frac{n_0 + k_2}{2} \rceil$ (i.e., $|\text{replies}| > \lceil \frac{n_0 + k_2}{2} \rceil$).

**Proof** Let us suppose by contradiction that the number of replies in the protocols of Figures 2.6 and 2.7 necessary to ensure validity is at most $\lceil \frac{n_0 + k_2}{2} \rceil$ and a read() operation $\text{op}$ returns the last value written or a value concurrently written.

Let $t'$ be the time at which the system reaches the maximum size $n_0 + k_2$ and let $\text{op}_r$ be a read operation issued at time $t'$ from a process $p_r$. Let $\text{op}_w$ be the last operation terminated before $p_r$. From the algorithm of Figure 2.8, $\text{op}_w$ terminates when $p_w$ has received at most $\lceil \frac{n_0 + k_2}{2} \rceil$ acknowledgment messages (line 05) meaning that at most $\lceil \frac{n_0 + k_2}{2} \rceil$ processes have received the value written and executed the update of the local copy of the register (line 08 Figure 2.8). In order to terminate the read operation $\text{op}_r$, $p_r$ needs to receive at most $\lceil \frac{n_0 + k_2}{2} \rceil$ replies (line 04 Figure 2.7). Considering that at the beginning of the read $n_0 + k_2$ processes belong to the computation and that at most $\lceil \frac{n_0 + k_2}{2} \rceil$ of them stores the new written value, it is possible that all the replies that $p_r$ will receives, come from the $\lceil \frac{n_0 + k_2}{2} \rceil$ processes that does not store the last value. Hence the read may return a value that is not the last one.

**Corollary 2** Consider $\lambda(t)$ and $\mu(t)$ be respectively the join and the leave distributions that keep the number of processes of the distributed system at any time in the range $[n_0 - k_1, n_0 + k_2]$ by satisfying the constraints of Lemma 2. In order to ensure both termination and validity in the algorithm in Figures 2.6 - 2.8 we have that $n_0 > 2k_1 + k_2$.

**Proof** The proof trivially follows by combining the two conditions on $|\text{replies}|$ stated in Lemmas 8 and 9.
and \(|replies_i| \leq \lceil \frac{n_0 + k_2}{2} \rceil\) can be satisfied by a number of values assumed by \(|replies_i|\). As an example consider the scenario where \(n_0 = 100, k_1 = 20\) and \(k_2 = 40\), we have that the two conditions are satisfied by any value of \(|replies_i|\) in the interval \([70, 80]\). In the following we consider \(C\) be one of such integers.

**Lemma 10** Consider \(\lambda(t)\) and \(\mu(t)\) be respectively the join and the leave distributions that keep the number of processes of the distributed system at any time in the range \([n_0 - k_1, n_0 + k_2]\) by satisfying the constraints of Lemma 2. Let \(n_0, k_1, \) and \(k_2\) be three integer such that \(n_0 > 2k_1 + k_2\). Let \(C\) be an integer such that \(C \leq n_0 - k_1\) and \(C > \lceil \frac{n_0 + k_2}{2} \rceil\). If \(\forall t : |A(t)| > C\), then any \(\text{join}()\) operation issued by a process \(p_i\) terminates, unless \(p_i\) leaves the system before the \(\text{join}()\) termination.

**Proof** Let us first observe that, in order to terminate its \(\text{join}()\) operation, a process \(p_i\) has to wait until its set \(replies_i\) contains at least \(C\) elements (line 05, Figure 2.6). Empty at the beginning of the join operation (line 01, Figure 2.6), this set is filled in by \(p_i\) when it receives the corresponding \(\text{REPLY}()\) messages (line 24 of Figure 2.6).

A process \(p_j\) sends a \(\text{REPLY}()\) message to \(p_i\) if one of the following two conditions is satisfied:

1. \(p_j\) is active and has received an \(\text{INQUIRY}()\) message from \(p_i\), (line 14, Figure 2.6), or 
2. \(p_j\) terminates its \(\text{join}()\) operation and \(<i,-> \in \text{reply}_\text{toj} \cup \text{dl}_\text{prev}_j\) (lines 09-10, Figure 2.6).

Let us suppose by contradiction that \(|replies_i|\) remains always smaller than \(C\). It follow that \(p_i\) does not receive enough \(\text{REPLY}()\) carrying the appropriate sequence number. Let \(t\) be the time at which the system becomes synchronous and let us consider a time \(t' > t\) at which a new process \(p_j\) invokes the join operation. At time \(t'\), \(p_j\) broadcasts an \(\text{INQUIRY}()\) message (line 04, Figure 2.6). As the system is synchronous from time \(t\) every process present in the system during \([t', t'+\delta]\) receives such \(\text{INQUIRY}()\) message by time \(t' + \delta\).

As \(p_i\) is not yet active when it receives \(p_j\)’s \(\text{INQUIRY}()\) message, \(p_i\) executes line 20 of Figure 2.6 and sends back a \(\text{DL}_\text{PREV}\) message to \(p_j\). Due to the assumption that every process that initiates a join operation remains in the distributed system at least \(3\delta\) time units, \(p_j\) receives \(p_i\)’s \(\text{DL}_\text{PREV}\) and executes consequently line 27 (Figure 2.6) adding \(<i,->\) to \(\text{dl}_\text{prev}_j\).

Due to the assumption that there are always at least \(C\) active processes in the computation, we have that at time \(t' + \delta\) at least \(C\) processes receive the \(\text{INQUIRY}()\) message of \(p_j\), and each of them will execute line 14 (Figure 2.6) and will send a \(\text{REPLY}()\) message to \(p_j\). Due to the synchrony of the system, \(p_j\) receives these messages
by time $t' + 2\delta$ and then stops waiting and becomes active (line 08, Figure 2.6). Consequently (lines 09-10) $p_j$ sends a reply to $p_i$ as $i \in \text{reply}_j \cup \text{dl}_\text{prev}_j$. By $\delta$ time units, $p_i$ receives that reply message and executes line 24, Figure 2.6. Due to churn rate, there are an infinity of processes invoking the join after time $t$ and $p_i$ will receive a reply from all of them so $p_i$ will fill in its set $\text{replies}_i$ and terminate its write operation.

Note that, if the churn is quiescent (i.e. it stops at some time $\bar{t}$) the proof still holds. In fact, by assumption, at time $\bar{t}$ at least $C$ processes are active and no one of them will leave the computation. It follows that there exist at least $C$ processes that will receive the inquiry messages related to pending join() operations (due to the broadcast property) and will reply.

Lemma 11 Consider $\lambda(t)$ and $\mu(t)$ be respectively the join and the leave distributions that keep the number of processes of the distributed system at any time in the range $[n_0 - k_1, n_0 + k_2]$ by satisfying the constraints of Lemma 2. Let $n_0$, $k_1$, and $k_2$ be three integer such that $n_0 > 2k_1 + k_2$. Let $C$ be an integer such that $C \leq n_0 - k_1$ and $C > \lceil \frac{n_0 + k_2}{2}\rceil$. If $\forall t : |A(t)| > C$, then any read() operation issued by a process $p_i$ terminates, unless $p_i$ leaves the system before the read() termination.

Proof The proof of the termination of the read() operation is the same as that of Lemma 10. In fact the read is a simplified case of the join algorithm in which the process $p_i$ that issues the read() operation is already active.

Lemma 12 Consider $\lambda(t)$ and $\mu(t)$ be, respectively, the join and the leave distributions that keep the number of processes of the distributed system at any time in the range $[n_0 - k_1, n_0 + k_2]$ by satisfying the constraints of Lemma 2. Let $n_0$, $k_1$, and $k_2$ be three integer such that $n_0 > 2k_1 + k_2$. Let $C$ be an integer such that $C \leq n_0 - k_1$ and $C > \lceil \frac{n_0 + k_2}{2}\rceil$. If $\forall t : |A(t)| > C$, and processes do not leave the system when executing a write() operation, then any write() operation invoked by a process $p_i$ terminates.

Proof Let us first assume that the read() operation invoked at line 01 terminates (this is proved in Lemma 11). Before terminating the write of a value $v$ with a sequence number $snb$ a process $p_i$ has to wait until its set $\text{write}\_\text{ack}_i$ contains at least $C$ elements (line 05, Figure 2.8). Empty at the beginning of the write operation (line 03), this set is filled in when the $\text{ack}(\_, snb)$ messages are delivered to $p_i$ (line 11). Such an ack message is sent by every process $p_j$ such that:

1. $p_j$ receives the corresponding write message from $p_i$ (line 09), or
2. $p_j$ receives a reply message for its join from $p_i$ (line 24, Figure 2.6).
Suppose by contradiction that \( p_i \) never fills in \( \text{write} \_\text{ack}_i \). This means that \( p_i \) misses \( \text{ack} \) messages carrying the sequence number \( \text{snb} \). Let us consider the time \( t \) at which the system becomes synchronous, i.e., every message sent by any process \( p_j \) at time \( t' > t \) is delivered by time \( t' + \delta \). Due to the assumption that the writer does not leave before the termination of its write, it follows that \( p_i \) will receive all the \( \text{inquiry} \) messages sent by processes joining after time \( t \). When it receives an \( \text{inquiry}() \) message from some joining process \( p_j \), \( p_i \) executes line 14 of Figure 2.6\(^5\) and sends a \( \text{reply} \) message to \( p_j \) with the sequence number \( \text{snb} \).

Since, after \( t \), the system is synchronous, \( p_j \) receives this \( \text{reply} \) message in at most \( \delta \) time units and executes line 24 of Figure 2.6 sending back an \( \text{ack}(-, \text{snb}) \) message to \( p_i \). Now let us consider that

1. by assumption a process that invokes a join operation does not leave the system for at least \( 3\delta \) time units, and

2. the system is now synchronous, such an \( \text{ack}(-, \text{snb}) \) message is received by \( p_i \) in at most \( \delta \) time units and consequently \( p_i \) executes line 11 and adds \( p_j \) to the set \( \text{write} \_\text{ack}_i \).

Due to the dynamicity of the system, processes never stop to join the distributed computation\(^6\). Due to the chain of messages \( \text{inquiry}(), \text{reply}(), \text{ack}() \), the reception of each message triggering the sending of the next one, it follows that \( p_i \) eventually receives at least \( C \text{ack}(-, \text{snb}) \) messages and terminates its write operation.

Note that, if the churn is quiescent (i.e. it stops at some time \( \bar{t} \)) the proof still holds. In fact, by assumption, at time \( \bar{t} \) at least \( C \) processes are active and no one of them will leave the computation. It follows that there exists at least \( C \) processes that will participate to the completion of the pending \( \text{join}() \) operations and thus they will contribute to the triggering of the message chain \( \text{inquiry}(), \text{reply}(), \text{ack}() \).

\[ \Box \text{Lemma 12} \]

**Theorem 3** Termination. Consider \( \lambda(t) \) and \( \mu(t) \) be, respectively, the join and the leave distributions that keep the number of processes of the distributed system at any time in the range \( [n_0 - k_1, n_0 + k_2] \) by satisfying the constraints of Lemma 2. Let \( n_0 \), \( k_1 \), and \( k_2 \) be three integer such that \( n_0 > 2k_1 + k_2 \). Let \( C \) be an integer such that \( C \leq n_0 - k_1 \) and \( C > \lceil \frac{n_0 + k_2}{2} \rceil \). If \( \forall t : |A(t)| > C \), then a process \( p_i \) that invokes either a \( \text{join}() \) or a \( \text{read}() \) or a \( \text{write}() \) operation and does not leave the system, \( p_i \) terminates its operation.

**Proof** The proof follows from Lemma 10, Lemma 11 and Lemma 12. \[ \Box \text{Theorem 3} \]

\(^5\)The writer process \( p_i \) executes line 14 of Figure 2.6 because a writer is always in the active mode.

\(^6\)It is important to recall that the churn model assumed here is not quiescent.
2.5. EVENTUALLY SYNCHRONOUS SYSTEMS

**Theorem 4** Validity. Consider $\lambda(t)$ and $\mu(t)$ be, respectively, the join and the leave distributions that keep the number of processes of the distributed system at any time in the range $[n_0 - k_1, n_0 + k_2]$ by satisfying the constraints of Lemma 2. Let $n_0$, $k_1$, and $k_2$ be three integer such that $n_0 > 2k_1 + k_2$. Let $C$ be an integer such that $C \leq n_0 - k_1$ and $C > \lceil \frac{n_0 + k_2}{2} \rceil$. If $\forall t: |A(t)| > C$, a read() operation returns either the last value written before the read invocation or a value written by a write() operation concurrent with the read() operation.

**Proof** Let $\text{write}_{\alpha}(v)$ be the $\alpha$-th write operation invoked on the register, and $W_{\alpha}(t)$ the set of processes that, at time $t$, store the value $v$ in their local copy of the register (to simplify the reasoning, and without loss of generality, it is assumed that no two write operations write the same value).

Let $t_0$ be the starting time of the computation. From the initialization statement, it follows that the $n_0$ processes, initially belonging to the computation, are active and store the initial value of the register (say $v_0$). It follows that $|A(t_0)| = |W_0(t_0)| = n_0$.

At time $t_1 = t_0 + 1$, $\mu(t_1)$ processes leave the distributed system and at most $\lambda(t_1)$ processes invoke the join() operation.

Note that, all the processes that leave were active at time $t_0$ and their local copy of the register contained $v_0$.

Since by assumption $\forall t: |A(t)| > C$, it follows that, in the worse case, at most one more process in $A(t_1)$ (and then also in $W_0(t_1)$) can leave before any process, signing up the computation, terminates its join() operation.

Let $p_i$ be the first process that terminates its join() operation and then becomes part of the computation. If no write() operation is concurrent with $p_i$’s join, then all of the replies received by $p_i$ come from processes in $W_0(t_1)$. Each of these processes stores the last value written (namely, the initial value $v_0$) in its local copy of the register together with the sequence number 0. When $p_i$ executes the lines 06-07 of the join() operation (Figure 2.6), it updates its local variable with the value $v_0$.

If there is a concurrent write() operation, $w = \text{write}_1(v_1)$, issued by an active process $p_j$, then there exists $\text{write}\_{\text{r}}(i, < v_1, 1 >)$ message broadcast by $p_j$. Due to the eventual delivery property of the broadcast primitive, such a message will be delivered eventually by processes belonging to the computation at time $t_B(w)$. It follows that before the end of this write, some active processes receive the $\text{write}\_{\text{r}}$ message and consequently update their local copy of the register with $v_1$ (line 05, Figure 2.8), while the other processes still keep the value $v_0$. Hence, at any time between the invocation and the end of write$_1(v_1)$, $|W_0(t)| = x$ and $|W_0(t)| = |A(t)| - x$, where $x$ is the number of processes that, at time $t$, have updated to $v_1$ their local copy of the register.

Due to the asynchrony of the system it is not possible to know when an active process replies an $\text{inquire}$ message: it can reply before receiving the $\text{write}\_{\text{r}}$ message or after. If all the processes reply before delivering the $\text{write}\_{\text{r}}$ message, then $p_i$ will return the last value written before the concurrent write $\text{write}_1(v)$. Otherwise, $p_i$ will
return a concurrently written value.

Note that, in order to terminate (say at time $t_E$), the $\text{write}_1(v_1)$ operation needs to receive at least $C+1$ $\text{ack}$ messages (line 03, Figure 2.8). Hence, we have $|W_1(t_E)| > C$.

By iterating the above reasoning, it follows that any $\text{join}()$ operation concurrent with $\text{write}_1(v_1)$ will return either the last written value or the value concurrently written, and any subsequent join will return the last written value.

Considering that (i) the same reasoning can be applied to all the following $\text{write}()$ operations and (ii) a $\text{read}()$ operation is just a simplified version of a $\text{join}()$, it follows that a $\text{read}()$ obtains the last value written if there are no concurrent write operations, or the last value written before the read invocation, or a value written by a write operation concurrent with it.

\( \square \text{Theorem 4} \)

We want to remark that, from Lemma 8 and Lemma 9, it follows that the constant $C$, defining the number of messages needed to ensure correctness of the algorithm, can be chosen between a range of values in the interval $[\lceil \frac{n_0+k}{2} \rceil, n_0-k_1]$ as soon as the values for $n_0$, $k_1$ and $k_2$ are such that $n_0 > 2k_1 + k_2$. In order to keep the message complexity of the algorithm as small as possible, it is preferable to consider $C$ as the smallest value in the interval (i.e., $C = \lceil \frac{n_0+k}{2} \rceil$).

### 2.6 Multi-writer/Multi-reader Regular Register Specification

In a multi-writer computation, processes may invoke $\text{write}()$ operations concurrently and then the validity property as specified by Lamport becomes ambiguous: which is the last value written before a $\text{read}()$ invocation if more than one $\text{write}()$ operation has been executed concurrently?

Some definitions for multi-writer regularity in static system (i.e. not subjected to churn) are given in [45] and in [50]. In the following, it is informally shown which of the three multi-writer validity conditions presented in [50] is satisfied by the implementations proposed in Section 2.3 and Section 2.5.

All regularity property considered, defines the value that can be returned by each $\text{read}()$ operation depending on the values written by the $\text{write}()$ operations that are relevant for the $\text{read}()$. A $\text{write}()$ operation is relevant for a $\text{read}()$ if it is concurrent or it precedes the $\text{read}()$ and does not exists any other $\text{write}()$ executed in the between.

- **Multi-Writer Regularity 1 (MWR1)** requires to each $\text{read}()$ operation $op$ to return any of the values written by some $\text{write}()$ that is relevant for $op$;
2.6. MULTI-WRITER/MULTI-READER REGULAR REGISTER
SPECIFICATION

- **Multi-Writer Regularity 2 (MWR2)** requires that, for each pair of `read()` operations, the `write()` operations that are relevant for both of them are perceived in the same order.

- **Multi-Writer Regularity 3 (MWR3)** requires that, given a `read()` operation `r`, if a process `p_i` sees a particular order of the relevant writes for `r`, all the processes have to agree upon this order.

Note that, MWR1 allows two `read()` operations `r_i` and `r_j`, issued respectively by processes `p_i` and `p_j` and having the same set of relevant writes, to behave independently; it follows that `p_i` and `p_j` may see a completely different order of the writes executed concurrently. Contrary, given two `read()` operations `r_i` and `r_j`, MWR2 and MWR3 constrain a particular order on the perception of writes operations relevant for both of them; therefore, any algorithm satisfying MWR2 and MWR3 should be based on some type of ordering able to distinguish concurrent operations.

**Synchronous System Protocol.** The protocol proposed in Section 2.3 satisfy the weakest condition for a multi-writer regular register (MWR1) and that it does not satisfy MWR2.

**Lemma 13** Let the join function $\lambda(t)$, the leave function $\mu(t)$ such that $\sum_{t=t_0}^{t_3+3\delta} \mu(\tau) < N(t) \forall t \geq t_0$ and consider the protocol described in Section 2.3. Every `read()` operation returns a value satisfying the MWR1 condition.

**Proof** (Sketch) It simply follows by Lemma 7 considering that the algorithms shown in Figures 2.3-2.4 cannot return a value overwritten by a most recent `write()`.

**Lemma 14** The protocol in Figures 2.3-2.4 does not satisfy the MWR2 condition.

**Proof** Consider the execution shown in Figure 2.9. It’s easy to see that it can be generated by the protocol proposed in Figures 2.3-2.4 but it does not satisfy MWR2. In fact, both the `write()` operations are relevant for both the `read()` but they are perceived in a different order by `p_j` and `p_h`.

**Eventually Synchronous System Protocol.** In the following it is shown that the protocol proposed in Section 2.5 satisfy the weakest condition for a multi-writer regular register (MWR1) and that it does not satisfy MWR2.

**Lemma 15** Let $n_0 > 2k_1 + k_2$ and let $C = \frac{n_0+k_2}{2} + 1$. Let assume that a process that invokes the `join()` operation remains in the system for at least $3\delta$ time units. If $\forall t : |A(t)| > \frac{n_0+k_2}{2}$ every `read()` operation satisfies MWR1.
**Proof** (Sketch) It simply follows from Theorem 4 considering that the algorithms cannot return a value overwritten by a most recent write().

**Lemma 15**

**Lemma 16** The protocol shown in Figures 2.6-2.8 is not able to satisfy the MWR2 condition.

**Proof** Consider the execution shown in Figure 2.10. It’s easy to see that it can be generated by the protocol proposed shown in Figures 2.6-2.8 but it does not satisfy MWR2. In fact, both the write() operations are relevant for both the read() but they are perceived in a different order by \( p_j \) and \( p_h \).

**2.6.1 Discussion**

In this chapter, has been proposed two protocols implementing a regular register in a dynamic distributed system where the number of processes of the computation changes in a given range: the first for a synchronous system and the second for an
2.6. MULTI-WRITER/MULTI-READER REGULAR REGISTER SPECIFICATION

An eventually synchronous system.

In a synchronous system, to propagate the value of the register along time, it is only required that at least one process is active in every time unit (i.e. \(|A(t)| \geq 1 \forall t\)). Note that such active process can change over time: the protocol only requires that at least an active process participates to the computation and it does not require that such a process is always the same.

Another interesting point of the proposed implementation, in a synchronous system, is that it does not give any constraint on the value of \(k_2\). Therefore, in a synchronous system it is possible to run this algorithm independently from the size variation (cfr. Section 2.3.2).

Note that, in the synchronous protocol, read operations are executed locally because the implementation relies on the upper bound on the communication delays. Moving to an eventually synchronous system, this implementation cannot work due to the unpredictability of such delays in the asynchrony periods. Hence, the price to pay for loosing synchrony is that read operations require to gather information from a certain number of active processes in the computation to retrieve a value satisfying the regularity property required (i.e. safety).

Due to the churn model, if the number of replies that a process should attend to terminate an operation is not properly selected, it could happen that the number of active processes is always below this number, therefore to avoid that the algorithm blocks, the number of active processes in the system has to be greater than \(\lceil \frac{n_0 + k_2}{2} \rceil\) and each read/write operation gather information from \(\lceil \frac{n_0 + k_1}{2} \rceil\) before executing the actual operation. This ensures that there is always an active process that keeps the latest value of the register during asynchrony periods. For example a write updates the value of \(\lceil \frac{n_0 + k_2}{2} \rceil\) active processes and a successive read gathering the information from \(\lceil \frac{n_0 + k_1}{2} \rceil\) processes, is always able to retrieve the latest written value.

Note that, in an eventually synchronous dynamic system the value of \(k_1\) and \(k_2\) cannot be chosen arbitrarily but they are constrained by the result of Corollary 2. Moreover, considering the particular case where the network size does not change (i.e. \(k_1 = k_2 = 0\) and \(\lambda(t) = \mu(t) = n_0\)) it is possible to obtain the numerical bound on the churn that can be tolerated by the protocol as shown in [6].
Chapter 3

Implementing a Set Object in the Presence of Churn

In this chapter, the notion of *Set Object* is introduced. A set can be seen as a collection of homogeneous data; it is generally not ordered and each element of the domain, that the set can collect, appears at most once in the set.

Each of the processes sharing the set $S$, can acquire its content through a *get* operation while it can add (remove) an element to $S$ through an *add (remove)* operation. Note that a set is a very useful abstraction in many modern applicative contexts. As an example, let us consider a collaborative intrusion detection system formed by several processes each one belonging to a distinct organization. A process adds an IP address to the set if it suspects that IP address (for instance, to do port scanning on some edge host of the organization). A process can remove an IP address if the suspicion reveals to be wrong [40, 41].

Due to the possible concurrency in the accesses to the set, a certain degree of consistency is required. Linearizability [32] requires that, given a set of operations issued on a shared object, all the processes perceives such operations in the same order. However, linearizable implementations are generally expensive. Thus, considering the semantics of the set object, in Section 3.2.2 a new consistency criteria, namely *value-based sequential consistency*, is introduced to guarantee consistency of *get()* operations. Value-based sequential consistency allows concurrent *get()* operations to return the same set in absence of concurrent *add()* and *remove()* operations. This is actually due to the fact that 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 each process. Note that, such a consistency criteria is weaker that linearizability [32] and sequential consistency. In fact, it only requires a total order among *add()* and *remove()* operations that modify the same value.
A value-based sequential consistent implementation returns a blacklist when a \texttt{get} is invoked and concurrent \texttt{get} operations will always return the same blacklist (in the absence concurrent \texttt{add/} \texttt{remove} operations).

In this Chapter, it is shown under which constraints on the churn and the accesses it is possible to implement a set object. In particular, in Section 3.3 it is shown that it is possible to implement a set object in a synchronous distributed system prone to continuous accesses and churn if processes have only a finite memory for local computation while in Section 3.4 it is shown that any implementation exists under the same assumptions moving to an eventually synchronous system.

Due to such impossibility, the semantic of the set object is weakened and a \(k\)-bounded set is introduced (Section 3.5). Informally, a \(k\)-bounded set object is a set that has limited memory of the execution history. In particular, given a \texttt{get()} operation, a \(k\)-bounded set behaves as a set where the execution history is limited only to the \(k\) most recent update operations. In this way, \(k\) can be used as a parameter to trade in-memory availability on the node used to implement the object with the need to approximate precisely a set object.

Although properties of a \(k\)-bounded set are weak, this object can be used to build some interesting abstractions, namely eventual participant detectors and shared lists.

### 3.1 Related Work

A weaker notion of set, namely \textit{weak-set}, has been introduced by Delporte-Gallet and Fouconnier in [18] to solve consensus in anonymous shared memory systems.

A weak set is a shared object that can be accessed by processes by means of two operations: \texttt{add()} and \texttt{get()} and it represents a restricted form of set object considered in this chapter, that does not include \texttt{remove()} operations.

The authors show how it is possible to implement a weak-set in a system with no churn\(^1\), by using a \textit{finite number of atomic registers} in two particular cases:

1. the set of values that can be added to the weak-set is bounded;
2. the set of processes sharing the weak-set is bounded and known by any process.

In the first case, the idea of the algorithm, implementing the weak-set, is to assign to each possible value a boolean multi-writer/multi-reader register. When a process wants to add the value \(v\), it writes \textit{true} in the correspondent register and when it wants to invoke a \texttt{get()} operation, it first reads from all the registers and then returns all the values such that the corresponding register returns \textit{true}.

\(^1\)The system model considered in [18] is asynchronous, prone to crash failures and enriched with an \(\Omega\) failure detector.
In the second case, similarly to the first one, a single-writer/multi-reader register is assigned to each process. When a process wants to add a value $v$, it simply write $v$ in its register and when it wants to invoke a get() operation, it reads all the register and returns the set of all the read values.

The system model considered in this thesis assumes (i) the domain of the set is finite but not bounded and (ii) the presence of continuous churn. These assumptions make no possible to extend the implementation suggested in [18] using a finite number of registers. Considering the first implementation and considering that the set domain is arbitrary, it is not possible to know, in each run, how many registers should be allocated. Considering the second implementation, due to the continuous churn, the number of processes involved in the whole computation can be infinite requiring a global unbounded number of registers to be associated to each process. Moreover, even considering at each time unit a bounded number of active processes, it could be not possible to safely re-assign registers belonging to left processes to new ones, due to the continuous refresh of nodes. This motivates the choice to look for a message-passing based implementation, like the ones shown in this Chapter.

Moreover, as pointed out by Delporte-Gallet and Fauconnier, a weak-set, due to the semantic of the object itself, is not necessarily linearizable [32]. More precisely, a get() operation does not care about the execution order of two concurrent add() operations that happened before it, 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 notion of set can be also related to the one of tuple space. A tuple space [23] is a shared memory object where generic data structures, called tuples, are stored and retrieved. A tuple space is defined by three operations: out($t$) which outputs the entry $t$ in the tuple space (i.e. write), in($\bar{t}$) which removes the tuple that matches with $\bar{t}$ from the tuple space (i.e. destructive read) and rd($\bar{t}$) that is similar to the previous one but without removing the tuple. A set object, as presented in this thesis, is something different from a tuple space. In fact, even if add($v$) and remove($v$) can be seen as a particular case of in($\bar{t}$) and out($t$) (i.e. where the tuple is composed only by one single value), the set object differs from a tuple space for its ability to return the complete content of the object without any parameter (while in a tuple space the rd($\bar{t}$) operation needs a tuple $\bar{t}$ as parameter to be matched).

3.2 Set Object $S$

A set object $S$ is a shared object used to store values. Without loss of generality, we assume that (i) $S$ contains only values taken from a finite domain (i.e. a subset of integers) and (ii) at the beginning of the computation $S$ is empty. A set $S$ 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. Given a set object $S$, any process can add, remove or get values to/from the set.

More precisely:

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

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

- The *get* operation, denoted $\texttt{get}()$, takes no input parameter. It returns a set containing the current content of $S$. 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 defined. In the following the notion of concurrency among operations is defined and the behavior of the $\texttt{get}()$ operation in case of concurrency is specified.

### 3.2.1 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 operations $op$ and $op'$, having respectively invocation times $t_B(op)$ and $t_B(op')$ and return times $t_E(op)$ and $t_E(op')$, we say that $op$ *precedes* $op'$ ($op < op'$) iff $t_E(op) < t_B(op')$. If $op$ does not precede $op'$ and $op'$ does not precede $op$ then they are *concurrent* ($op||op'$).

Considering all the operations invoked on the set object and the precedence relation introduced so far, it is possible to define the history of the computation as a partial order between the operations induced by the precedence relation. More formally an *execution history* can be defined as follow:

**Definition 6 (Execution History)** Let $H$ be the set of all the operations issued on the set object $S$. An execution history $H = (H, <)$ is a partial order on $H$ satisfying the relation $\prec$. 
3.2. SET OBJECT S

Note that, a get() operation does not modify the content of the set. Therefore, in order to define which are the admissible values that can be returned by each get() operation, it is possible to consider the sub-history of the execution containing only the operations that update the set (i.e. add() and remove() operations) and the considered get() instead of the entire history $\hat{H}$. To this aim let us introduce the concept of update sub-history induced by a get() operation.

**Definition 7 (Update sub-history induced by a get() operation)** Let $\hat{H} = (H, <)$ be the execution history of all the add(v), remove(v) and get() operations invoked on the shared object. The update sub-history of $\hat{H}$ induced by a get() operation $op$, denoted as $\hat{U}_{\hat{H}, op} = (U, <)$, is defined as follows:

- $U \subseteq H$;
- $U = \{ o \in H | (o = \text{add}(v) \lor o = \text{remove}(v)) \land t_B(o) < t_E(op) \} \cup \{ op \}$;
- for each pair of operations $o, o'$ such that $o, o' \in U$, if $o$ precedes $o'$ in $\hat{H}$ then $o$ precedes $o'$ in $\hat{U}_{\hat{H}, op}$.

As an example, in Figure 3.2 is shown the updated sub-history induced by the get() operation $op$ issued by $p_j$ on the history $\hat{H}$ (Figure 3.1).

In the following, when it is clear from the context the execution history $\hat{H}$ and the operation $op$ to which the updated sub-history refers to, the notation $\hat{U}$ is used instead of $\hat{U}_{\hat{H}, op}$.

![Figure 3.1: Execution History $\hat{H}$ of a set object.](image)

![Figure 3.2: Update sub-history $\hat{U}_{\hat{H}, op}$ of a set object.](image)
\[ \Pi_{\hat{H}} = \{ \pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6, \pi_7, \pi_8, \pi_9, \pi_{10}, \pi_{11}, \pi_{12} \} \]

\[
\pi_1 = (\text{add}(1)i, \text{get}(k), \text{add}(2)j, \text{remove}(2)i, \text{remove}(1)k, \text{get}(j), \text{add}(3)i) \\
\pi_2 = (\text{add}(1)i, \text{get}(k), \text{add}(2)j, \text{remove}(2)i, \text{remove}(1)k, \text{add}(3)i, \text{get}(j)) \\
\pi_3 = (\text{add}(1)i, \text{get}(k), \text{add}(2)j, \text{remove}(2)i, \text{get}(j), \text{remove}(1)k, \text{add}(3)i) \\
\pi_4 = (\text{add}(1)i, \text{get}(k), \text{add}(2)j, \text{remove}(2)i, \text{get}(j), \text{add}(3)i, \text{remove}(1)k) \\
\pi_5 = (\text{add}(1)i, \text{get}(k), \text{add}(2)j, \text{remove}(2)i, \text{add}(3)i), \text{get}(j), \text{remove}(1)k) \\
\pi_6 = (\text{add}(1)i, \text{get}(k), \text{add}(2)j, \text{remove}(2)i, \text{add}(3)i), \text{get}(j), \text{remove}(1)k, \text{get}(j) \\
\pi_7 = (\text{add}(1)i, \text{get}(k), \text{remove}(2)j, \text{add}(2)i, \text{get}(j), \text{add}(3)i, \text{remove}(1)k) \\
\pi_8 = (\text{add}(1)i, \text{get}(k), \text{remove}(2)j, \text{add}(2)i, \text{get}(j), \text{remove}(1)k, \text{add}(3)i) \\
\pi_9 = (\text{add}(1)i, \text{get}(k), \text{remove}(2)j, \text{add}(2)i, \text{add}(3)i), \text{get}(j), \text{remove}(1)k) \\
\pi_{10} = (\text{add}(1)i, \text{get}(k), \text{remove}(2)j, \text{add}(2)i, \text{add}(3)i), \text{get}(j), \text{remove}(1)k, \text{get}(j) \\
\pi_{11} = (\text{add}(1)i, \text{get}(k), \text{remove}(2)j, \text{add}(2)i, \text{remove}(1)k, \text{add}(3)i), \text{get}(j) \\
\pi_{12} = (\text{add}(1)i, \text{get}(k), \text{remove}(2)j, \text{add}(2)i, \text{remove}(1)k, \text{get}(j), \text{add}(3)i) \}
\]

\[ \hat{H} \]

Figure 3.3: Permutation Set of the Execution History \( \hat{H} \)

Note that, for a given execution history containing concurrent operations, it is possible to find several linearizations where concurrent operations are ordered differently. In the following we introduce the concepts of consistent permutation to identify each of these linearization and permutation set to identify all the consistent permutations following from a given execution history.

**Definition 8 (Permutation \( \pi \) Consistent with \( \hat{H} \))** Given an execution history \( \hat{H} = (H, \prec) \), a permutation \( \pi \) of all the operations belonging to \( H \) is consistent with \( \hat{H} \) if, for any pair of operations \( op, op' \) in \( \pi \), \( op \) precedes \( op' \) in \( \pi \) whenever \( op \) precedes \( op' \) in \( \hat{H} \).

As an example, consider the execution history \( \hat{H} \) shown in Figure 3.1. One of the possible permutations consistent with \( \hat{H} \) is \( \pi = (\text{add}(1)i, \text{get}(k), \text{add}(2)j, \text{remove}(2)i, \text{remove}(1)k, \text{get}(j), \text{add}(3)i)^2 \).

In the following the notation \( op \to_{\pi} op' \) is used to represent the precedence relation in the permutation \( \pi \) (i.e., the operation \( op \) precedes the operation \( op' \) in the sequence given by the permutation \( \pi \)).

**Definition 9 (Permutation set)** Let \( \hat{H} = (H, \prec) \) be the execution history of the set object. A permutation set of an execution history \( \hat{H} \), denoted as \( \Pi_{\hat{H}} \), is the set of all the permutations \( \pi \) that are consistent with \( \hat{H} \).

As an example, consider the execution history \( \hat{H} \) shown in Figure 3.1. The permutation set \( \Pi_{\hat{H}} \) of the execution history \( \hat{H} \) is shown in Figure 3.3.

\[ \text{With the notation } op(id) \text{ it is represented the operation } op \text{ issued by the process with identifier id} \]
3.2. SET OBJECT $S$

Note that if $op \rightarrow_{\pi_i} op'$ for some $\pi_i \in \Pi_{\hat{H}}$, it does not follow that $op < op'$ in $\hat{H}$. Contrary, if $op \rightarrow_{\pi_i} op' \forall \pi_i \in \Pi_{\hat{H}}$ then $op < op'$ in $\hat{H}$.

Based on the concepts introduced above, we are now in the position to defined what is an admissible set for a given get() operation $op$.

**Definition 10 (Set Generated by a Permutation)** Let $\hat{H} = (H, <)$ be an execution history and let $op$ be a get() operation of $H$. Given the update sub-history $\hat{U}_{\hat{H},op} = (U, <)$ induced by $op$ on $\hat{H}$, let $\pi$ be a permutation consistent with $\hat{U}_{\hat{H},op}$, then the set of values $V$ generated by $\pi$ for $op$ contains all the values $v$ such that:

1. $\exists \text{add}(v) \in \pi : \text{add}(v) \rightarrow_{\pi} op$ and
2. $\not\exists \text{remove}(v) \in \pi : \text{add}(v) \rightarrow_{\pi} \text{remove}(v) \rightarrow_{\pi} op$

As an example, consider the execution history $\hat{H}$ shown in Figure 3.1 and its update sub-history $\hat{U}$ induced by the operation $op$ = get(j) (Figure 3.2). Given a permutation $\pi_1 = (\text{add}(1)i, \text{add}(2)j, \text{remove}(2)i, \text{remove}(1)k, \text{get}(j)$) (belonging to $\Pi_{\hat{U}}$ consistent with $\hat{U}$), the set of values $V$ generated by $\pi_1$ is $V = \emptyset$.

**Definition 11 (Admissible set for a get() operation)** Given an execution history $\hat{H} = (H, <)$ of a set object $S$, let $op$ be a get() operation of $H$. Let $\hat{U}_{\hat{H},op}$ be the update sub-history induced by $op$ on $\hat{H}$. An admissible set for $op$, denoted as $V_{ad}(op)$, is any set generated by any permutation $\pi$ belonging to the permutation set of $\Pi_{\hat{U}}$.

As an example, consider the execution history $\hat{H}$ of Figure 3.1 and consider the get() operation $op$ issued by $p_j$. Given the permutation set $\Pi_{\hat{U}}$, all the possible admissible sets for $op$ are: $V_{ad1} = \emptyset$, $V_{ad2} = \{3\}$, $V_{ad3} = \{1\}$, $V_{ad4} = \{1\}$, $V_{ad5} = \{1, 3\}$, $V_{ad6} = \{3\}$, $V_{ad7} = \{1, 2\}$, $V_{ad8} = \{1, 2\}$, $V_{ad9} = \{1, 2, 3\}$, $V_{ad10} = \{2, 3\}$, $V_{ad11} = \{2, 3\}$, $V_{ad12} = \{2\}$.

### 3.2.2 Value-based Sequential Consistency Condition

A consistency condition defines which are the values that a get() operation is allowed to return. In a shared memory context, a set of formal consistency conditions has been defined [47] as constraints on the partial order of read() and write() operations issued on the shared memory. In order to specify a condition for a set object, we introduce the concept linear extension of an history.

**Definition 12 (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) $op_1 < op_2 \Rightarrow op_1 \rightarrow_s op_2$ and (iii) $\rightarrow_s$ is a total order.
Let us introduce now the notion of value-based sequential consistency for a set object. Informally, this consistency condition requires that any group of 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. add(v) and add(v')), these operations can be perceived in different order by different processes. More formally we have the following.

**Definition 13 (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 \( \hat{S}_i = (S, \rightarrow_s) \) such that for any pair of concurrent operations op and op', where op is an add(v), op' is a remove(v'), and \( v = v' \), if \( op \rightarrow_s op' \) for some \( p_i \) then \( op \rightarrow_s op' \) for any other process \( p_j \).

As an example, consider the execution history \( \hat{H} \) shown in Figure 3.1. Given the three processes \( p_i, p_j \) and \( p_k \), the linear extensions that let \( \hat{H} \) be value-based sequential consistent are:

- \( \hat{S}_i = \text{add}(1)_i, \text{add}(2)_j, \text{remove}(2)_i, \text{remove}(1)_k, \text{add}(3)_i, \text{get()} \)
- \( \hat{S}_j = \text{add}(1)_i, \text{add}(2)_j, \text{remove}(2)_i, \text{remove}(1)_k, \text{add}(3)_i, \text{get()} \)
- \( \hat{S}_k = \text{add}(1)_i, \text{add}(2)_j, \text{remove}(2)_i, \text{add}(3)_i, \text{remove}(1)_k, \text{get()} \)

**Relations among consistency criteria.**

Value-based sequential consistency is weaker than linearizability and sequential consistency but it is stronger than causal consistency.

Given an execution history, it is **sequential consistent** if it admits a linear extension in which all the get() operations returns an admissible set. In particular, if the provided linear extension is ordered accordingly to the real time execution, the history is said **linearizable**. Thus, in order to satisfy such criteria all the processes need to agree on a total order of operations. Contrary, value-based sequential consistency still allows to have a partial order of the operations, with the constraint that only operations occurring on the same value have to be totally ordered.

Contrary, causal consistency [4] requires only that each get() operation returns an admissible set with respect to some linear extensions. As an example, considering the execution history \( \hat{H} \) shown in Figure 3.1, the following linear extensions are causal consistent but not value-based sequential consistent.

- \( \hat{S}_i = \text{add}(1)_i, \text{remove}(2)_i, \text{add}(2)_j, \text{remove}(1)_k, \text{add}(3)_i, \text{get()} \)
- \( \hat{S}_j = \text{add}(1)_i, \text{add}(2)_j, \text{remove}(2)_i, \text{remove}(1)_k, \text{add}(3)_i, \text{get()} \)
- \( \hat{S}_k = \text{add}(1)_i, \text{add}(2)_j, \text{remove}(2)_i, \text{add}(3)_i, \text{remove}(1)_k, \text{get()} \)
3.2. SET OBJECT S

In Figure 3.4 are represented the relations among the four consistency criteria.

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. 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 causal consistency. 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.

3.2.3 Admissible set at time \( t \)

**Definition 14 (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 \( t_\theta(op) \leq t \) then \( op \in H_t \).

As an example, consider the history \( \hat{H} \) depicted in Figure 3.5. 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 \( \text{add}(4)i, \text{get}(i), \text{get}(j), \text{remove}(4)j, \text{add}(1)j, \text{add}(3)k \) and \( \text{remove}(3)k) \).
Definition 15 (Admissible Sets of values at time t) An admissible set of values at time \( t \) for \( S \) (denoted \( V_{ad}(t) \)) is any possible admissible set \( V_{ad}(op) \) for an instantaneous\(^3\) \( get() \) operation \( op \) that would be executed at time \( t \).

As an example, consider the execution of Figure 3.5. The possible admissible sets at time \( t \) are, by definition, all the admissible sets for a "virtual" \( get() \) operation \( op \) executed instantaneously at time \( t \) (i.e. \( t_{op}(op) = t_{E}(op) = t \)). Given the sub-history \( \hat{H}_t \) at time \( t \) (represented in Figure 3.5), it is possible to define the update sub-history induced by \( op \) by removing the operations \( get(i) \) and \( get(j) \) and the corresponding permutation set is shown in Figure 3.6.

\[
\Pi_{\hat{H}} = \{ \pi_1 = (\text{add}(4)i, \text{add}(3)k, \text{remove}(4)j, op, \text{add}(1)j, \text{remove}(3)k) \\
\pi_2 = (\text{add}(4)i, \text{add}(3)k, \text{remove}(4)j, op, \text{remove}(3)k, \text{add}(1)j) \\
\pi_3 = (\text{add}(4)i, \text{add}(3)k, \text{remove}(4)j, \text{add}(1)j, \text{remove}(3)k) \\
\pi_4 = (\text{add}(4)i, \text{add}(3)k, \text{remove}(4)j, \text{add}(1)j, \text{remove}(3)k, op) \\
\pi_5 = (\text{add}(4)i, \text{add}(3)k, \text{remove}(4)j, \text{remove}(3)k, op, \text{add}(1)j) \\
\pi_6 = (\text{add}(4)i, \text{add}(3)k, \text{remove}(4)j, \text{remove}(3)k, \text{add}(1)j, op) \\
\pi_7 = (\text{add}(3)k, \text{add}(4)i, \text{remove}(4)j, op, \text{add}(1)j, \text{remove}(3)k) \\
\pi_8 = (\text{add}(3)k, \text{add}(4)i, \text{remove}(4)j, \text{remove}(3)k, \text{add}(1)j) \\
\pi_9 = (\text{add}(3)k, \text{add}(4)i, \text{remove}(4)j, \text{remove}(3)k, op, \text{add}(1)j) \\
\pi_{10} = (\text{add}(3)k, \text{add}(4)i, \text{remove}(4)j, \text{remove}(3)k, \text{add}(1)j) \\
\pi_{11} = (\text{add}(3)k, \text{add}(4)i, \text{remove}(4)j, \text{remove}(3)k, op, \text{add}(1)j) \\
\pi_{12} = (\text{add}(3)k, \text{add}(4)i, \text{remove}(4)j, \text{remove}(3)k, \text{add}(1)j, op) \}
\]

Figure 3.6: Permutation Set induced by \( op \) on the Update Sub-history \( \hat{U} \)

Considering all the permutations \( \pi_i \) consistent with \( \hat{U} \) shown above, four possi-\(^3\)An operation \( op \) is instantaneous if and only if \( t_{op}(op) = t_{E}(op) \).
\(^4\)Note that \( op \) is concurrent with \( \text{add}(1)j \) and \( \text{remove}(3)k \) while it follows \( \text{add}(4)i \), \( \text{get}(i) \), \( \text{get}(j) \), \( \text{remove}(4)j \) and \( \text{add}(3)k \).
ble admissible sets for op, and then possible admissible sets at time \( t \), follows:

(i) \( V_{ad}(t) = \emptyset \)

(ii) \( V_{ad}(t) = \{1\} \)

(iii) \( V_{ad}(t) = \{3\} \)

(iv) \( V_{ad}(t) = \{1, 3\} \)

3.3 Synchronous System

This section presents a value-based sequentially consistent protocol implementing a set object in a synchronous dynamic distributed system.

Due to the complexity of the set object, for ease of presentation, it is assumed that the join distribution and the leave distributions are the same and constant (i.e. \( \lambda(t) = \mu(t) = nc \) \( \forall t \) where \( n = n_0 \) is the number of processes belonging to the computation at time \( t_0 \) and \( c \in [0, 1] \) is fraction of processes). Note that, assuming the node function \( N(t) = n_0 \) constant along the time is not a restriction because the analysis can be easily extended to the model used in Chapter 2.

3.3.1 Value-based Sequential Consistent Protocol

Local variables at process \( p_i \). Each process \( p_i \) has the following local variables.

- Two variables denoted \( \text{set}_i \) and \( \text{sn}_i \); \( \text{set}_i \) contains the local copy of the set; \( \text{sn}_i \) is an integer that represents the sequence number for the current update operation.

- A FIFO set variable \( \text{last}_i \) used to maintain an history of recent update operations executed by \( p_i \). Such variable contains 4-uples \(<\text{type}, \text{val}, \text{sn}, \text{id}>\) each one characterizing an update operation of type \( \text{type} = \{A \text{ or } R\} \) (respectively for \text{add()} and \text{remove()}) of the value \( \text{val} \), with a sequence number \( \text{sn} \), issued by a process with identity \( \text{id} \).

- A boolean \( \text{active}_i \), initialized to \( \text{false} \), that is switched to \( \text{true} \) just after \( p_i \) has joined the system.

- Three set variables, denoted \( \text{replies}_i \), \( \text{reply}_i \), and \( \text{pending}_i \), that are used in the period during which \( p_i \) joins the system. \( \text{replies}_i \) contains the 3-uples \(<\text{set}, \text{sn}, \text{ops}>\) that \( p_i \) has received from other processes during its join period, while \( \text{reply}_i \) contains the identifier of processes that are joining the system concurrently with \( p_i \) (as far as \( p_i \) knows). The set \( \text{pending}_i \) contains the 4-uples \(<\text{type}, \text{val}, \text{sn}, \text{id}>\) each one characterizing 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 \( \text{set}_k \) 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 \( \text{set}_k \)), \( \text{sn}_k = 0 \), \( \text{active}_k = \text{false} \), and \( \text{pending}_k = \text{replies}_k = \text{reply}_k = \emptyset \).
The `join()` operation. The algorithm implementing the join operation for a set object, is described in Figure 3.7, 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 $\delta$ 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 $\delta$ time units, i.e., the maximum round trip delay (lines 03-04). When this period terminates, $p_i$ first updates its local variables `set`, `sn`, and `last_ops`, to the most up to date values it has received (lines 05-06) and then executes all the operations concurrent with the join, contained in `pending`, and not yet executed, as if the `UPDATE` message is just received (lines 07-08). Then, $p_i$ becomes active (line 09), which means that it can answer the inquiries it has received from other processes, and does it if `reply_to ≠ ∅` (line 10). Finally, $p_i$ activates the `GARBAGE_COLLECTION()` procedure to avoid that the `last_ops` variable grows to infinite (line 11) and then returns `ok` to indicate the end of the `join()` operation (line 12).
When a process $p_i$ receives a message $\text{inquiry}(j)$, it answers $p_j$ by sending back a $\text{reply}(<\text{set}_i, \text{sn}_i, \text{last}_\text{ops}_i>)$ message containing its local variables if it is active (line 20). Otherwise, $p_i$ postpones its answer until it becomes active (line 21 and line 15). Finally, when $p_i$ receives a message $\text{reply}(<\text{set}, \text{sn}, \text{ops}>)$ from a process $p_j$ it adds the corresponding 3-uple to its set $\text{replies}_i$ (line 24).

**Why the $\text{wait}(\delta)$ statement at line 02 of the $\text{join}()$ operation?** To motivate the $\text{wait}(\delta)$ statement at line 02, let us consider the execution of the $\text{join}()$ operation depicted in Figure 3.8(a). At time $t$, the processes $p_i$, $p_h$ and $p_k$ are the three processes composing the system. Moreover, the process $p_j$ executes $\text{join}()$ just after $t$. The set is initially empty. Due to the “timely delivery” property of the broadcast invoked by $p_i$ to send the $\text{update}$ 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($\delta$) statement at line 02, its execution of the lines 05-12 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 3.8(b) shows that this incorrect scenario cannot occur if $p_j$ is forced to wait for $\delta$ 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 3.9. The get is purely local (i.e., fast): it consists in returning the current value of the local variable $set_i$.

```
operation get(): % issued by any process $p$, %
(01) return($set_i$).
```

```
operation add(v): % issued by any process $p$,%
(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 $\leftarrow last_ops_i \cup \{< A, v, sn_i, i >\}$;
(06) wait($\delta$);
(07) return(ok).
```

```
operation remove(v): % issued by any process $p$,%
(08) $sn_i \leftarrow sn_i + 1$;
(09) broadcast update($R$, $v$, $sn_i$, $i$);
(10) $set_i \leftarrow set_i \setminus \{v\}$;
(11) last_ops $\leftarrow last_ops_i \cup \{< R, v, sn_i, i >\}$;
(12) wait($\delta$);
(13) return(ok).
```

(14) when update(type, val, sn_j, j) is delivered: % at any process $p$, %
(15) if ¬active then pending $\leftarrow pending \cup \{< type, val, sn_j, j >\}$
(16) else execute update(type, val, sn_j, j)
(17) endif.
```

Figure 3.9: The get(), add() and remove() protocol for a synchronous system (code for $p_i$)

The add(v) and the remove(v) operations. The add() and the remove() algorithms are shown in Figure 3.9. Both the add() operation and the remove() operation 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
3.3. SYNCHRONOUS SYSTEM

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 \(pi\) 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 3.9), it broadcasts an \texttt{update(type, val, sn, id)} message (line 03 and line 09 of Figure 3.9) where \texttt{type}\ is a flag that
identify the type of the update (i.e. \(A\) for an add() operation or \(R\) for a remove() operation), \texttt{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
3.9) and it stores in its \texttt{last_ops}_i variable the tuple \(< type, val, sn, id >\) that identifies
such operation (line 05 and line 11 of Figure 3.9). Then \(pi\) waits for \(\delta\) time units (line
06 and line 12 of Figure 3.9) to be sure that all the active processes have received the
\texttt{update} message and finally it returns by the operation (line 07 and line 13 of Figure
3.9).

When \(pi\) receives the \texttt{update(type, val, sn, j)} from a process \(pj\), if it is not active,
it puts the current \texttt{update} message in its \texttt{pending}_i buffer and will process it as soon
as it will be active, otherwise it executes the \texttt{update()} procedure shown in Figure 3.10.

The \texttt{update()} procedure. In the \texttt{update()} procedure (Figure 3.10) \(pi\), checks if the
sequence number \(sn_j\), corresponding to the current operation, is greater than the one
stored by \(pi\) and if it is so, \(pi\) execute the operation (lines 01-04). Contrary, \(pi\) checks
if, in the set of the last executed operation \texttt{last_ops}_i, there is some operation occurred
on the same value \texttt{val}; if there is not such an operation, \(pi\) executes the current one
(lines 05 - 12) 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, \(pi\) executes
the operation (lines 13 - 21). Finally \(pi\) updates it sequence number (line 22).

Garbage Collection  Let us remark that the \texttt{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 operations is high, each process becomes immediately overloaded of information. To avoid this problem it is possible to define a \texttt{garbageCollection()} procedure that periodically removes from the \texttt{last_ops}_i variable the information related to “old” operation.
The thread managing the garbage collection is very easy and it is shown in Figure
3.11. It is activated the first time at the end of the \texttt{join()} operation (line 17, Figure
procedure update(type, val, sn, j) % at any process \( p \) %
(01) if \( (sn > sn) \) then \( last_{ops} \leftarrow last_{ops} \cup \{ \langle type, val, sn, j \rangle \} \);
(02) if \( (type = A) \) then \( set \leftarrow set \cup \{ \langle val, sn \rangle \} \);
(03) else \( set \leftarrow set \setminus \{ \langle val, - \rangle \} \);
(04) endif
(05) else
(06) temp \leftarrow \{ X \in last_{ops} | X = \langle - , val, - , - \rangle \} %
(07) if \( (temp = \emptyset) \)
(08) then \( last_{ops} \leftarrow last_{ops} \cup \{ \langle type, val, sn, j \rangle \} \);
(09) if \( (type = A) \) then \( set \leftarrow set \cup \{ \langle val, sn \rangle \} \);
(10) else \( set \leftarrow set \setminus \{ \langle val, - \rangle \} \);
(11) endif
(12) else if \((type = A) \wedge
(\forall R, - , sn, id \in temp \mid (sn, id) > (sn, j)) \)
(13) then \( set \leftarrow set \cup \{ \langle val, sn \rangle \} \);
(14) endif
(16) if \((type = R) \wedge
(\forall A, - , sn, id \in temp \mid (sn, id) > (sn, j)) \)
(17) then \( set \leftarrow set \setminus \{ \langle val, - \rangle \} \);
(18) endif
(19) endif
(20) endif
(21) endif
(22) \( sn \leftarrow \max(sn, sn) \).

Figure 3.10: The update() protocol for a synchronous system (code for \( p_i \))

3.7) and then is always running. Every \( 2\delta \) time unit, it checks the operations not known in the previous (line 05) period and discard the ones already executed (line 06).

3.3.2 Correctness proof

In this section the correctness of the proposed protocol is proved. In particular, given the algorithms shown in Figures 3.7 - 3.10, the termination of each operation is first shown (Theorem 5) and then the value-based sequential consistency of every execution history generated is proved.

To demonstrate that the protocol generates only value-based sequential consistent histories if the churn is below a certain bound (Theorem 6) two main steps are defined: (i) if every active process maintains at any time an admissible set, the execution history is always value based sequential consistent (Lemma 21); (ii) if the churn is below a certain bound, every get() operation returns an admissible set (Lemma 20).

To get this result it is also proved that if the churn is below a certain bound, a process that issued a join operation becomes active endowing an admissible set (Lemma 19).
3.3. SYNCHRONOUS SYSTEM

Figure 3.11: The garbageCollection() procedure for a set object in a synchronous system (code for \( p_i \))

\[
\text{procedure garbageCollection():}\n\begin{align*}
(01) & \quad \text{temp} \leftarrow \emptyset; \quad \text{old Ops} \leftarrow \emptyset; \\
(02) & \quad \text{while(true) every } 2\delta \\
(03) & \quad \text{temp} \leftarrow \emptyset; \\
(04) & \quad \text{for each op =< type, val, sn, id >\in last Ops, do} \\
(05) & \quad \quad \text{if (op \notin (last Ops \cap old Ops)) then temp \leftarrow op;} \\
(06) & \quad \quad \text{else last Ops } \leftarrow \text{last Ops}/op; \\
(07) & \quad \quad \text{endIf} \\
(08) & \quad \text{endFor;} \\
(09) & \quad \text{old Ops } \leftarrow \text{temp;} \\
(10) & \quad \text{endWhile}
\end{align*}
\]

Theorem 5 If a process invokes join() and does not leave the system for at least \( 3\delta \) 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 \( \delta \) 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 and line 04 of Figure 3.7 and at line 06 and line 12 of Figure 3.9 terminates. \( \square \) Theorem 5

Lemma 17 Let \( c < 1/(3\delta) \). \( \forall t : |A[t, t + 3\delta]| \geq n(1 - 3\delta c) > 0. \)

Proof Let first consider the case \( t_0 = 0 \) where all the processes are active (i.e. \( |A(t_0)| = n \)). Due to definition of \( c \), at time \( t_1 \), \( nc \) processes leave the system and \( nc \) processes invoke the join() operation; hence, \( |A[t_0, t_0 + 1]| = n - nc \). During the second time unit, \( nc \) new processes enter the system and replace the \( nc \) processes that left the system during that time unit. In the worst case, the \( nc \) processes that left the system are processes that were present at time \( t_0 \) (i.e., they are not processes that entered the system between \( t_0 \) and \( t_0 + 1 \)). So, \( |A[t_0, t_0 + 2]| \geq n - 2nc \). If a period of \( 3\delta \) time units is considered, i.e. the longest period needed to terminate a join() operation, it follows that \( |A[t_0, t_0 + 3\delta]| \geq n - 3\delta nc = n(1 - 3\delta c) \). Moreover, as \( c < 1/3\delta \), then \( |A[t, t + 3\delta]| \geq n(1 - 3\delta c) > 0 \). It is easy to see that the previous reasoning depends only on (1) the fact that there are \( n \) processes at each time \( t \), and (2) the definition of the churn rate \( c \), from which it is possible to conclude that \( \forall t : |A[t, t + 3\delta]| \geq n(1 - 3\delta c). \) \( \square \) Lemma 17

Lemma 18 Let \( t_0 \) be the time at which the computation of a set object \( S \) starts, \( \hat{H} = (H, \prec) \) an execution history of \( S \), and \( \hat{H}_{t_1 + 3\delta} = (H_{t_1 + 3\delta}, \prec) \) the sub-history of \( \hat{H} \)
at time $t_1 + 3\delta$. Let $p_i$ be a process that invokes $\text{join}()$ on $S$ at time $t_1 = t_0 + 1$, if $c < 1/(3\delta)$ then at time $t_1 + 3\delta$ the local copy $s_i$ of $S$ maintained by $p_i$ will be an admissible set at time $t_1 + 3\delta$.

**Proof** Let suppose by contradiction that the local copy $s_i$ of the set object $S$, maintained by $p_i$, is not an admissible set at time $t_1 + 3\delta$.

If $s_i$ is not an admissible set at time $t_1 + 3\delta$ then it is not an admissible set for any operation $op = \text{get()}$ executed instantaneously at time $t_1 + 3\delta$. Considering the updated sub-history $\hat{U}$ induced by $op$ on $\hat{H}$ and its the permutation set $\Pi_{\hat{H}}$, it means that there not exists any permutation $\pi_i$, belonging to the permutation set $\Pi_{\hat{H}}$ induced by $op$, such that $\pi_i$ generates $s_i$.

It follows that one of the following conditions holds:

1. there exists a value $v$, generated by every permutation $\pi_i$ of the permutation set $\Pi_{\hat{H}}$, such that $v$ does not belong to $s_i$ (i.e. $\exists v : \forall v_i \in \Pi_{\hat{H}} (\exists \text{add}(v) \in \pi_i : \text{add}(v) \rightarrow s_i, op) \wedge (\not\exists \text{remove}(v) \in \pi_i : \text{add}(v) \rightarrow s_i, \text{remove}(v) \rightarrow s_i, op) \wedge (v \not\in s_i)$);

2. there exists a value $v$ belonging to $s_i$ such that it can not be generated by any permutation $\pi_i$ of the permutation set $\Pi_{\hat{H}}$ (i.e. $\exists v \in s_i : \forall v_i \in \Pi_{\hat{H}} (\exists \text{add}(v), \text{remove}(v) \in \pi_i : \text{add}(v) \rightarrow s_i, \text{remove}(v) \rightarrow s_i, op))$.

**Case 1.** Let $p_j$ be the process issuing the $\text{add}(v)$ operation. Note that, if the $\text{add}(v)$ precedes $op$ in every $\pi_i$, it follows that $\text{add}(v)$ precedes $op$ in the execution history $\hat{H}$.

Since $op$ is instantaneously executed at time $t_1 + 3\delta$ (i.e. $t_B(op) = t_E(op) = t_1 + 3\delta$) then $t_E(\text{add}(v)) < t_1 + 3\delta$. With respect to the $\text{join}()$ invocation time, the $\text{add}(v)$ operation could be started after or before; let consider separately the two cases:

- If $t_B(\text{add}(v)) \geq t_1$ then $p_i$ is already inside the system when the $\text{add}(v)$ operation starts and then, due to the broadcast property, $p_i$ will receive the update message sent by $p_j$ and will include such operation into the buffer $\text{pending}_i$.

At time $t_1 + 3\delta$, $p_i$ will execute lines $07 - 12$ of Figure 3.7 and then will execute the $\text{update}$ procedure for the buffered $\text{add}(v)$ operation inserting $v$ in $s_i$.

Since there not exist any $\text{remove}(v)$ operation in $\pi_i$ between the $\text{add}(v)$ and $op$, $v$ belongs to the set and there is a contradiction.

- If $t_B(\text{add}(v)) < t_1$, $p_i$ has no guarantees to receive the update message sent from $p_j$ but, due to the broadcast property, every active process at time $t_B(\text{add}(v))$ will receive it and will execute the operation. Since $t_B(\text{add}(v)) \geq t_0$ and any update operation is bounded by $\delta$, the wait statement in line 02 assures that such update operation is done by every active process before $p_i$ sends the $\text{INQUIRY}$ message. Therefore, any active process that replies to $p_i$ has in its local copy...
of the set the value \( v \) and, due to Lemma 17, there exists at least one process inside the computation from time \( t_1 \) to time \( t_1 + 3\delta \) that replies to \( p_i \). Upon the reception of such set, \( p_i \) will execute line 24 of Figure 3.7 by including the \( s_{ij} \) received (containing \( v \)) in its \( \text{reply}_i \) buffer. At time \( t_1 + 3\delta \), \( p_i \) will take from its buffer the entry with the highest sequence number and will copy such set into its local copy. Considering that (i) each received \( s_{ik} \) will contain \( v \) and (ii) there not exists any \( \text{remove}(v) \) operation in \( \pi_i \), between the \( \text{add}(v) \) and \( \text{op} \), \( v \) belongs to the set and there is a contradiction.

**Case 2.** Considering that \( s_{ij} \) is empty at the beginning of the \( \text{join}() \) operation (line 01, Figure 3.7), if \( v \in s_{ij} \) then (i) \( v \) belongs to some \( s_{ij} \) received by some active process \( p_j \) (lines 23 - 24, Figure 3.7) or (ii) \( v \) has been added by processing the buffer \( \text{pending} \).

At the beginning of the computation, each process \( p_j \) has an empty \( s_{ij} \) variable. If some process \( p_j \) sends to \( p_i \) a \( s_{ij} \) variable containing \( v \), it follows that \( p_j \) has executed 02 or 09 or 13 of the \( \text{update} \) procedure (Figure 3.10). Such procedure is activated when an \( \text{update} \) message is delivered to an active process (line 16, Figure 3.9); let \( t \) be the time when \( p_j \) delivers the \( \text{update}(\langle A, v, -, - \rangle) \) message and triggers the \( \text{update} \) procedure. Note that, \( \text{update} \) messages are sent by active processes when an \( \text{add}()\)/\( \text{remove}() \) operation is triggered (line 03 and line 09, Figure 3.9).

It follows that there exists an \( \text{add}(v) \) operation issued before time \( t \) such that its invocation time precedes \( \text{op} \) (i.e. \( t_\theta(\text{add}(v)) < t < t_1 + 3\delta \)). Therefore, the \( \text{add}(v) \) precedes or is concurrent with \( \text{op} \).

If the \( \text{add}(v) \) operation precedes or is concurrent with \( \text{op} \) then there exists at least one permutation \( \pi_i \), consistent with \( \hat{U} \), where \( \text{add}(v) \rightarrow_{\pi_i} \text{op} \).

By hypothesis, if there exists an \( \text{add}(v) \) then there exists also a \( \text{remove}(v) \) and, for each permutation \( \pi_i \) in the permutation set, \( \text{add}(v) \rightarrow_{\pi_i} \text{remove}(v) \rightarrow_{\pi_i} \text{op} \). It follows that \( \text{add}(v) \prec \text{remove}(v) \) in \( \hat{H} \).

If \( \text{add}(v) \prec \text{remove}(v) \) in \( \hat{H} \) then when a process \( p_k \) issued the \( \text{remove}(v) \) operation, it has already executed the \( \text{add}(v) \) and in particular it has executed line 22 of the \( \text{update} \) procedure. Hence, the sequence number attached to the \( \text{remove}(v) \) operation will be greater than the one attached to the \( \text{add}(v) \). It follows that every active process will execute first the \( \text{add}(v) \) and then the \( \text{remove}(m) \), and it will execute line 03 of the \( \text{update} \) procedure by removing \( v \) from their local copies of the set and incrementing its update sequence number.

Due to Lemma 17, there exists at least one process \( p_j \) inside the computation from time \( t_1 \) to time \( t_1 + 3\delta \) that replies to \( p_i \) and such that \( p_j \) has executed both the operations. Note that \( p_j \)’s set has the highest sequence number and it does not contain \( v \). Hence, when \( p_i \) will select a copy of the set, it cannot find \( v \) and we have a contradiction.

The same reasoning applies to \( p_i \) while processing the \( \text{pending} \) buffer and the claim
Lemma 19 Let $\tilde{H} = (H, \prec)$ be the execution history of a set object $S$, and $p_i$ a process that invokes $\text{join()}$ on the set $S$ at time $t$. If $c < 1/(3\delta)$ then at time $t + 3\delta$ the local copy $set_i$ of $S$ maintained by $p_i$ will be an admissible set at time $t + 3\delta$.

Proof Let suppose by contradiction that the local copy $set_i$ of the set object $S$, maintained by $p_i$, is not an admissible set at time $t + 3\delta$. If $set_i$ is not an admissible set at time $t + 3\delta$ then it is not an admissible set for any operation $op = \text{get()}$ executed instantaneously at time $t + 3\delta$. Considering the updated sub-history $\tilde{U}$ induced by $op$ on $\tilde{H}$ and its the permutation set $\tilde{\Pi}_\tilde{U}$, its means that there not exists any permutation $\pi_i$, belonging to the permutation set $\tilde{\Pi}_\tilde{U}$, such that $\pi_i$ generates $set_i$.

It follows that one of the following conditions holds:

1. there exists a value $v$, generated by every permutation $\pi_i$ of the permutation set $\tilde{\Pi}_\tilde{U}$, such that $v$ does not belong to $set_i$ (i.e. $\exists v : \forall \pi_i \in \tilde{\Pi}_\tilde{U} \ (\exists \text{add}(v) \in \pi_i : \text{add}(v) \rightarrow_{\pi_i} op) \land (\exists \text{remove}(v) \in \pi_i : \text{add}(v) \rightarrow_{\pi_i} \text{remove}(v) \rightarrow_{\pi_i} op) \land (v \notin set_i)$);

2. there exists a value $v$ belonging to $set_i$ such that it can not be generated by any permutation $\pi_i$ of the permutation set $\tilde{\Pi}_\tilde{U}$ (i.e. $\exists v \in set_i : \forall \pi_i \in \tilde{\Pi}_\tilde{U} \ (\exists \text{add}(v), \text{remove}(v) \in \pi_i : \text{add}(v) \rightarrow_{\pi_i} \text{remove}(v) \rightarrow_{\pi_i} op)$).

Case 1. Let $p_j$ be the process issuing the $\text{add}(v)$ operation. Note that, if the $\text{add}(v)$ precedes $op$ in every $\pi_i$, it follows that $\text{add}(v)$ precedes $op$ in the execution history $\tilde{H}$.

Since $op$ is instantaneously executed at time $t + 3\delta$ (i.e. $t_B(op) = t_E(op) = t + 3\delta$) then $t_E(\text{add}(v)) < t + 3\delta$. With respect to the $\text{join()}$ invocation time, the $\text{add}(v)$ operation could be started after or before; let consider separately the two cases:

- If $t_B(\text{add}(v)) \geq t$ then $p_i$ was already inside the system when the $\text{update}$ operation starts and then, due to the broadcast property, $p_j$ will receive the $\text{update}$ message sent by some process $p_j$ and will include such operation to the buffer $\text{pending}_i$. At time $t + 3\delta$, $p_i$ will execute lines 07–12 of Figure 3.7 and then will execute the $\text{add}(v)$ operation inserting $v$ in $set_i$. Since there not exist any $\text{remove}(v)$ operation, in $\pi_i$ between the $\text{add}(v)$ and $op$, $v$ belongs to the set and there is a contradiction.

- If $t_B(\text{add}(v)) < t$, $p_i$ may not receive the $\text{update}$ message from the process issuing the update. If $v$ is not contained in any received $set_j$ then each active
process \( p_j \) replying to \( p_i \) has not received the update for the \( \text{add}(v) \) or has received but an \( \text{add}(v) \) and a \( \text{remove}(v) \). Let consider a generic process \( p_j \) replying to \( p_i \); it could be active or not at time \( t_B(\text{add}(v)) \).

\[ \text{Case 2.} \] Considering that \( \text{set}_i \) is empty at the beginning of the \( \text{join}() \) operation (line 01, Figure 3.7), if \( v \in \text{set}_i \) then (i) \( v \) belongs to some \( \text{set}_j \) received by some active process \( p_j \) (lines 23 - 24, Figure 3.7) or (ii) \( v \) has been added by processing the buffer \( \text{pending}_i \).

At the beginning of the computation, each process \( p_j \) has an empty \( \text{set}_j \) variable. If some process \( p_j \) sends to \( p_i \) a \( \text{set}_j \) variable containing \( v \), it follows that \( p_j \) has executed 02 or 09 or 13 of the \( \text{vpa} \) procedure (Figure 3.10). Such procedure is activated when an \( \text{update} \) message is delivered to an active process (line 16, Figure 3.9); let \( t' \) be the time when \( p_j \) delivers the \( \text{update}(<\ A,\ v,\ -,->) \) message and triggers the \( \text{update} \) procedure. Note that, \( \text{update} \) messages are sent by active processes when an \( \text{add}() / \text{remove}() \) operation is triggered (line 03 and line 09, Figure 3.9).

It follows that there exists an \( \text{add}(v) \) operation issued before time \( t' \) such that its invocation time precedes \( \text{op} \) (i.e. \( t_B(\text{add}(v)) < t' < t + 3\delta \)). Therefore the \( \text{add}(v) \) precedes or is concurrent with \( \text{op} \).

If the \( \text{add}(v) \) operation precedes or is concurrent with \( \text{op} \) then there exists at least one
permutation $\pi_i$, consistent with $\hat{U}$ induced by $op$, where $\text{add}(v) \rightarrow_{\pi_i} op$.
By hypothesis, if there exists an $\text{add}(v)$ then there exists also a $\text{remove}(v)$ and, for each permutation $\pi_i$ in the permutation set, $\text{add}(v) \rightarrow_{\pi_i} \text{remove}(v) \rightarrow_{\pi_i} op$. It follows that $\text{add}(v) \prec \text{remove}(v)$ in $\hat{H}$.
If $\text{add}(v) \prec \text{remove}(v)$ in $\hat{H}$ then when a process $p_k$ issued the $\text{remove}(v)$ operation, it has executed the $\text{add}(v)$ and in particular it has executed line 22 of the $\text{update}$ procedure. Hence, the sequence number attached to the $\text{remove}(v)$ operation will be greater than the one attached to the $\text{add}(v)$. It follows that every active process will execute first the $\text{add}(v)$ and then the $\text{remove}(v)$, and it will execute line 03 of the $\text{update}$ procedure by removing $v$ from their local copies of the set.
Due to Lemma 17, there exists at least one process $p_j$ inside the computation from time $t$ to time $t + 3\delta$ that replies to $p_i$ and maintains a set not containing $v$ and having the highest sequence number. Moreover, due to Lemma 18, $p_j$ maintains an admissible set (i.e. $p_j$ has executed both the operations and has a set with the highest sequence number that does not contain $v$). Hence, when $p_j$ will select a copy of the set, it cannot find $v$ and we have a contradiction.

The same reasoning applies to $p_i$ while processing the pending buffer and the claim follows. \hfill $\square$

\textbf{Lemma 20} Let $S$ be a set object and let $op$ be a $\text{get()}$ operation issued on $S$ 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. $V = V_{\text{ad}}(op)$).

\textbf{Proof} Let us suppose by contradiction that there exists a process $p_j$ that issues a $\text{get()}$ operation $op$ on the set object $S$ that returns a set of value $V$ not admissible for $op$. Considering the updated sub-history $\hat{U}$ induced by $op$ on $\hat{H}$ and its the permutation set $\Pi_\hat{U}$, if $V$ is not an admissible set then there not exists any permutation $\pi_i$, belonging to the permutation set $\Pi_\hat{U}$ induced by $op$, such that $\pi_i$ generates $set_i$.
It follows that one of the following conditions holds:

1. there exists a value $v$, generated by every permutation $\pi_i$ of the permutation set $\Pi_\hat{U}$, such that $v$ does not belong to $set_i$ (i.e. $\exists v : \forall \pi_i \in \Pi_\hat{U} \ (\exists \text{add}(v) \in \pi_i : \text{add}(v) \rightarrow_{\pi_i} op) \land (\exists \text{remove}(v) \in \pi_i : \text{add}(v) \rightarrow_{\pi_i} \text{remove}(v) \rightarrow_{\pi_i} op)$
   $\land (v \notin set_i)$);

2. there exists a value $v$ belonging to $set_i$ such that it can not be generated by any permutation $\pi_i$ of the permutation set $\Pi_\hat{U}$ (i.e. $\exists v \in set_i : \forall \pi_i \in \Pi_\hat{U} \ (\exists \text{add}(v), \text{remove}(v) \in \pi_i : \text{add}(v) \rightarrow_{\pi_i} \text{remove}(v) \rightarrow_{\pi_i} op$).

\textbf{Case 1}. Let $p_j$ be the process issuing the $\text{add}(v)$ operation. Note that, if the $\text{add}(v)$ precedes $op$ in every $\pi_i$, it follows that $\text{add}(v)$ precedes $op$ in the execution history $\hat{H}$. Since there exists the $\text{add}(v)$ operation then there exists also a process $p_j$ issuing such operation that executes the algorithm of Figure 3.9.
3.3. SYNCHRONOUS SYSTEM

- If \( p_i = p_j \), then \( p_i \) has executed lines 04-05 of Figure 3.9 and has added \( v \) to its local copy \( set_i \) of the set object. Since there not exists any \( \text{remove}(v) \) operation between the \( \text{add}(v) \) and \( op \) and since the \( \text{get()} \) operation returns the content of the local copy of the set without modifying it (line 01 Figure 3.9) 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 3.9 by sending an \( \text{update} \) \((A, v, sn_j, j)\) message by using the broadcast primitive. Due to the broadcast property, every process that is active at time \( t \) will receives the update up to time \( t + \delta \).

\( \diamond \) If \( p_i \in A(t) \), 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 3.9); otherwise it checks if it has in its \( \text{last}_i \) operation already executed that “collides” with the current one (i.e. if there is a \( \text{remove}(v) \) of the same element issued by a process with a lower identifier). Since there not exist any \( \text{remove}(v) \) operation between the \( \text{add}(v) \) and \( op \) and since the \( \text{get}() \) operation returns the content of the local copy of the set without modifying it (line 01 Figure 3.9) then \( v \in V \) and we have a contradiction.

\( \diamond \) \( p_i \notin A(t) \), 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 \( p_j \) into the \( \text{pending}_i \) variable, and will execute the update just before become active (lines 07-12 Figure 3.7); \( p_i \) will add \( v \) to its local copy of the set. Since there not exists any \( \text{remove}(v) \) operation between the \( \text{add}(v) \) and \( op \) and since the \( \text{get}() \) operation returns the content of the local copy of the set without modifying it (line 01 Figure 3.9) then \( v \in V \) and we have a contradiction. Contrarily, 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 19, 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 \( \text{remove}(v) \) operation between the \( \text{add}(v) \) and \( op \) and since the \( \text{get}() \) operation returns the content of the local copy of the set without modifying it (line 01 Figure 3.9) 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 \( \text{set}_i \) 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 19). If \( v \) belongs to an admissible set at time \( t \), it means that \( v \) is generated by some permutation \( \pi_i \) belonging to the permutation set \( \Pi_{\hat{U}} \) of the update sub-history at time \( t \). It follows that there exists an \( \text{add}(\cdot) \) operation that terminates or is running at time \( t \). Note that, each permutation \( \pi_i \) belonging to the permutation set \( \Pi_{\hat{U}} \) is a prefix for at least one permutation belonging to the permutation set \( \Pi_{\hat{H}} \). Since there not exist any \( \text{remove}(\cdot) \) operation between the \( \text{add}(\cdot) \) and \( \text{op} \), \( v \) is never removed and we have a contradiction.

- If \( p_i \) has added \( v \) to \( \text{set}_i \) as consequence of an \( \text{update} \) message it means that there exists a process \( p_j \) that has sent it. An \( \text{update} \) message is generated by a process \( p_j \) when the \( \text{add}(\cdot) \) operation is issued; this means that there exist an \( \text{add}(\cdot) \) operation that precedes or is concurrent with \( \text{op} \). Hence, there exists at least one permutation \( \pi_i \) containing \( \text{add}(\cdot) \). Since there not exist any \( \text{remove}(\cdot) \) operation between the \( \text{add}(\cdot) \) and \( \text{op} \) in \( \pi_i \), \( v \) is never removed and there is a contradiction.

\[ \square \text{Lemma 20} \]

**Lemma 21** Let \( S \) be a set object and let \( \hat{H} = (H, \prec) \) be an execution history of \( S \) generated by the algorithm in Figure 3.7 and Figure 3.9. 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 3.7 and Figure 3.9 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 \( \text{op} = \text{add}(\cdot) \) and \( \text{op}' = \text{remove}(\cdot) \) (with \( v = v' \)) appear in a different order. Without loss of generality, let us suppose that \( \text{op} \) and \( \text{op}' \) have no other concurrent operation, \( \text{op} \) is issued by a process \( p_h \), \( \text{op}' \) is issued by a process \( p_k \) and \( t_B(\text{op}) < t_B(\text{op}') \).

At time \( t < t_B(\text{op}) \) all the processes has the same value for their sequence numbers\(^5\) and let us suppose that \( s_{n_w} = x \) for every \( p_w \). When process \( p_h \) issues the \( \text{add}(\cdot) \) operation, it increments its sequence number \( s_{n_w} = x + 1 \) by executing line 02 of Figure 3.9 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 \( \text{update} \) message of \( p_h \) before issuing the \( \text{remove}(\cdot) \) operation (Figure 3.12(a)) or (ii) process \( p_k \) receives the \( \text{update} \) message of \( p_h \) just after issuing the \( \text{remove}(\cdot) \) operation (Figure 3.12(b)).

\(^5\)Let us recall that the sequence number maintained by every process counts the number of updates issued on the local copy of the set.
3.3. SYNCHRONOUS SYSTEM

Let us show what happens 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 $\text{remove}(v)$ operation, it increments its sequence number $sn_k = x + 2$ by executing line 02 of Figure 3.9 and attaches such value to the $\text{update}(R, v, x + 2, k)$ message. Let us consider the scenario depicted in Figure 3.12(a) where $p_i$ receives first the update message of the $\text{remove}(v)$ sent by $p_k$ and then the update message of the $\text{add}(v)$ sent by $p_h$ while $p_j$ receives first the update message of the $\text{add}(v)$ sent by $p_h$ and then the update message of the $\text{remove}(v)$ 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 adding the value $v$ and storing the tuple $< A, v, x + 2, h >$ in the $\text{last}\_\text{ops}_i$ 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 $\text{last}\_\text{ops}_i$ 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 $\text{add}(v)$ is equal to execute $\text{add}(v)$ followed by a $\text{remove}(v)$ 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_i} 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\((R, v, x + 2, k)\) 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 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 message 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 3.12(b) where \( p_i \) receives first the update message of the remove\((v)\) sent by \( p_h \) and then the update message of the add\((v)\) sent by \( p_k \) 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 finds 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 finds 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 $op \rightarrow_{\hat{S}_j} op'$.

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

\[ \square \text{Lemma 21} \]

From Lemma 20 and Lemma 21 follow the theorem that concludes our proof.

**Theorem 6** Let $S$ be a set object and let $\hat{H} = (H, \prec)$ be an execution history of $S$ generated by the algorithm in Figure 3.7 and Figure 3.9. If $c < 1/(3\delta)$, $\hat{H}$ is always value-based sequential consistent.

### 3.4 Eventually Synchronous System: Impossibility Result

In the following we will show that it is not possible to define an algorithm $A_{set}$ implementing the set object, by exchanging messages, that uses finite memory if (i) the accesses to the set are continuous, (ii) the churn is continuous and (iii) the system is eventually synchronous.

To this aim, an algorithm $A_{set}$ is modeled as an I/O automaton where transitions are associated to algorithm actions (i.e. input, output and internal action). An automaton is composed by: the signature, the set of the states, an initial state, state-transition relations and a task partition [42].

$A_{set}$ is executed by each process and $V$ denotes the arbitrary bounded domain of the set.

$A_{set}$ has the following signature:

**Input:**
- receive($v$)$_i$, $v \in V$
- get()$_i$
- add($v$)$_i$, $v \in V$
- remove($v$)$_i$, $v \in V$
- join()$_i$

**Output:**
- send($v$)$_i$, $v \in V$
- get_return($V$)$_i$, $V \in 2^{|V|}$

Figure 3.13 represents the automaton, implementing a set object, executed by each process $p_i$.

\[ \text{Lemma 22} \] Let $A_{set}$ be an automata implementing the set object. If the accesses to the set are continuous and the states of the automata are represented by the list of operations issued on the object then $A_{set}$ uses infinite memory.
Proof The lemma trivially follows by considering that accesses to the set are continuous and then the list of operations tends to infinite thus requiring infinite memory to be stored.

Lemma 23 Let $\mathcal{A}_{set}$ be an automata implementing the set object. If the churn is continuous, the system is eventually synchronous and the states of the automata are represented by subset of values of the domain of the set then $\mathcal{A}_{set}$ is not able to output an admissible set.

Proof Let consider a get() input occurring at time $t$ and let us assume that there exists in the computation at least one process $p_i$ whose state $V \in 2^{|V|}$ is an admissible set. In order that the output action get_return() might return $V$, the following necessary condition must hold: "$p_i$ has to belong to the computation for all the time required by the state-transition generated by the input get()." The combined effect of the continuous churn and the eventual synchrony of the system can prevent this condition to hold for every input get(). The eventual synchrony indeed implies that no finite bound can be established on the time taken by a transition to be completed (finite bounds can be established only when the synchrony period starts). This implies that for any value of churn $c$ greater than 0 (see Section 4) the member of the computation at the time the transition starts could be completely replaced at the time the transition ends. As a consequence, $p_i$ could leave the computation before the transition be completed and the necessary condition is not satisfied.

Theorem 7 Let $\mathcal{A}_{set}$ be an automata implementing the set object. If (i) the accesses to the set are continuous, (ii) the churn is continuous and (iii) the system is eventually synchronous then there not exists any algorithm $\mathcal{A}_{set}$ able to implement the set object using finite memory.

Proof It follows by Lemma 22 and Lemma 23.
3.5 \textit{k}-bounded Set Object

Due to the impossibility shown in Theorem 7, the specification of the set object is weakened and the notion of \textit{k}-bounded set shared object is introduced. Informally, a \textit{k}-bounded set object is a set that has limited memory of the execution history. In particular, given a \texttt{get()} operation, a \textit{k}-bounded set behaves as a set where the execution history is limited only to the \textit{k} most recent update operations and where all the other operations are forgotten. Hence, each \texttt{get()} operation defines a “window” on the execution history and operations out of the window are hidden to the \textit{k}-bounded set and appear as never invoked.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{windows.png}
\caption{Windows for a 3-bounded Set shared object.}
\end{figure}

As an example consider the execution shown in Figure 3.14 for a \textit{k}-bounded set object where \(k = 3\). For each of the two \texttt{get()} operations, the execution history is restricted to the last 3 update operations. Let consider the \texttt{get()} invoked by \(p_k\) then the set returned by the operation is \(\{1, 2, 3\}\) (as the one that would be returned by the same \texttt{get()} invoked on a set). Considering the second \texttt{get()} invoked by \(p_j\), the set returned will be \(\emptyset\), instead of \(\{3\}\) that would be returned by the same \texttt{get()} invoked on a set (this is actually due to the \texttt{add(1)} and \texttt{add(3)} operations that are out of the window for such \texttt{get()} and then they are forgotten).

Note that, weakening the set to a \textit{k}-bounded set and allowing to remember only the recent history, it is possible to make an implementation even if processes has limited memory space. In fact, the parameter \(k\), that defines the width of the window, is directly related to the available memory: higher is the available space, higher is the value of \(k\) that can be used. Moreover, the value of the parameter \(k\) gives also a measure of the approximation that the \textit{k}-bounded set is with respect to the set: higher is the value of \(k\), less are the operations that are forgot.

Due to the concurrency, the meaning of “most recent update operations” may not be well defined. As an example, consider the execution shown in Figure 3.15. Given the \texttt{get()} operation issued by \(p_j\), the \texttt{add(4)} issued by \(p_i\) and the \texttt{remove(1)} issued by \(p_k\) have to be considered as recent operations? Depending on which operations are considered or not, different sets of recent operations are considered (i.e. blue rectangle, red rectangle and green rectangle).

In order to specify the correct behavior of the \textit{k}-bounded set object in case of
concurrency, let us introduce the notions of *k-cut permutation induced by an operation* \( op_i \) and *k-cut permutation set*. Informally, a *k-cut permutation induced by an operation* \( op_i \) is a subset of a permutation (induced by \( op_i \)) consistent with the execution history. This subset contains \( k \) operations preceding \( op_i \) in the permutation. The *k-cut permutation set*, is the set of permutations induced by \( op_i \) obtained by the permutation set \( \Pi_{\hat{H}} \) of the execution by considering for each permutation its *k-cut*.

**Definition 16 (k-cut permutation induced by \( op_i \))** Given an execution history \( \hat{H} = (H,\prec) \) let \( \pi = (op_1, op_2, \ldots, op_n) \) a permutation consistent with \( \hat{H} \). Given an operation \( op_i \) of \( \pi \) and an integer \( k \), the *k-cut permutation* induced by \( op_i \) on \( \pi \), denoted as \( \pi_k(op_i) \), is the sub-set of \( \pi \) ending with \( op_i \) and including the \( k \) operations that precede \( op_i \) in \( \pi \) (i.e. \( \pi_k(op_i) = (op_{i-k}, \ldots, op_{i-1}, op_i) \)).

As an example, consider the execution history \( \hat{H} \) shown in Figure 3.1 and consider the permutation \( \pi_1 = (\text{add}(1)i, \text{get}()k, \text{add}(2)j, \text{remove}(2)i, \text{remove}(1)k, \text{get}()j, \text{add}(3)i) \) consistent with \( \hat{H} \). If \( op = \text{get}()j \) and \( k = 3 \), the 3-cut permutation induced by \( op \) on \( \pi_1 \) is \( \pi_1(3)(op) = (\text{add}(2)j, \text{remove}(2)i, \text{remove}(1)k, \text{get}()j) \).

**Definition 17 (k-cut permutation set induced by \( op_i \))** Given an execution history \( \hat{H} = (H,\prec) \) its permutation set. Given an operation \( op \) of \( H \) and an integer \( k \), the *k-cut permutation set* induced by \( op \) on \( \Pi_{\hat{H}} \), denoted as \( \Pi_{k,\hat{H}}(op) \), is the set of all the *k-cut permutations* induced by \( op \) on each permutation \( \pi \) of \( \Pi_{\hat{H}} \).

As an example, consider the execution history \( \hat{H} \) depicted in Figure 3.1 and consider the permutation set \( \Pi_{\hat{H}} \) of the execution history \( \hat{H} \) shown above. If \( op = \text{get}()j \) and \( k = 3 \), the 3-cut permutation set induced by \( op \) is shown in Figure 3.16.

**Definition 18 (Admissible set for a get() operation)** Given an execution history \( \hat{H} = (H,\prec) \) of a *k*-bounded set object, let \( op = \text{get}() \) be an operation of \( H \). Let \( \hat{U}_{\hat{H},op} \) be the update sub-history induced by \( op \) on \( \hat{H} \) and let \( k \) be an integer. An *admissible set* for \( op \), denoted as \( V_{ad}(op) \), is any set generated by any permutation \( \pi \) belonging to the *k-cut permutation set* \( \Pi_{k,\hat{U}_{\hat{H},op}} \).
as an example, consider the execution history $\hat{H}$ shown in Figure 3.1 and its update sub-history $\hat{U}$ induced by the operation $op = \text{get}(i)$. Given its 3-cut permutation set $\Pi_{3,\hat{U}}$, all the possible admissible sets for $op$ are $V_{ad1} = \emptyset$, $V_{ad2} = \{3\}$, $V_{ad3} = \{1\}$, $V_{ad4} = \{3\}$, $V_{ad5} = \{3\}$, $V_{ad6} = \{1, 2\}$, $V_{ad7} = \{2, 3\}$, $V_{ad8} = \{2, 3\}$, $V_{ad9} = \{2, 3\}$, $V_{ad10} = \{2\}$.

### 3.5.1 Protocol for an Eventually Synchronous System

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

- A set variable denoted $\text{set}$, containing the local copy of the set.

- two integers $\text{update\_sn}_i$ and $\text{get\_sn}_i$ that represent the sequence numbers used by $p_i$ to distinguish, respectively, its successive update operations ($\text{add()}$ and $\text{remove()}$) and $\text{get()}$ operations.

- a variable $\text{running}_i$ that stores the tuple corresponding to the update operation ($\text{add()}$ or $\text{remove()}$) currently executed by the process. It is initialized to a default value $< \text{null, \bot, 0, i}>$.

- A set variable $\text{last\_ops}_i$, that can contain at most $k$ entries, used to maintain an history of recent update operations executed by $p_i$. Such variable contains 4-uples $<\text{type, val, sn, id}>$ each one characterizing an operation of type $\text{type} = \{\text{A or R}\}$ (respectively for $\text{add()}$ and $\text{remove()}$) of the value $\text{val}$, with a sequence number $\text{sn}$, issued by a process with identity $\text{id}$.

- A boolean $\text{active}_i$, initialized to $\text{false}$, that is switched to $\text{true}$ just after $p_i$ has joined the system.

- Two boolean variables $\text{getting}_i$ and $\text{updating}_i$, whose value is true when $p_i$ is executing respectively a $\text{get()}$ operation or an $\text{add()}$/remove$()$ operation.
Cap 3: Set Object under Churn

• Four set variables, denoted $replies_i$, $reply_{to_i}$, $pending_i$ and $dl\_prev_i$ that are used in the period during which $p_i$ joins the system. The local variable $replies_i$ contains the pairs $<sn, ops>$ 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. The set $dl\_prev_i$ is a variable where $p_i$ stores the processes that have acknowledged its inquiry message while they were still joining or while they were accessing the set by issuing the get(). Once $p_i$ becomes active terminating the join, it has to send a reply to all the processes in the $dl\_prev_i$ set to prevent them to be blocked forever.

• An $update\_ack_i$ set used by $p_i$ while adding or removing a value.

The $update\_ack_i$ set is used to remember the processes that have acknowledged its operation.

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 $set_k$), $update\_sn_k = get\_sn_k = 0$, $active_k = false$, $getting_k = false$, $updating_k = false$, and $last\_ops_k = pending_k = replies_k = update\_ack_k = reply\_to_k = dl\_prev_k = \emptyset$.

The join() operation. The algorithm implementing the join operation for a set object, is described in Figure 3.17, and involves all the processes that are currently present (be them active or not).

After having initialized its local variables, the process $p_i$ broadcasts an inquiry$(i, get\_sn_i)$ message to inform the other processes that it enters the system and wants to obtain the history of most recent operations (line 02). Then, after it has received “enough” replies (line 03), $p_i$ creates the “global” history of operations by doing the union of all the partial history received (lines 04-06). Then, $p_i$ adds to the created history, the operations received during the waiting period (line 07) and order the obtained list by using the sort() function (line 08): the sort function simply order the set variable $history_i$ using the lexicographic order on the pairs $(sn, id)$. Now, for each operation in the $history_i$ set, $p_i$ execute the update() procedure (lines 09-11). 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 $reply\_to \neq \emptyset$. $p_i$ sends such a reply message also to the processes in its set $dl\_prev_i$ to prevents them from waiting forever (line 13-15). Finally, $p_i$ returns ok to indicate the end of the join() operation (line 16).

When a process $p_i$ receives a message inquiry$(j, sn_j)$, it answers $p_j$ by sending back a reply$(<update\_sn_i, last\_ops_i>, sn_j)$ message containing its local variables
operation join(i):

(01) \(set, \leftarrow \emptyset; \ update\_sn, \leftarrow 0; \ get\_sn, \leftarrow 0; \ running, \leftarrow \langle \text{null}, \bot, 0, 0 \rangle; \ last\_ops, \leftarrow \emptyset; \ pending\_i, \leftarrow \emptyset; \ replies\_i, \leftarrow \emptyset; \ update\_ack\_i, \leftarrow \emptyset; \ reply\_to\_i, \leftarrow \emptyset; \ dl\_prev\_i, \leftarrow \emptyset; \ active\_i, \leftarrow \text{false}; \ getting\_i, \leftarrow \text{false}; \ updating\_i, \leftarrow \text{false};\)

(02) broadcast inquiry(i, get\_sn);  
(03) wait until\(|replies\_i| < \frac{k}{2}\);  
(04) for each \(\langle \rightarrow, ls \rangle \in replies\_i\) do
(05) history\_i \leftarrow history\_i \cup ls;  
(06) endfor
(07) history\_i \leftarrow history\_i \cup pending\_i;  
(08) sort(history\_i);  
(09) for each \(\langle type, val, sn, id \rangle \in history\_i\) do
(10) execute update\((type, val, sn, id)\);  
(11) end for;  
(12) active\_i \leftarrow \text{true};  
(13) for each \(\langle j, sn \rangle \in (reply\_to\_i \cup dl\_prev\_i)\) do
(14) send reply\((\langle update\_sn, last\_ops, sn \rangle)\) to \(p_j\);  
(15) end for;  
(16) return(ok);  

(17) when inquiry\((j, sn)\) is delivered:
(18) if (active\_i) then send reply\((\langle update\_sn, last\_ops, sn \rangle)\) to \(p_j\);  
(19) if (getting\_i) then send dl\_prev\((j, sn)\);  
(20) endif
(21) if (updating\_i) then send update\((running)\);  
(22) endif
(23) else
(24) reply\_to\_i \leftarrow reply\_to\_i \cup \{j, sn\};  
(25) send dl\_prev\((j, get\_sn)\);  
(26) end if;

(27) when reply\((\langle usn, ops, sn \rangle)\) is received:
(28) if (get\_sn = sn) then replies, \leftarrow replies, \cup \{\langle \text{usn, ops}, sn \rangle\};  
(29) endif

(30) when dl\_prev\((j, sn)\) is received:
(31) dl\_prev\_i \leftarrow dl\_prev\_i \cup \{\langle j, sn \rangle\}.

Figure 3.17: The join() protocol for a k-bounded set object in an eventually synchronous system (code for \(p_i\))
if it is active (line 18-22) and, in case it is accessing the set by means of a get() operation, it sends also a DL_PREV() message or in case it is accessing the set by means of an add() or remove() operation, it sends also an update() message. If $p_i$ is not active, it postpones its answer until it becomes active (line 24 and line 14) and sends a DL_PREV() message (line 25).

When $p_i$ receives a message REPLY($< sn, ops >, sn$) from a process $p_j$, if the reply message is an answer to its INQUIRY($j, sn_j$) it adds the corresponding pair to its set replies$_i$ (line 28).

Finally, when $p_i$ receives a message DL_PREV($j, sn_j$), it adds its content to the set DL_PREV$_i$ (line 31), in order to remember that it has to send a reply to $p_j$ when it will become active (lines 13-15).

**The get() operation.** The algorithms for the get() operation is described in Figure 3.18. A get() operation is a simplified version of the join() operation described so far. Hence, the code of the get() operation, described in Figure 3.18, is a simplified version of the code of the join() operation.

Each get invocation is identified by a pair ($id, sn$) where $id$ is the process index
and \( sn \) is the sequence number of the get \( \text{get}_n \) (line 05)\(^6\). So, \( p_i \) first empties the sets used to store information during the operation (lines 01-04), sets its status \( \text{getting}_i \) to true (line 04) and then it broadcasts a get request \( \text{get}(i, \text{get}_n) \).

After it has received “enough” replies, \( p_i \) recomputes the “global” history by making the union of all the partial history received so far and the operations received during the waiting period (lines 07-11). Then \( p_i \) order the history obtained by invoking the \( \text{sort} \) function (line 12) and it executes all the operations by invoking the \( \text{update} \) procedure (lines 13-15). Finally, \( p_i \) sets the status \( \text{getting} \) to false (line 16) and then returns the set (line 17).

When a process \( p_i \) receives a message \( \text{get}(j, sn_j) \), it answers \( p_j \) by sending back a \( \text{reply}(\langle \text{update}_n, \text{last_ops}_i >, sn_j) \) message containing its local variables if it is active (line 19). If \( p_i \) is not active, it postpones its answer until it becomes active (line 20).

The \text{add}\((v)\) and \text{remove}\((v)\) operations. The algorithms implementing the \text{add}\((v)\) and the \text{remove}\((v)\) operations are shown in Figure 3.19. The structure of the two protocols is basically the same and in the following the term \text{update} operation is sometimes used as synonym for both the operations.

When a process \( p_i \) wants to update (i.e. by adding or removing a value to) its \( k \)-bounded set, it first perform a \text{get}() operation, in order to obtain the most updated sequence number (line 01, line 09), increments the \text{update}_n (line 02, line 10), empties its \text{update}_\text{ack}_i set to store the acknowledgements to the current update (line 03, line 11), sets its state to \text{updating} and stores the 4-uple corresponding to the current operation (line 04, line 12) and then it sends an \text{update}() message to perform the current update (line 05, line 13) and waits until it receives enough \text{ack} (line 06, line 14). Once \( p_i \) is unblocked from the wait statement, it resets its state by setting its variable \text{update}_i to false and its \text{running}_i variable to a default value (line 07, line 15) and finally it returns from the operation (line 08, line 16).

When an \text{update}(\langle \text{type}, \text{val}, sn_j, j >) \) message is delivered to \( p_i \) from some process \( p_j \), if \( p_i \) is not active, it puts the 4-uple corresponding to the current operation into its \text{pending}_i set (line 18) and will process the operation at the end of the \text{join}() , once it will be active (line 07, Figure 3.17) otherwise, \( p_i \) execute the \text{update} procedure for the current operation (line 19).

When an \text{ack}(sn, j) \) message is delivered to \( p_i \) from some process \( p_j \), if the sequence number \( sn \) attached to the message is the same as the current operation then \( p_i \) adds \( j \) to the set of processes that have acknowledged its operation (line 22).

The \text{update}() procedure. The protocol of the \text{update}() procedure is shown in Figure 3.20. Such procedure is called by every operation and its aim is to keep updated the local variables of a process \( p_i \) to the most recent \( k \) operations.

\(^6\)The invocation corresponding to the pair \((i, 0)\) is the \text{join}() operation issued by \( p_i \).
More in detail, when a process $p_i$ execute the $\text{update}(<\text{type},\text{val},\text{sn}_i, j >)$ procedure, it first sends an $\text{ack}(\text{sn}_i, i)$ message to prevent the process $p_i$ to be blocked forever (line 01) and then checks if the current operation is one of the most recent ones (lines 02 - 08). In particular, $p_i$ selects from the recent operations stored in its $\text{last_ops}_i$ variable, the one with the smaller pair $(\text{sn}, \text{id})$ (line 02) and then, if it stores less than $k$ operations, it puts the current one in the $\text{last_ops}_i$ variable (line 03), otherwise $p_i$ substitutes (if possible) the one with smallest pair $(\text{sn}, \text{id})$ with the current one (lines 04 - 07). Once $p_i$ has the correct set of recent operations, it empties the variable $\text{set}_i$ corresponding to the local copy of the set (line 09), sorts the set of recent operations according the lexicographic order of pairs $(\text{sn}, \text{id})$ (line 10) and then executes the operations contained in the $\text{last_ops}_i$ set according to this order (lines 11 - 15). Finally $p_i$ updates its sequence number to the maximum between its own and the one received with the update (line 16).
3.5. \textit{K-BOUNDED SET OBJECT}

\begin{figure}[h]
\centering
\begin{verbatim}
procedure update(<type, val, sn, j>) % at any process \( p_i \)
(01) send ack (sn, i) to \( p_j \);
(02) let \(< -,-, sn, id > \in last_ops; \)
such that \((\forall i, -,-, sn', id' > \in last_ops; : (sn, id) \leq (sn', id'))\);
(03) if \(|last_ops;| < k\) then \( last_ops; \leftarrow last_ops; \cup\{<type, val, sn, j>\} \);
(04) else if \((sn, j) > (sn, id)\)
(05) then \( last_ops; \leftarrow last_ops; \cup\{|< -,-, sn, id >\} \);
(06) last_ops; \leftarrow last_ops; \cup\{<type, val, sn, j>\} ;
(07) endif
(08) endif
(09) set, \leftarrow 0;
(10) sort (last_ops);:
(11) for each \(<type, val, sn, id > \in last_ops;\)
(12) if \((type = A)\) then \( set, \leftarrow set, \cup\{val\} \);
(13) else \( set, \leftarrow set, /\{val\} \);
(14) endif
(15) endfor
(16) update_snj, \leftarrow max(update_snj, snj).
\end{verbatim}
\caption{The \textit{update}() protocol for an eventually synchronous system (code for \( p_i \))}
\end{figure}

3.5.2 Correctness proof

\textbf{Lemma 24} Let \( n \) be the number of processes belonging to the computation at time \( t_0 \). If (i) \( |A(t)| > \lceil \frac{n}{4} \rceil \forall t \) and (ii) a process \( p_i \) invokes a join() operation and does not leave the system for at least \( 3\delta \) time units then it eventually returns from the join() operation.

\textbf{Proof} Let us first observe that, in order to terminate its join() operation, a process \( p_i \) has to wait until its set \( \text{replies}_i \) contains, at least, \( \lceil \frac{n}{4} \rceil \) elements (line 03, Figure 3.17). This set is filled by \( p_i \) when it receives the \textit{reply}() messages for the current operation (line 28 of Figure 3.17). A process \( p_j \) sends a \textit{reply}() message to \( p_i \) if (i) either it is active and has received an \textit{inquiry} message from \( p_i \), (line 18, Figure 3.17), or (ii) it terminates its join() operation and \(< i, - > \in \text{reply}_j \cup \text{dl}_j\) (lines 13-15, Figure 3.17).

Let us suppose by contradiction that \(|\text{replies}_i| \) remains smaller than \( \lceil \frac{n}{4} \rceil \). This means that \( p_i \) does not receive enough \textit{reply}() carrying the appropriate sequence number. Let \( t \) be the time at which the system becomes synchronous and let us consider a time \( t' > t \) at which a new process \( p_j \) invokes the join operation. At time \( t' \), \( p_j \) broadcasts an \textit{inquiry} message (line 02, Figure 3.17). As the system is synchronous from time \( t \), every process present in the system during \([t', t' + \delta]\) receives such \textit{inquiry} message by time \( t' + \delta \).

As \( p_i \) is not active yet when it receives \( p_j \)'s \textit{inquiry} message, the process \( p_i \) executes line 25 of Figure 3.17 and sends back a \textit{dl}_j message to \( p_j \). Due to
the assumption that every process that joins the system remains inside for at least 3δ time units, \( p_j \) receives \( p_i \)'s \( \text{dl}_\text{prev} \) and executes consequently line 31 (Figure 3.17) adding \(<i, -> \) to \( \text{dl}_\text{prev} \). Due to the assumption that there are always at least \( \lceil n^2 \rceil \) active processes in the system, we have that at time \( \tau' + \delta \) at least \( \lceil n^2 \rceil \) processes receive the \( \text{inquiry} \) message of \( p_j \), and each of them will execute line 18 (Figure 3.17) and send a \( \text{reply} \) message to \( p_j \). Due to the synchrony of the system, \( p_j \) receives these messages by time \( \tau' + 2\delta \) and then stops waiting and becomes active (line 12, Figure 3.17). Consequently (lines 13-15) \( p_j \) sends a \( \text{reply} \) to \( p_i \) as \( i \in \text{reply}_to_j \cup \text{dl}_\text{prev} \). By \( \delta \) time units, \( p_i \) receives that \( \text{reply} \) message and executes line 28, Figure 3.17. Due to churn rate, there are an infinity of processes invoking the join after time \( t \) and \( p_i \) will receive a reply from any of them so \( p_i \) will fill in its set \( \text{replies}_i \) and terminate its join operation.

**Lemma 25** Let \( n \) be the number of processes belonging to the computation at time \( t_0 \). If (i) \( |A(t)| > \lceil \frac{n}{2} \rceil \) \( \forall t \) and (ii) each process that invokes a \( \text{join}() \) operation does not leave the system for at least 3δ time units then a process \( p_i \) that invokes a \( \text{get}() \) operation and does not leave the system eventually returns from such operation.

**Proof** Since the \( \text{get}() \) is a simplified case of a \( \text{join}() \), the proof is the same of Lemma 24.

**Lemma 26** Let \( n \) be the number of processes belonging to the computation at time \( t_0 \). If (i) \( |A(t)| > \lceil \frac{n}{2} \rceil \) \( \forall t \) and (ii) each process that invokes a \( \text{join}() \) operation does not leave the system for at least 3δ time units then a process \( p_i \) that invokes an \( \text{add}() \) operation or a \( \text{remove}() \) operation and does not leave the system eventually returns from such operation.

**Proof** Let us first assume that the \( \text{get}() \) operation invoked at line 01 and line 09 terminates (this is proved in Lemma 25). Before terminating the \( \text{add}() \) (or \( \text{remove}() \)) of a value \( v \) with an update sequence number \( \text{usn} \) a process \( p_i \) has to wait until its set \( \text{update}_\text{ack}i \) contains at least \( \lceil \frac{n}{2} \rceil \) elements (line 06 and line 14, Figure 3.19).

Empty at the beginning of the \( \text{add}(v) \) operation (line 03, Figure 3.19) and at the beginning of the \( \text{remove}(v) \) operation (line 11, Figure 3.19), this set is filled in when the \( \text{ack}(\text{usn}, -) \) messages are delivered to \( p_i \) (line 22, Figure 3.19).

Such an ack message is sent by every process \( p_j \) when the \text{update} procedure is activated due to (i) the receipt of an update message from a process \( p_i \) (line 19, Figure 3.19) or (ii) the termination of a \( \text{join}() \) operation and \( <-, -, \text{usn}, i > \in \text{history}_j \cup \text{pending}_j \) (lines 09 - 11, Figure 3.17).

Suppose by contradiction that \( p_i \) never fills in \( \text{update}_\text{ack}i \). This means that \( p_i \) misses \( \text{ack}() \) messages carrying the sequence number \( \text{usn} \). Let us consider the time \( t \) at which the system becomes synchronous, i.e., every message sent by any process
$p_j$ at time $t' > t$ is delivered by time $t' + \delta$. Due to the assumption that the process issuing the update does not leave before the termination of its operation, it follows that $p_j$ will receive all the \texttt{inquiry} messages sent by processes joining after time $t$.

When it receives an \texttt{inquiry()} message from some joining process $p_j$, $p_i$ executes lines 18-23 of Figure 3.17 and sends a \texttt{reply} message to $p_j$ as reply to its request and forwards an \texttt{update} message with the sequence number $usn$.

Since, after $t$, the system is synchronous, $p_j$ receives both the \texttt{reply} message and the \texttt{update} message in at most $\delta$ time units. When $p_j$ receives the \texttt{update} message, since it is not yet active, it puts the current update $\langle -, -, usn, i \rangle$ in the $pending_j$ set (line 18, Figure 3.19). Due to Lemma 24 $p_j$ will terminate the join executing lines 07-11 by processing the pending update (and also the one of $p_j$) and sending the \texttt{ack}($ubn$, $-$) message to $p_i$ (line 01, Figure 3.20).

As (1) by assumption a process that joins the system does not leave for at least $3\delta$ time units and (2) the system is now synchronous, such an \texttt{ack}($usn$, $-$) message is received by $p_i$ in at most $\delta$ time units and consequently $p_i$ executes line 22, Figure 3.19 and adds $p_j$ to the set $update\_ack_i$.

Due to the dynamicity of the system, processes continuously join the system. Due to the chain of messages \texttt{inquiry()}, \texttt{update()}, \texttt{ack}(), the reception of each message triggering the sending of the next one, it follows that $p_i$ eventually receives $\lceil n^2 \rceil$ \texttt{ack}($usn$, $-$) messages and terminates its update operation.

\begin{proof}
It follows from Lemma 24, Lemma 25 and Lemma 26.
\end{proof}

\textbf{Theorem 8} Let $n$ be the number of processes belonging to the computation at time $t_0$. If (i) $|A(t)| > \lceil \frac{n}{2} \rceil \forall t$ and (ii) each process that invokes a $\texttt{join()}$ operation does not leave the system for at least $3\delta$ time units then a process $p_i$ that invokes a $\texttt{join()}$, $\texttt{get()}$, $\texttt{add()}$ or $\texttt{remove()}$ operation and does not leave the system eventually terminates its operation.

\begin{proof}
It follows from Lemma 24, Lemma 25 and Lemma 26.
\end{proof}

\textbf{Lemma 27} Let $\tilde{H} = (H, <)$ an execution history of a $k$-bounded set object $S$. Given the algorithms shown in Figures 3.17-3.20, there exists a total order of the pairs $(sn, id)$ that identify each $\texttt{add}(v)$ and $\texttt{remove}(v)$ operation such that the total order is consistent with the partial order given by the execution history $\tilde{H}$.

\begin{proof}
Let $op$ and $op'$ two update operations and let $sn(op)$ and $sn(op')$ the corresponding sequence numbers given to the operations by the algorithm.

---

1. The process $p_i$ that issues an update operation always executes lines 18-21 of Figure 3.17 because it is always in the active mode.
Case 1: $op \prec op' \Rightarrow sn(op) < sn(op')$. Let us suppose by contradiction that the two sequence numbers are the same. Without loss of generality, let $p_i$ be the process that issues $op$, $p_j$ be the process issuing $op'$ and let $x$ be the sequence number associated by $p_i$ to its operation.

- $\forall op'' : op \prec op'' \prec op'$: when $p_i$ starts an update operation (both $\text{add}(v)$ or $\text{remove}(v)$), it sends an $\text{UPDATE}(<-, -, x, i>)$ message and waits until it receives at least $(\frac{n}{2} + 1)$ $\text{ack}$ messages carrying the sequence number $x$ (line 06 and line 14, Figure 3.19). This means that in the system there exists at least $(\frac{n}{2} + 1)$ processes that executes line 16, Figure 3.20 and update their $\text{update_sn}_k$ to a value that is greater equal to $x$. When $p_j$ starts its operation, it first issues a $\text{get}()$ to retrieve the most up-to-date sequence number (line 01 and line 09, Figure 3.19). During the $\text{get}()$ operation, processes sent to $p_j$ their local copies of the list $\text{last_ops}_k$ and to terminate the $\text{get}()$, $p_j$ executes the $\text{UPDATE}$ procedure by updating its $\text{update_sn}_j$ variable. Since $|A(t)| < \lceil \frac{n}{2} \rceil \ \forall t$ there exists at least one process that has seen the operation of $p_j$ and that send a sequence number that is at least $x$. Doing the update, $p_j$ always select the maximum sequence number and at the end of the $\text{get}()$ it has in its $\text{update_sn}_j$ variable a value grater than $x$. Since before sending the $\text{UPDATE}$ message $p_j$ increments its sequence number (line 02 and line 10, Figure 3.19) the operation of $p_j$ will have a sequence number grater that $p_i$ and there is a contradiction.

- $\exists op'', \ldots, op^i : op \prec op'' < \cdots < op^i \prec op'$: in this case, the statement simply follows by the point above observing that the operator “$<$” is transitive (i.e. if $sn(op) < sn(op'')$ and $sn(op'') < sn(op')$ then $sn(op) < sn(op')$).

Case 2: $op || op'$. In this case, the two sequence numbers assigned by the algorithm to the operation, depend by the delivery order of the broadcast messages sent during the operation. In case of concurrency of the broadcasts, sequence numbers can be the same. However, in the $\text{UPDATE}$ procedure, the operations are ordered and executed according to the pairs $(sn, id)$ and since the process identifiers are unique in the system, there exists a total order also among concurrent operations.

\[\Box\] Lemma 27

Lemma 28 Let $\hat{H} = (H, \prec)$ an execution history of a $k$-bounded set object $S$ and let $p_i$ be a process that invokes a $\text{join}()$ operation at time $t$. Let $op$ be an instantaneous $\text{get}()$ operation issued at time $t_E(\text{join})$ and $\hat{U}$ be the update sub-history of $\hat{H}$ induced by $op$. Given an integer $k$, if $|A(t)| > \lceil \frac{n}{2} \rceil \ \forall t$, then at the end of the $\text{join}()$ operation, $p_i$ maintains in its $\text{last_ops}_i$ variable a $k$-cut permutation induced by $op$ consistent with $\hat{U}$.

Proof (Sketch)
In order to terminate the join(), a process \( p_i \) has to wait until it receives at least \( \left\lceil \frac{n}{2} \right\rceil + 1 \) replies from other processes (line 03, Figure 3.17). Each reply received by \( p_i \) from a process \( p_j \) contains the list of the most recent operations saw by \( p_j \).

**Step 1.** Let \( p_i \) be the first process that terminates the join() operation. Let \( O \) be the set of operations started before the end of the join() operation.

1. \(|O| \leq k\). Since \( p_i \) is the first process that concludes the join() operation, the set of active processes at the end of the join() is a subset of the processes that were active at time \( t_0 \) (i.e. \( A(t_E(\text{join})) \subseteq A(t_0) \)). Therefore, for each update operation \( op \) belonging to \( O \), there exists at least one process \( p_k \) that has sent an \( \text{ack}() \) message for \( op \) (line 01, Figure 3.20), that has inserted a tuple \( <-,-,sn,id> \) in its \( \text{last} \_\text{ops}_k \) variable (line 03, Figure 3.20) and that has sent a \( \text{REPLY}() \) message to \( p_i^8 \). When \( p_i \) executes lines 04-06 of Figure 3.17, it obtains an \( \text{history}_i \) variable containing all the operations terminated before the join() invocation, and possibly some of the concurrent ones. Executing line 08 of Figure 3.17, \( p_i \) will obtain an history totally ordered by the pairs \( (sn,id) \) and due to Lemma 27 this total order is consistent with the partial order of the execution history \( \hat{U} \). Since the \( \text{UPDATE} \) procedure is executed following such total order and considering that all the operations will be stored in the \( \text{last} \_\text{ops}_i \) variable, it is possible to conclude that at the end of the join() operation, the \( \text{last} \_\text{ops}_i \) variable contains a k-cut permutation consistent with \( \hat{U} \).

2. \(|O| > k\). Let consider the worse case where there exists a process \( p_j \) that replies to all the \(|O| \) operations. Note that, for the first \( k \) \( \text{UPDATE} \) messages received by \( p_j \), it always executes line 03 of Figure 3.20 because it has empty slots in its \( \text{last} \_\text{ops}_j \) variable. When \( p_j \) receives the \((k+1)\)-th \( \text{UPDATE} \) message, it executes line 04, Figure 3.20 and checks if the current update is “new” with respect to the ones contained in its \( \text{last} \_\text{ops}_j \) variable, and if it is so, it substitutes the one with the smaller pair \( (sn,id) \) with the new one (line 05-06, Figure 3.20). Note that substituting the update operation identified by the smaller pairs \( (sn,id) \), the total order is preserved. Since \( p_i \) is the first process that concludes the join() operation, the set of active processes at the end of the join() is a subset of the processes that were active at time \( t_0 \) (i.e. \( A(t_E(\text{join})) \subseteq A(t_0) \)). Therefore, for each update operation \( op \) belonging to \( O \), there exists at least one process \( p_k \) that has sent an \( \text{ack}() \) message for \( op \) (line 01, Figure 3.20), that has inserted a tuple \( <-,-,sn,id> \) in its \( \text{last} \_\text{ops}_k \) variable (line 03, Figure 3.20). Considering that an operation \( op \) is deleted by the \( \text{last} \_\text{ops}_k \) variable from some process \( p_k \) iff there exist \( k \) operations that follows \( op \) in the total order, and that all the operations have been acknowledged by at least one

\(^8\text{Note that, since the total number of operations is smaller than } k, \text{ processing the } \text{UPDATE} \text{ message, each process always executes line 03, Figure 3.20.} \)
process that replies to \( p_i \), follows that even in this case at the end of the \( \text{join}() \) operation, the \( \text{last_ops}_i \) variable contains a \( k \)-cut permutation consistent with \( \hat{U} \).

**Step i.** Let \( p_i \) be the \( i \)-th process that terminates the \( \text{join}() \) operation. Let \( \mathcal{P} \) be the set of processes that send a \textit{REPLY} message to \( p_i \). The set \( \mathcal{P} \) can be partitioned in the set of processes that were active at time \( t_0 \) and the set of processes that became active at some \( t > t_0 \). Note that all the processes that were active at time \( t_0 \) will send to \( p_i \) a copy of their \( \text{last_ops}_k \) variable where, for each new update, they substitute the “older” operation and then, their list represents \( k \)-cut permutation consistent with \( \hat{U} \). Each process that became active after \( t_0 \) has received a list from some other processes. Iterating the reasoning we come back to the situation of step 1 and the Lemma follows.

**Lemma 29** Let \( \hat{H} = (H, <) \) an execution history of a \( k \)-bounded set object \( S \) and let \( p_i \) be a process that invokes a \textit{get()} operation \( op \). Let \( \hat{U} \) be the update sub-history of \( H \) induced by \( op \). Given an integer \( k \), if \( |A(t)| > \lceil \frac{n}{2} \rceil \forall t \), then at the end of the \textit{get()} operation, \( p_i \) maintains in its \( \text{last_ops}_i \) variable a \( k \)-cut permutation induced by \( op \) consistent with \( \hat{U} \).

**Proof**

In order to terminate the \textit{get()} , a process \( p_i \) has to wait until it receives at least \( (\frac{n}{2} + 1) \) replies from other processes (line 06, Figure 3.18). Each reply received by \( p_i \) from a process \( p_j \) contains the list of the most recent operations saw by \( p_j \). Let \( \mathcal{P} \) be the set of processes that send a \textit{REPLY} message to \( p_i \). The set \( \mathcal{P} \) can be partitioned in the set of processes that were active at time \( t_0 \) and the set of processes that became active at some \( t > t_0 \).

All the processes that were active at time \( t_0 \) will send to \( p_i \) a copy of their \( \text{last_ops}_k \) variable where, for each new update, they substitute the “older” operation (i.e. the operation identified by the smallest pair \((sn, id)\)) and then, their list represents \( k \)-cut permutation induced by the \textit{get()} consistent with \( \hat{U} \).

Each process \( p_j \) that became active after \( t_0 \), due to Lemma 28, terminates its \textit{join()} operation maintaining in its local variable \( \text{last_ops}_j \) a \( k \)-cut permutation consistent with \( \hat{U} \) at time \( t_E(\text{join}) \) and then for each new update, it substitutes the “older” operation. Hence, also for such processes the list sent to \( p_j \) represents a \( k \)-cut permutation consistent with \( \hat{U} \) and the Lemma follows.

**Theorem 9** Let \( S \) be a \( k \)-bounded set object and let \( op \) be a \textit{get()} operation issued on \( S \) by some process \( p_i \). If \( |A(t)| > \lceil \frac{n}{2} \rceil \forall t \), the set \( V \) returned by \( op \) is an admissible set (i.e. \( V = V_{ad}(op) \)).
3.5. K-BOUNDED SET OBJECT

Proof Let us suppose by contradiction that the set $V$ returned by a $\text{get}(\cdot)$ operation $op$ is not an admissible set. The set $V$ is calculated in lines 09-15, Figure 3.20. Due to Lemma 29, such operations represents a $k$-cut permutation induced by $op$ on $\hat{U}$ and let’s call $\pi_i$ such permutation.

If $V$ is not an admissible set for $op$, it means that it is not generated by any $\pi_i \in \Pi_{k,\hat{U}(op)}$ and then one of the following conditions holds:

1. there exists a value $v$, generated by every permutation $\pi_i$ of the permutation set $\Pi_{\hat{U}}$, such that $v$ does not belong to $set_i$ (i.e. $\exists v : \forall \pi_i \in \Pi_{k,\hat{U}(op)} (\exists \text{add}(v) \in \pi_i : \text{add}(v) \rightarrow_{\pi_i} op) \land (\forall \text{remove}(v) \in \pi_i : \text{add}(v) \rightarrow_{\pi_i} remove(v) \rightarrow_{\pi_i} op) \land (v \notin set_i)$);

2. there exists a value $v$ belonging to $set_i$ such that it can not be generated by any permutation $\pi_i$ of the permutation set $\Pi_{\hat{U}}$ (i.e. $\exists v \in set_i : \forall \pi_i \in \Pi_{k,\hat{U}(op)} (\exists \text{add}(v), \text{remove}(v) \in \pi_i : \text{add}(v) \rightarrow_{\pi_i} \text{remove}(v) \rightarrow_{\pi_i} op)$).

Case 1. If there exists an $\text{add}(v)$ in the permutation $\pi_i$ it means that in the last_opsi variable there exists a tuple $<A,v,-,->$ and then $p_i$ will execute line 12 of Figure 3.20. Since the last_opsi variable is ordered according to the partial order of the execution and since there not exists any tuple $<R,v,-,->$ in the last_opsi ordered variable after $<A,v,-,->$ it follows that after executing line 12 of Figure 3.20, $p_i$ will not execute 13 of Figure 3.20 in the following and then $v$ will be in the seti variable returned having a contradiction.

Case 2. At line 09 of Figure 3.20, the variable seti that will be returned is emptied. If $v \in V$ (i.e. $v \in set_i$), it means that $p_i$ has executed line 12 of Figure 3.20 and after it has not executed line 13 of Figure 3.20. If $p_i$ has executed line 12 and after it has not executed line 13 it means that in the last_opsi variable there exists a tuple $<A,v,-,->$ and there is not any tuple $<R,v,-,->$ coming after. Due to Lemma 29 all the operations in the last_opsi variable represent a $k$-cut permutation consistent with $\hat{U}$ and then it follows that there exists an $\text{add}(v) \in \pi_i$ and there is not any $\text{remove}(v) \in \pi_i : \text{add}(v) \rightarrow_{\pi_i} \text{remove}(v)$ and we have a contradiction.

\[\square\text{Theorem 9}\]

Theorem 10 Let $S$ be a $k$-bounded set object and let $\hat{H} = (H,\prec)$ be an execution history of $S$ generated by the algorithms in Figures 3.17 - 3.20. If every active process $p_i$ maintains an admissible $k$-bounded set, $\hat{H}$ is always value-based sequential consistent.

Proof The proof follows immediately by Lemma 27. Since update operations are ordered according to the pair $(sn,id)$, each process executing line 12 of Figure 3.18 will order all the operations in the same way.

\[\square\text{Theorem 10}\]
3.6 $k$-bounded Set Applications

In this section, are presented two applications built on top of the $k$-bounded set, namely eventual participant detector and shared list.

3.6.1 Eventual Participant Detector

An eventual participant detector is an oracle that, when queried, returns the composition of the group. Such an oracle can be characterized by two properties:

- **Completeness**: eventually, every process that leaves permanently the group is no more returned by the oracle;

- **Accuracy**: eventually, each process that remains forever in the group is always returned by the oracle.

The basic idea is to use the $k$-bounded set as repository for the identifiers of processes that decide to participate to the group. In this case, the parameter $k$ is function of the number of processes part of the group. Higher is the number of participants that eventually will be permanently part of the group, higher is the value of $k$ that have to be used.

When a process decides to join the group, it simply joins the $k$-bounded set computation and as soon as it returns from the join, it puts its identifier in the repository by invoking the $\text{add()}$ operation on the $k$-bounded set and repeats periodically such operation. This is actually needed because a $k$-bounded set object keeps memory only of recent update operations. When a process leaves the group, it simply stops to add its identifier in the repository. When a process wants to have the current membership...
3.6. \textit{K-BOUNDED SET APPLICATIONS}

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{operation} join\_shList(i) \\
(01) \hspace{1em} list, \leftarrow 0; \\
(02) \hspace{1em} participate, \leftarrow \text{true}; \\
(03) \hspace{1em} join(); \\
(04) \hspace{1em} return(ok). \\
\hline
\textbf{operation} leave\_shList(i) \\
(01) \hspace{1em} participate, \leftarrow \text{false}; \\
(02) \hspace{1em} return(ok). \\
\hline
\textbf{operation} get\_shList() \\
(01) \hspace{1em} list, \leftarrow \text{get}(); \\
(02) \hspace{1em} return(list). \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{operation} add\_shList(v) \\
(01) \hspace{1em} \textbf{while} (participate,) \textbf{every} \Delta \text{time} \\
(02) \hspace{1em} add(v); \\
(03) \hspace{1em} \textbf{endWhile} \\
\hline
\textbf{operation} rem\_shList(v) \\
(01) \hspace{1em} \textbf{while} (participate,) \textbf{every} \Delta \text{time} \\
(02) \hspace{1em} remove(v); \\
(03) \hspace{1em} \textbf{endWhile} \\
\hline
\end{tabular}
\end{center}

of the group it executes the get operation. Let us remark that in a eventual participant
detector, each identifier is added or removed by only one process.
In Figure 3.21 it is shown the pseudo-code for an eventual participant detector.

The intuition of the eventual participant detector correctness follows from the ob-
servation that eventually each \texttt{add()} and \texttt{remove()} operation issued on the \textit{k}-bounded
set will terminates in a bounded time.
Thus, computing the maximum time needed to complete such operations in the syn-
chrony periods and knowing the maximum number of processes participating to the
group, it is possible to select the right value of \textit{k} such that each operation is forgotten
only when it is newly issued (cfr. \texttt{PERSISTENCE()} procedure).

3.6.2 \textbf{Shared List}

A Shared List extends the semantic of a participant detector by allowing each pro-
cess to add and to remove any value from the domain of the set (e.g. a subset of
integers) using the operations \texttt{add\_shList} and \texttt{rem\_shList}. Moreover process can get
the content of SharedList by executing a \texttt{get\_shList}. A shared list \textit{SL} can thus be
characterized by the following properties:

- \textbf{Eventual Value Persistence:} if there exists a time \(t\) after that a value \(v\) be-
longing to \(SL\) is no longer removed from the list (i.e., no more \texttt{rem\_shList}(v)
are executed), then there exists a time \(t' > t\) such that \(v\) will belong to the list
returned by any \texttt{get\_shList} operation;
• **Eventual Value Removal:** if there exists a time $t$ after that a value $v$ is no longer added in $SL$ by any process (i.e., no more $add\_shList(v)$ are executed), then there exists a time $t' > t$ such that $v$ will no longer belong to the list returned by any $get\_shList$ operation.

The basic idea of the algorithm, shown in Figure 3.22, is that when a process executes a $add\_shList(v)$ (resp $rem\_shList(v)$), it actually calls a $add()$ (resp. $remove()$) on the $k$-bounded set. Note that to ensure the two properties, each operation has to be repeated periodically. Also a shared list could be used for implementing a blacklist system in a collaborative large scale environment for security. However, in this case, the result obtained by getting elements from the shared object are only an approximation of the real set of bad processes: higher is the value of $k$, better is the approximation of the black list.
Conclusion

This thesis has provided a study of two shared objects, namely a regular register and a set, in a distributed system prone to continuous churn. The main results are (i) the bounds on the arrival and departure rates following from the implementation of the objects (where an implementation exists) and (ii) the impossibility to implement such shared objects under some timing assumption (i.e. asynchrony in the case of the regular register and partially synchrony for the set), using finite memory, unless the system becomes static.

To this aim, a generic churn model, based on deterministic joining and departure distributions, has been proposed and used to find bounds on the maximum churn rate tolerated to ensure that objects behave correctly with respect to their specifications. This model is general enough to have the infinite arrival model and the constant size system as special cases.

The first lesson learnt, dealing with churn, is that to cope with the autonomic behavior of processes, specific procedures (i.e. join and leave procedures) need to be defined.

In the particular case of shared objects, the aim of such join procedure, is to “align” entering processes to the computation, by transferring the current state of the object. In this thesis, the same approach has been used to define the join() procedure, for both regular register and set object. The general idea is to have a heavy procedure (i.e. the join), executed once by each process and lightweight protocols for the operations accessing the objects (and then executed often). Other protocols may be defined, using lightweight join but, the counterpart is in the major cost of other operations (all the operations have to be based on request-reply pattern, while actually, in synchronous systems, they can be executed locally or in pull mode).

Note that, the join operation is the one that constrains the join and leave rates and then, it is possible to say that there exists a trade-off between the level of churn that can be tolerated and the complexity of the protocols implementing the shared objects.

Another interesting point, following by the analysis of dynamic systems, is the conjecture that having two degrees of uncertainty, i.e. the asynchrony and the continuous churn, makes no possible the implementation of shared object abstractions: at
least one of the two has to be quiescent.
In fact, even considering simple shared objects, like registers, an implementation exists if the churn is quiescent and the system is asynchronous [3] or if the system is eventually synchronous and the churn is continuous but it is impossible to find an algorithm, based on message-passing, assuming fully asynchrony and continuous churn (cfr. Chapter 2).

It is also important to note that among shared objects, not all of them can be implemented, in a dynamic system, under the same synchrony assumption. An explicative example is given by the impossibility to build a set object with continuous accesses and churn in an eventually synchronous system, while under the same assumptions a register can be implemented.
This is actually due to the fact that the semantics of a set object imposes to any implementation to maintain all the information about the past history, while a register is an object, whose state depends only on the last operations. From this observation, follows the intuition that any other complex object (i.e. queue, stack, lists etc...) cannot be implemented under the same assumptions. It is important to remark that the impossibility comes from the combination of continuous churn, unbounded number of accesses to the object, finite memory space and asynchrony periods. Under this assumption, it has been shown that only a weaker object, namely a $k$-bounded set, approximating the behavior of the set can be implemented.

Concluding, this thesis has taken a first step towards the analysis of problems and abstractions that can be implemented in a distributed system subject to churn. However, a lot of work remains to do trying to answer to the following questions:

- can the proposed approach be refined and extended to consider explicitly failures?

- can the proposed approach be refined and extended to consider mobility?

- Does the proposed solution remain correct if byzantine processes belong to the computation? How the bounds on the churn rate change to deal with such further uncertainty form?

- Can the proposed approach be extended to solve other distributed problems like mutual exclusion, leader election etc...?
Bibliography


