Implementing Set Objects in Dynamic Distributed Systems

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

[1] *Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti*
Universitá degli Studi di Roma "La Sapienza"
Via Ariosto 25, I-00185 Roma, Italy
e-mail: {baldoni, bonomi}*@dis.uniroma1.it*

[2] *Institut Universitaire de France, IRISA,*
Université de Rennes, Campus de Beaulieu, F-35042 Rennes, France
e-mail: raynal@irisa.fr

Abstract

This paper considers a *set object*, i.e., a shared object allowing users (processes) to add and remove elements to the set, as well as taking consistent snapshots of its content. Specifically, we show that there not exists a protocol implementing a set object using finite memory if the underlying distributed system is eventually synchronous and affected by continuous arrivals and departures of processes (phenomenon also known as *churn*). Then, we analyze the relationship between system model assumptions and object specification in order to design protocols implementing the set object using finite memory. Along one direction (strengthening the system model), we propose a protocol implementing the set object in synchronous distributed systems and, along the other direction (weakening the object specification), we introduce the notion of a k*-bounded set object* proposing a protocol working on an eventually synchronous system.

Notes: Some results presented in this paper has appeared in a preliminary form at EUROPAR 2010 in a paper entitled *Value-based Sequential Consistency for Set Objects in Dynamic Distributed Systems*.

Corresponding Author: Silvia Bonomi
Phone: +39 06 7727 4017
Fax: +39 06 7727 4002

1. Introduction

Data structures (such as stacks, queues, files, etc.,) are fundamental notions of computing science. As indicated by their names, they encapsulate data in a structured way and each of such structures returns an appropriate answer to each client's request according to the specification of the data structure (i.e., consistency). In a concurrent or distributed context, such data structures are called *concurrent objects* and the consistency issues become more complex to be tackled. Another important issue of these data structures is their availability, that is, the property of providing a response in a finite time to requests coming from clients, despite possible failures. Additionally, this response should be fast rather than slow. High availability and fast responses have been classically achieved through data replication and in-memory processing [34]. In such case, a set of servers cooperates (i.e., they executes a computation) in order to give the illusion to clients of a single highly available data structure. If we consider such objects being implemented on the top of modern large scale distributed infrastructures or on top of pervasive systems, we have that the implementation has to cope with continuous arrivals and departures of processes (phenomenon also know as *churn*) caused by, for example, failures, voluntary leaves etc. Therefore, the object implementation should be able to tolerate continuous churn without compromising the consistency of the object. The paper focuses on the study of the design of a specific data structure, namely the *set object*, and its implementation in the presence of continuous churn.

A set object $\mathcal{S}$ is a shared object that stores a (possibly empty) finite set of elements. A process can acquire the content of $\mathcal{S}$ through a get() operation while it can add (respectively remove) an element to $\mathcal{S}$ through an add() (resp. remove()) operation. The set object has to satisfy a strong consistency requirement that will be carefully specified later in the text. Informally, strong consistency means that once an add() (resp. remove()) operation returns successfully to the client, all subsequent get() operations see the effects of that add() (resp. remove()) operation. The paper analyzes the problem of implementing a set object using finite memory in the presence of continuous churn. In the setting considered in this paper, indeed, a straightforward implementation of the set object done through a bounded number of registers is impossible [7], thus new approaches have to be envisaged.

Contribution of the paper. The paper provides an in-deep investigation of a set object. Specifically, the paper provides the following contributions:

- A new consistency condition, well-suited for a set object, called *per-element sequential consistency* is introduced. While it allows concurrent get() operations to return the same output in the absence of concurrent add() and/or remove() operation, this condition is weaker than linearizability [13]. This is because processes are required to see the same order only of concurrent add() and remove() operations that act on a same element. Concurrent operations executed on distinct elements can be perceived in different order by distinct processes.

- An impossibility result for the implementation of a set object using finite memory. More specifically, considering a distributed system affected by continuous churn and characterized by the absence of "stable processes" (i.e., processes remaining forever in the distributed system), it is shown that there not exists a message-passing protocol implementing a set object using finite memory in an eventually synchronous system.

- A study of the relationship between system model assumptions and object specification to design distributed protocols implementing the set object with finite memory.
  - We consider a synchronous distributed system affected by continuous churn and we provide a protocol implementing a set object using a finite memory space and we prove its correctness.
  - We consider a weaker form of set object called k-*bounded set object* and provide an implementation on top of an eventually synchronous distributed system prone to continuous churn along with its correctness proof. Informally, a k-bounded set object is a set that has limited memory of the execution history. In particular, given a $\mathsf{get}()$ operation, a k-bounded set behaves as a set where the execution history is limited only to the k most recent update operations.

Roadmap. The rest of the paper is structured as follows Section 2 discusses the related work, Section 3 presents the system model, Section 4 presents the set object specification and the per-element sequential consistency criterion. Section 5 presents an impossibility result for eventually synchronous systems. Then, Section 6 presents two protocols: one implementing a set in a synchronous system and another one implementing a k-bounded set object in an eventually synchronous system. Finally, Section 7 concludes the paper.

2. Related Work

Several works have been done recently in the general context of dynamic distributed systems. Some modeling the concept of churn [14, 18], others address the implementation of concurrent data structures, namely a register data structure, in message passing distributed systems [2, 4, 20].

Churn Models. The concept of churn has been initially formalized through *infinite arrival models* introduced by Merritt et al. [23] and subsequently refined by Aguilera [1]. Such models are able to capture the evolution of the network removing the constraint of having a predefined and constant size n. However, such models do not give any indication on how the joins or the leaves happen during time.

A different approach is to make implicit the notion of churn by abstracting it through the definition of a set of *configurations* [20, 2]. A configuration is represented by the current set of replicas running the algorithm and a new configuration is created as soon as join or leave operations occur. In particular, Lynch et al., [20] manage configuration changes by using a consensus primitive while Aguilera et al., [2] define a reconfiguration procedure where a new configuration is suggested by the participants. As in the infinite arrival models, configurations have the main advantage of considering a distributed system where the number of replicas involved can change along time. However, they need a certain degree of knowledge about the participants to move from a configuration to the new one and are particularly suited to model quiescent churn (i.e., to model dynamic systems where arrivals and departures eventually end).

More recently, other models have been proposed to take into account the process behavior. This is done by considering both probabilistic distribution [18], or deterministic distribution [14], on the

join and leave of replicas (in both cases the system size remains constant). As a consequence, the main advantage of such models is to allow an analytical evaluation of the churn and they can be used to represent both continuous and quiescent churn.

Regular Registers in Dynamic Distributed Systems. Lynch et al., present in [20] the implementation of an *atomic read/write* object where the churn is abstracted by the notion of system reconfiguration. A reconfiguration could happen every time that a process joins or leaves the system. To be valid a reconfiguration requires processes to agree upon a unique sequence of configuration changes. This agreement implies the need of consensus and thus cannot be implemented in a fully asynchronous system. Starting from this result, Aguilera et al., [2] shown that it is possible to realize an atomic read/write object without using consensus. In particular, the authors shown that an atomic register can be implemented on a fully asynchronous distributed system provided that the number of reconfiguration operations is finite and that there is a majority of correct replicas in each reconfiguration. Thus, the churn is quiescent and confined in specific time intervals[1]. Finally in [4], we proved that if the churn is not quiescent, it is not possible to implement a regular register in a fully asynchronous system.

From Registers to Sets. Baldoni et al. [5] provided an implementation of a regular register in an eventually synchronous distributed system with continuous churn, while in Section 5 we will show that a set object is impossible to be realized in the same churn condition. The impossibility stems from the non-quiescence of the set accesses. In a register implementation, in fact, such a dimension does not play any role as a register does not need to store anything else than the last written value. This is not true for a set, which must be able to reconstruct the update operation history. If the update operations are continuous, this implies that a set implementation needs infinite memory.

Shapiro et al. [28] introduce the notion of *strong eventual consistency* and consider how this consistency criterion can be supported by emulating generic data types in a distributed network prone to process crashes and recoveries. The proposed solution is then applied to a set object implementation where processes keep their memory intact after a recovery (e.g., by storing data on a stable storage). Although the object considered is the same, the model and the consistency condition considered in [28] are different from the ones addressed in this paper. In particular, this paper assumes that replicas use only volatile memory, i.e., when a process fails (leaves), it looses the content of the set, and if a process rejoins the computation, it has no knowledge about the state of the set (i.e., no stable storage is used to preserve the set state). In addition, we consider a strong consistency condition that requires replicas to agree on the state of the set at any time, while this agreement is required only eventually in [28].

Wuu et al., considered the implementation of a distributed dictionary in a system prone to crashes and recoveries [33]. A dictionary is an abstraction of a set data type, which differs form the one considered in this paper due to the following limitations: (i) $\mathsf{delete}(v)$ operation can be invoked only if the element v is already contained in the dictionary, and (ii) every element v can be inserted in the dictionary at most once. The authors provide an implementation of the dictionary

[1]This assumption has been also employed in [31] and in [20].

by using limited stable storage with logging operations.

A weaker notion of set, namely *weak set*, has been introduced in [7]. A weak set is an object representing a restricted form of set that does not include remove operations. The authors show how it is possible to implement a weak set in a system with no churn, by using a *finite number of atomic registers* assuming either (i) the number of processes is finite and known or (ii) the domain of the weak set is bounded and known by processes. Unless using an unbounded number of registers, these solutions do not work in the setting considered by this paper because the domain of a set is finite but arbitrary and the continuous churn can change the set of processes participating in the computation during the whole execution.

Other Shared Data Objects. Using shared data objects is a popular method to allow inter-process communication and coordination tasks. A tuple space [9] is a shared memory object, mainly used for coordination purpose, where generic homogeneous data structures, called *tuples*, are stored and retrieved. A tuple space is defined by three operations: $\mathsf{out}(T)$ which outputs the entry T in the tuple space (i.e., write), $\mathsf{in}(\bar{t})$ which removes the tuple that matches with $\bar{t}$ from the tuple space (i.e., destructive read) and $\mathsf{rd}(\bar{t})$ that is similar to the previous one but without removing the tuple. Tuples selection is done through an associative matching with templates: two tuples match if they have the same length and every corresponding pair of elements has the same type or the same value. Thus, templates can be seen as filters that select the desired tuples. Several tuple space implementations exist, both centralized (e.g., [22]) and distributed (e.g., [24], [19]).
The set object, as specified in this paper, differs from a tuple space because it is independent from the semantic of its domain elements. Even though $\mathsf{add}(v)$ and $\mathsf{remove}(v)$ can be seen as a particular case of $\mathsf{in}(\bar{t})$ and $\mathsf{out}(t)$ (i.e., where the tuple is composed only from one element), the $\mathsf{get}()$ differs from the $\mathsf{rd}(\bar{t})$ operation because it does not require to match any template but it just returns the current content of the object. Thus, it is suitable for keeping information stored with heterogeneous formats.

Other examples of shared objects are represented by the definition of abstract data types, emulated on top of message passing systems, with specific consistency requirements (i.e., linearizability, serializabitity, eventual consistency, etc.). Kosa [15] studied the impact of different consistency models (i.e., sequential consistency, linearizability and hybrid consistency) on operation worst-case response times for arbitrary abstract data types, providing lower bounds for a static failure-free distributed system. More recently, Roh et al. [25] considered replicated abstract data types supporting eventual consistency as basic building blocks for highly responsive collaborative applications. A set object, as presented in this paper, is a particular replicated data type with strong consistency requirements, emulated on top of a message passing system. To the best of our knowledge, this paper is the first one considering the emulation of a set object with strong consistency in dynamic settings.

3. System Model

Dynamic Distributed System. In a dynamic distributed system, processes may join and leave the system at their will. In order to model processes continuously arriving to and departing from the system, we assume the infinite arrival model (as defined in [23]). The set of processes that

can participate in the distributed system, i.e. the distributed system *population*, is composed by an unknown (but finite) number of processes $\Pi = \{\dots p_i, p_j, p_k \dots\}$, each one having a unique identifier (i.e. its index). The distributed system is composed, at each time, by a subset of the population. A process enters the distributed system by executing the connect() procedure. Such operation aims at connecting the new process to the processes that already belong to the system. A process leaves the system by means of a disconnect() operation. In the following we assume the existence of a *Connectivity and Communication Service* (CCS), i.e. a protocol managing the arrival and the departures of processes from the distributed system; such protocol is also responsible for the connectivity maintenance among processes part of the distributed system. Some examples of topologies and protocols keeping the system connected in a dynamic environment are [17], [16], [29], [32].

The passage of time is measured by a fictional global clock, represented by integer values, not accessible by processes. We will use the term *time unit* to denote the smallest measurable discrete unit of time. In the following, we will denote with t_i the i-th time unit from the start time of the system and with $t + x$ the x-th time unit after t.

Processes belonging to the distributed system communicate by exchanging messages through either point-to-point reliable channels or broadcast primitives. We assume that both the communication primitives do not create, drop or modifies messages. We will refine the specification of such communication primitives later in Section 6.1 and Section 6.2 where we will consider different timing assumptions.

Distributed Computation. Processes belonging to the distributed system may decide autonomously to join a distributed computation running on top of the system (e.g. the set computation). Hence, a distributed computation is composed, at each instant of time, by a subset of processes of the distributed system. A process p_i, belonging to the distributed system, that wants to join the distributed computation has to execute the join() operation. Such operation, invoked at some time t, is not instantaneous and it takes time to be executed; how much this time is, depends from the specific implementation provided for the join() operation. However, from time t, the process p_i can receive and process messages sent by any other process that participate in the computation. When a process p_j, participating in the distributed computation, wishes to leave, it executes the leave() operation. Without loss of generality, we assume that if a process leaves the computation and later wishes to re-join, it executes again the join() operation. Figure 1 and Figure 2 show, respectively, the system architecture and the distributed system and distributed computation layers.

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

Definition 1. *A process is* active *in the distributed computation from the time it returns from the* join() *operation until the time it starts executing the* leave() *operation.* $A(t)$ *denotes the set of processes that are active at time* t*, while* $A([t_1, t_2])$ *denotes the set of processes that are active during the whole interval* $[t_1, t_2]$ *(i.e.* $p_i \in A([t_1, t_2])$ *iff* $p_i \in A(t)$ *for each* $t \in [t_1, t_2]$*).*

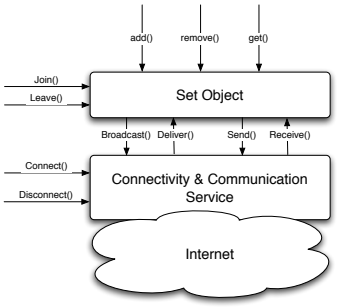


Figure 1: System Architecture

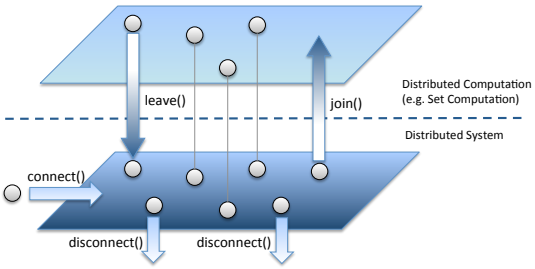


Figure 2: Distributed System *vs* Distributed Computation

Failure Model. Processes of the distributed system may fail by crashing (i.e., by stoping executing any action) and a failure can happen at any time. Let us remark that both the disconnect() operation and the leave() operation are implemented as passive leaves, thus they are equivalent to a crash failure. Thus, in the following we do not distinguish among failures and voluntarily departures.

Churn Model. In this paper we are interested in how a distributed computation is affected by the churn. We assume that (i) there always exist at least N processes in the distributed system and (ii) the CCS ensure that the connectivity is always preserved at the distributed system layer i.e., we assume that processes in the distributed computation are always able to communicate each other and no partitions occur at the distributed system layer. Concerning the computation churn, we consider at time t_0 (when the distributed computation starts) $n \leq N$ processes participating in the distributed computation and all of them are active (i.e., $|A(t_0)| = n$). The computation churn is continuous, i.e., at each time unit, $c \times n$ processes invoke the join() operation and $c \times n$ processes leave the computation, where $c \in [0, 1]$ is a fraction identifying the computation *churn rate*. As a consequence, the computation population size remains always constant. However, since joining processes are not active, the size of $A(t)$ changes with time due to the differences among the time intervals needed by different joining processes to complete their join() operation. Finally, the churn model does not guarantee that a given process, once joined, will remain part of the computation forever (absence of stable processes).

4. Set Object

A set object $\mathcal{S}$ is a replicated shared object used to store a collection of elements. It can be accessed by each process participating in the distributed computation without any restriction about the type of access i.e., each process is allowed to get elements from the object and to modify its content. We assume that processes can access the set object continuously i.e., each process may access the object infinitely often and the number of operations issued on the set is unbounded. $\mathcal{S}$ contains only elements taken from a finite but unknown domain $\mathcal{D}$ (e.g., a subset of the natural numbers) and it is initially empty. Informally, a set $\mathcal{S}$ can be accessed through three operations:

- The *add* operation, denoted add(v), takes an input parameter v representing an element of $\mathcal{D}$ and returns an add_return confirmation when the operation ends. It adds v to $\mathcal{S}$. If v is

already in the set, the add() operation has no effect.

- The *remove* operation, denoted remove(v), takes an input parameter v representing an element of $\mathcal{D}$ and returns a remove_return confirmation when the operation ends. If v belongs to $\mathcal{S}$, it suppresses it from $\mathcal{S}$. Otherwise it has no effect.

- The *get* operation, denoted get(), takes no input parameter. It returns a set containing the current content of $\mathcal{S}$ (i.e., all the elements added and not removed). It does not modify the content of the object.

Generally, each of these operations is not instantaneous and takes time to be executed; how much this time is depends on the specific implementation of each operation.

Protocol Model. A protocol $\mathcal{P}_{set}$ implementing a set object is a collection of distributed algorithms, one for each operation (i.e., $\mathcal{P}_{set}$= $\{\mathcal{A}_J, \mathcal{A}_A, \mathcal{A}_R, \mathcal{A}_G\}$ where $\mathcal{A}_J$, $\mathcal{A}_A$, $\mathcal{A}_R$ and $\mathcal{A}_G$ are respectively the distributed algorithm implementing the join(), the add(), the remove() and the get() operations). Each algorithm is composed of a sequence of computation and communication steps. A computation step is represented by the computation executed locally to each process while a communication step is represented by the sending and the delivering events of a message.

At any time t, a set object is maintained by the set of processes belonging to the distributed computation. No agreement abstraction is assumed to be available at each process (i.e., processes are not able to use consensus or total order primitives to agree upon the current elements stored in the set or to agree upon the set of processes participating in the distributed computation). Moreover, we assume that each process has the same role in the distributed computation (i.e., there is no special server acting as a coordinator)[2].

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). However, given two operations executed by two different processes, they may overlap.

In the remain of the section, we will provide the specification of the set object. To this hand, we will first provide in Section 4.1 some basic definitions that will help us in formalizing the set object specification (Section 4.2). Finally, once we specified the expected outcome of a get() operation, we will formalize our consistency criteria, namely *Per-element Sequential Consistency*, that relates the outputs of any get() operation to the whole execution (Section 4.3).

4.1. Basic Definitions

Every operation op is characterized by an $invocation$ event and a $reply$ event occurring at times $t_B(op)$ and $t_E(op)$ respectively. These events occur at two time instants (invocation time and reply time) according to the fictional global time. An operation op is *complete* if both the invocation

[2]Quorum-based protocols (e,g., [21]) are captured by this model because $\mathcal{P}_{set}$ does not require that all the processes execute each operation while $\mathcal{P}_{set}$ is not a state machine replication protocol (e.g., [27]) due to the lack of agreement abstraction.

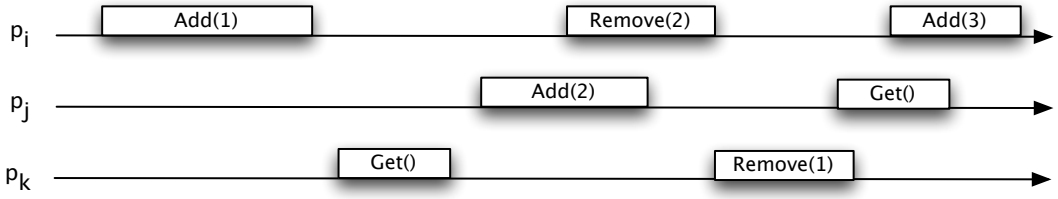


Figure 3: Execution History $\widehat{H}$ of a set object

event and the reply event occur (i.e. the process executing the operation does not crash between the invocation and the reply). Contrary, an operation op is said to be *failed* if it is invoked by a process that crashes before the reply event occurs. We use these time instants to model concurrency among operations through a precedence relation $\prec$.

Definition 2. *Given two operations op_1 and op_2, op_1* precedes *op_2 ($op_1 \prec op_2$) if and only if $t_E(op_1) < t_B(op_2)$. If op_1 does not precede op_2 and op_2 does not precede op_1 then they are* concurrent *($op_1||op_2$).*

Given a $\mathsf{add}(v)$ (respectively $\mathsf{remove}(v)$) operation, the element v is said to be added (respectively removed) when the operation is complete. As a consequence, failed $\mathsf{add}(v)$ (respectively $\mathsf{remove}(v)$) operations are incomplete operations. As in the case of atomic register [11], we consider that if a process crashes during a $\mathsf{add}(v)$ (respectively $\mathsf{remove}(v)$) operation, such operation is concurrent with all the successive operations and a fictional $reply$ event is added at the end of the computation for any failed $\mathsf{add}(v)$ and $\mathsf{remove}(v)$operation.

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

Let us remark that an execution history is a partial order. As a consequence, there may exist different possible ways to define an operation sequence (also called *serialization*) containing operations of the execution according to the partial order. To formalize this point, we introduce the notions of *consistent permutation* and *permutation set*.

Definition 4 (Permutation π Consistent with $\widehat{H}$)**.** *Given an execution history $\widehat{H} = (H, \prec)$, a* permutation *π of all the operations belonging to H is* consistent *with $\widehat{H}$ if, for any pair of operations op_1, op_2 in π such that op_1 precedes op_2 in $\widehat{H}$ then op_1 precedes op_2 in π.*

Considering the execution history $\widehat{H}$ shown in Figure 3, a permutation consistent with $\widehat{H}$ is $\pi = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{add}(2)j, \mathsf{remove}(2)i, \mathsf{remove}(1)k, \mathsf{get}()j, \mathsf{add}(3)i)$[3].

[3]Notation $op()id$ represents the operation op issued by the process with identifier id

Definition 5 (Permutation set)**.** *Let $\widehat{H} = (H, \prec)$ be the execution history of the set object. A* permutation set *of an execution history $\widehat{H}$, denoted $\Pi_{\widehat{H}}$, is the set of all the permutations π that are consistent with $\widehat{H}$.*

Considering again the execution history $\widehat{H}$ shown in Figure 3, the permutation set $\Pi_{\widehat{H}}$ of the execution history $\widehat{H}$ is shown in Figure 4.

$$
\begin{aligned}
\Pi_{\widehat{H}} = \{ \quad & \pi_1 = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{add}(2)j, \mathsf{remove}(2)i, \mathsf{remove}(1)k, \mathsf{get}()j, \mathsf{add}(3)i), \\
& \pi_2 = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{add}(2)j, \mathsf{remove}(2)i, \mathsf{remove}(1)k, \mathsf{add}(3)i, \mathsf{get}()j), \\
& \pi_3 = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{add}(2)j, \mathsf{remove}(2)i, \mathsf{get}()j, \mathsf{remove}(1)k, \mathsf{add}(3)i), \\
& \pi_4 = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{add}(2)j, \mathsf{remove}(2)i, \mathsf{get}()j, \mathsf{add}(3)i, \mathsf{remove}(1)k), \\
& \pi_5 = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{add}(2)j, \mathsf{remove}(2)i, \mathsf{add}(3)i, \mathsf{get}()j, \mathsf{remove}(1)k), \\
& \pi_6 = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{add}(2)j, \mathsf{remove}(2)i, \mathsf{add}(3)i, \mathsf{remove}(1)k, \mathsf{get}()j), \\
& \pi_7 = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{remove}(2)j, \mathsf{add}(2)i, \mathsf{get}()j, \mathsf{add}(3)i, \mathsf{remove}(1)k), \\
& \pi_8 = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{remove}(2)j, \mathsf{add}(2)i, \mathsf{get}()j, \mathsf{remove}(1)k, \mathsf{add}(3)i), \\
& \pi_9 = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{remove}(2)j, \mathsf{add}(2)i, \mathsf{add}(3)i, \mathsf{get}()j, \mathsf{remove}(1)k), \\
& \pi_{10} = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{remove}(2)j, \mathsf{add}(2)i, \mathsf{add}(3)i, \mathsf{remove}(1)k, \mathsf{get}()j), \\
& \pi_{11} = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{remove}(2)j, \mathsf{add}(2)i, \mathsf{remove}(1)k, \mathsf{add}(3)i, \mathsf{get}()j), \\
& \pi_{12} = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{remove}(2)j, \mathsf{add}(2)i, \mathsf{remove}(1)k, \mathsf{get}()j, \mathsf{add}(3)i) \\
\}.
\end{aligned}
$$

Figure 4: Permutation Set of the Execution History $\widehat{H}$

In the following, the notation $op \rightarrow_{\pi} op'$ is used to represent the precedence relation in the permutation π (i.e., the operation op precedes the operation op' in the sequence given by the permutation π). Let us remark that if $op \rightarrow_{\pi_i} op'$ for some $\pi_i \in \Pi_{\widehat{H}}$, it does not imply that $op \prec op'$ in $\widehat{H}$. On the contrary, if $op \rightarrow_{\pi_i} op' \; \forall \; \pi_i \in \Pi_{\widehat{H}}$ then $op \prec op'$ in $\widehat{H}$.

4.2. Set Object Specification

Let us recall that a $\mathsf{get}()$ operation does not modify the content of the set. Therefore, in order to define which are the elements that can be returned by each $\mathsf{get}()$ according with the operation executed on the set, let us introduce the concept of *sub-history induced by a* $\mathsf{get}()$ *operation*.

Definition 6 (Sub-history induced by a $\mathsf{get}()$ operation)**.** *Let $\widehat{H} = (H, \prec)$ be the execution history of all the* $\mathsf{add}(v)$*,* $\mathsf{remove}(v)$*, and* $\mathsf{get}()$*, operations invoked on the shared object. The sub-history of $\widehat{H}$ induced by a* $\mathsf{get}()$*, operation op, denoted $\widehat{U}_{\widehat{H},op} = (U, \prec)$, is defined as follows:*

- *$U \subseteq H$;*
- *$U = \{o \in H | (o = \mathsf{add}(v) \; \vee \; o = \mathsf{remove}(v)) \wedge t_B(o) < t_E(op)\} \cup \{op\}$;*
- *for each pair of operations o, o' such that $o, o' \in U$, if o precedes o' in $\widehat{H}$ then o precedes o' in $\widehat{U}_{\widehat{H},op}$*

As an example, Figure 5 shows the sub-history induced by the $\mathsf{get}()$ operation op issued by p_j on the history of Figure 3. For simplicity whenever it is clear from the context, $\widehat{U}_{\widehat{H},op}$ is referred to as $\widehat{U}$.

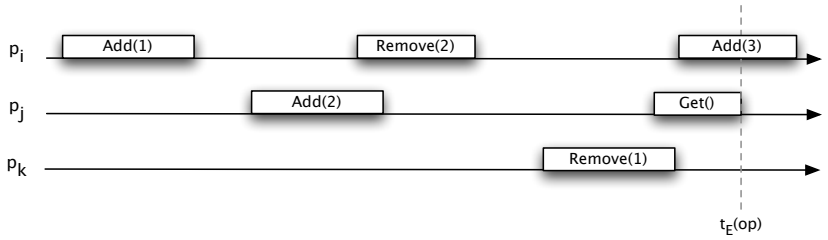


Figure 5: Update sub-history $\widehat{U}_{\widehat{H},op}$. of a set object

Definition 7 (Set Generated by a Permutation)**.** *Let $\widehat{H} = (H, \prec)$ be an execution history, and op be a* get() *operation of H. Given the sub-history $\widehat{U}_{\widehat{H},op} = (U, \prec)$ induced by op on $\widehat{H}$, let π be a permutation consistent with $\widehat{U}_{\widehat{H},op}$, then the set of elements V_π generated by π for op contains all the elements v such that:*

1. $\exists$ add$(v) \in \pi$: add$(v) \rightarrow_\pi op$ *and*
2. $\nexists$ remove$(v) \in \pi$: add$(v) \rightarrow_\pi$ remove$(v) \rightarrow_\pi op$

As an example, consider the execution history $\widehat{H}$ shown in Figure 3 and its sub-history $\widehat{U}$ induced by the operation op =get()j (Figure 5). Given a permutation $\pi_1 =$ (add(1)i, add(2)j, remove(2)i, remove(1)k, get()j) (with $\pi_1 \in \Pi_{\widehat{U}}$ consistent with $\widehat{U}$), the set of elements V_{π_1} generated by π_1 is $V_{\pi_1} = \emptyset$.

We are now in the position to define when a set V is an admissible set for a given get() operation op.

Definition 8 (Admissible set for a get() operation)**.** *Given an execution history $\widehat{H} = (H, \prec)$ of a set object $\mathcal{S}$, let op be a* get() *operation of H. Let $\widehat{U}_{\widehat{H},op}$ be the sub-history induced by op on $\widehat{H}$. An admissible set for op, denoted $V_{ad}(op)$, is any set generated by any permutation π belonging to the permutation set of $\Pi_{\widehat{U}}$.*

Let us note that, given a get() operation, if more than one consistent permutation exist (i.e. if the permutation set contains more than one permutation), then the get() has more than one admissible set. As an example, consider again the execution history $\widehat{H}$ of Figure 3 and consider the get() operation op issued by p_j. Given the permutation set $\Pi_{\widehat{U}}$, all the possible admissible sets for op are: $V_{\pi_1} = \emptyset$, $V_{\pi_2} = V_{\pi_6} = \{3\}$ $V_{\pi_3} = V_{\pi_4} = \{1\}$ $V_{\pi_5} = \{1, 3\}$ $V_{\pi_7} = V_{\pi_8} = \{1, 2\}$ $V_{\pi_9} = \{1, 2, 3\}$ $V_{\pi_{10}} = V_{\pi_{11}} = \{2, 3\}$ $V_{\pi_{12}} = \{2\}$.

Definition 9 (Set Object Specification)**.** *A protocol $\mathcal{P}_{set}$ implements a set object in the system model introduced in Section 3, if the following conditions holds:*

- Termination*: if a process p_i invokes an operation op on the set object and does not leave the computation , it eventually returns from op.*
- Get Validity*: any* get() *operation op invoked on the set object returns an admissible set for op.*

Let us note that the set object specification specifies only when a set is admissible (or valid) for a given execution history but it does not say anything about the result of two get() operations computed on the same execution history. As an example, let us consider the following scenario: there exists an add(v) concurrent with a remove(v) and two get() operations op_1 and op_2 executed after the end of both the add(v) and remove(v). This simple execution generates the following consistent permutations $\pi_1 = (\mathsf{add}(v), \mathsf{remove}(v))$ and $\pi_2 = (\mathsf{remove}(v), \mathsf{add}(v))$ generating respectively the following admissible sets $V_{\pi_1} = \emptyset$ and $V_{\pi_2} = \{v\}$. According to the specification of the set, we may have that op_1 and op_2 return respectively V_{π_1} and V_{π_2} providing two sets that are admissible (as they are the result of a consistent permutation) but seem to be not consistent. In the next Section we will specify which is the consistency condition for the set object that we require to our protocol $\mathcal{P}_{set}$.

4.3. Per-element Sequential Consistency Condition

A consistency condition defines which is the set among the admissible ones that a get() operation is allowed to return given the execution history. In a shared memory context, a set of formal consistency conditions has been defined [26] 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 concepts of *legal get* and *linear extension of an history*.

Definition 10 (Legal get()). *Let V be set returned by a* get()*operation op. This operation is legal if V is an admissible set for op.*

Definition 11 (Linear extension of an history). *A linear extension $\widehat{S} = (S, \rightarrow_s)$ of an execution history $\widehat{H}$ is a topological sort of its partial order where (i) $S = H$, (ii) $op_1 \prec op_2 \Rightarrow op_1 \rightarrow_s op_2$ and (iii) $\rightarrow_s$ is a total order.*

Let us introduce now the notion of *per-element 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 admissible set. Moreover, due to the semantic of the set when considering *concurrent operations involving different elements* (e.g., add(v) and add(v')), these operations can be perceived in different order by different processes. Formally,

Definition 12 (Per-element sequential consistency). *A history $\widehat{H} = (H, \prec)$ is* per-element sequentially consistent *iff for each process p_i there exists a linear extension $\widehat{S}_i = (S, \rightarrow_{s_i})$ such that:*

- *every* get() *operation is legal and*
- *for any pair of concurrent operations op and op', where op is an* add(v)*, op' is a* remove(v') *and $v = v'$, if $op \rightarrow_{s_i} op'$ for some p_i then $op \rightarrow_{s_j} op'$ for any other process p_j.*

As an example, consider the execution history $\widehat{H}$ shown in Figure 3. Given the three processes p_i, p_j and p_k the linear extensions such that $\widehat{H}$ is per-element sequentially consistent are the following:

- $\widehat{S}_i$ = add(1)i, add(2)j, remove(2)i, remove(1)k, add(3)i, get()j

- $\widehat{S}_j$ = $\mathsf{add}(1)i$, $\mathsf{add}(2)j$, $\mathsf{remove}(2)i$, $\mathsf{remove}(1)k$, $\mathsf{add}(3)i$, $\mathsf{get}()j$

- $\widehat{S}_k$ = $\mathsf{add}(1)i$, $\mathsf{add}(2)j$, $\mathsf{remove}(2)i$, $\mathsf{add}(3)i$, $\mathsf{remove}(1)k$, $\mathsf{get}()j$.

Relations among consistency criteria. Per-element sequential consistency is weaker than linearizability and sequential consistency but it is stronger than causal consistency.

Given an execution history, it is *sequentially consistent* if it admits a linear extension in which all the $\mathsf{get}()$ operations return an admissible set. In particular, if the provided linear extension is ordered according 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. In contrast, per-element sequential consistency still allows to have a partial order of the operations, with the constraint that only operations occurring on the same element have to be totally ordered. An example of linear extensions that are per-element sequential consistent but not sequential consistent is the one discussed in the previous section as $\widehat{S}_k$ follows a different order with respect to $\widehat{S}_i$ and $\widehat{S}_j$.

On the contrary, causal consistency [3] requires only that each $\mathsf{get}()$operation returns an admissible set with respect to some linear extension. As an example, considering the execution history $\widehat{H}$ shown in Figure 3, the following linear extensions are causally consistent but not per-element sequential consistent.

- $\widehat{S}_i$ = $\mathsf{add}(1)i$, $\mathsf{remove}(2)i$, $\mathsf{add}(2)j$, $\mathsf{remove}(1)k$, $\mathsf{add}(3)i$, $\mathsf{get}()j$

- $\widehat{S}_j$ = $\mathsf{add}(1)i$, $\mathsf{add}(2)j$, $\mathsf{remove}(2)i$, $\mathsf{remove}(1)k$, $\mathsf{add}(3)i$, $\mathsf{get}()j$

- $\widehat{S}_k$ = $\mathsf{add}(1)i$, $\mathsf{remove}(2)i$, $\mathsf{add}(2)j$, $\mathsf{remove}(1)k$, $\mathsf{add}(3)i$, $\mathsf{get}()j$.

Figure 6 shows the relationships among the four consistency criteria.

Note that, if the set domain $\mathcal{D}$ is composed of a single element (i.e., $|\mathcal{D}| = 1$), then per-element sequential consistency is equivalent to sequential consistency. In fact, in this case, any pair of concurrent operations take as input parameter the same element and need to be ordered in the same way by any process. In addition, by definition, also non-concurrent operations follow a total order. 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 element (e.g., its identifier). In this case, per-element sequential consistency boils down to causal consistency. Since each element is associated to one process and each process executes operations sequentially, it follows that concurrent operations take as input different elements and each process can perceive them in a different order, exactly as in causal consistency.

5. An Impossibility Result in Eventually Synchronous Dynamic System

In this Section, we will show that there not exists any protocol $\mathcal{P}_{set}$ implementing a set object using finite memory in a non-synchronous distributed system prone to continuous churn. The intuition behind the impossibility lies in the fact that eventual synchrony requires processes to coordinate among them during each operation. As a consequence, all the protocols implementing the

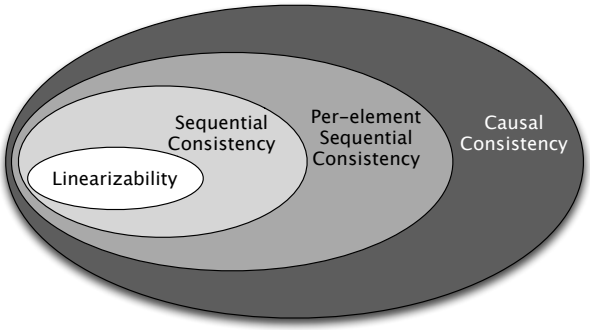


Figure 6: Consistency Criteria Relations

set object require to the process issuing the operation to wait for the reception of some messages. In the meanwhile, due to the continuous churn, processes participating in the distributed computation might changes. Therefore, to ensure the operations' termination, processes need to store ongoing operations locally and to transfer this information to newcomers in order that the latter can help the operation termination (cfr. Lemma 4). The combined effect of both eventual synchrony and churn makes impossible for a process to decide when it can discard stored information (Lemma 5). Thus, considering that each process can invoke an operation and the number of processes is potentially infinite, we get to the need of infinite memory at each process (Theorem 1).

Before going in to the details of the proof, let us introduce some preliminary definitions.

Definition 13 (Process state at time t)**.** *We define as* process state *of a process* p_i at time t *the set of all its local variables together with their values at time* t*.*

Definition 14 (Configuration at time t)**.** *A* Configuration *at time* t*, denoted as* $C(t)$*, is a set of pair* $< p_i, st_i >$ *where* p_i *is a process participating in the computation at time* t *and* st_i *is the state of* p_i *at time* t*.*

As a first step, let us prove that eventual synchrony imposes some constraints on the message patterns occurring in the protocol (cfr. Lemma 1 - Lemma 3):

Lemma 1. *Let* $\mathcal{P}_{set} = \{\mathcal{A}_J, \mathcal{A}_A, \mathcal{A}_R, \mathcal{A}_G\}$ *be a protocol implementing a set object. Any algorithm* $\mathcal{A}_A$ *(*$\mathcal{A}_R$ *respectively) must involve at least one* send $-$ reply *communication pattern (i.e., two communication steps).*

Proof Let us consider the initial configuration $C(t_0)$ holding at time t_0 when the set computation starts. Let $op = \mathsf{add}(v)$ be the first update operation terminated at some time $t > t_0$ and let $C(t)$ be the configuration holding at time t. Let as assume that the algorithms $\mathcal{A}_J$ and $\mathcal{A}_G$ are implemented by two perfect instantaneous oracles that, when executed, returns the union of all the active process states in the computation.

Let us assume by contradiction that one communication step is enough to provide a correct implementation of the set object.

Let us divide processes of the distributed computation in two groups: (i) the process p_i that issued the operation op and (ii) all the other processes. At time t_0, all the processes (including p_i) start from the initial configuration where all the local states are equals and the set object is empty.

At time t an update operation is terminated and the set object contains the element v. Thus, any following get() operation must return the set $\{v\}$ (until a remove(v) is invoked).

If a get() operation must return the set $\{v\}$ it means that at some point during the update operation (i.e., between time t_0 and t) the state of at least an active process must change storing some information about the addition of v to the set. Let us note that updating the state of p_i is not sufficient as the system is dynamic and p_i could leave just after the operation is terminated. As a consequence, p_i must necessarily send a message to notify that an update operation is running and to trigger the state update in at least one other process i.e., the send communication step is necessary.

Let us now show that the send communication step is not sufficient to provide a correct implementation of $\mathcal{A}_A$.

In oder to be correct, $\mathcal{A}_A$ must ensure the *termination* property. As a consequence, p_i must be able to decide when it can trigger the add_return event. In particular, this can be done when at least one process[4] updated its state.

Let us recall that (i) processes communicate only by exchanging messages, (ii) they do not know the membership of the computation and (iii) the system is eventually synchronous. As a consequence, the only way p_i has to know that at least one other process p_j updated its state is to wait for an acknowledgement from p_j. As a consequence, a second communication step, i.e., a reply step, is necessary for a correct implementation of $\mathcal{A}_A$

The same arguments applies for a correct implementation of $\mathcal{A}_R$ and the claim follows.

$\square_{Lemma\ 1}$

Lemma 2. *Let $\mathcal{P}_{set} = \{\mathcal{A}_J, \mathcal{A}_A, \mathcal{A}_R, \mathcal{A}_G\}$ be a protocol implementing a set object. Any algorithm $\mathcal{A}_G$ ($\mathcal{A}_J$ respectively) must involve at least one* send − reply *communication pattern (i.e., two communication steps).*

Proof The claim simply follows by observing that even in a static eventually synchronous system "read" operations (i.e., operations that return the state of the shared object) cannot be executed locally. $\square_{Lemma\ 2}$

Lemma 3. *Let $\mathcal{P}_{set} = \{\mathcal{A}_J, \mathcal{A}_A, \mathcal{A}_R, \mathcal{A}_G\}$ be a protocol implementing a set object. Let Q_A, Q_R, Q_J and Q_G be the number of (reply) messages that an* add(), *a* remove(), *a* join() *and a* get() *operation respectively must wait before terminating the corresponding algorithm execution. $Q_x > \frac{n}{2}$ (with $x \in \{A, R, J, G\}$).*

Proof From Lemma 1 and Lemma 2 it follows that any algorithm $\mathcal{A}_x$ (with $x \in \{A, R, J, G\}$) requires to the running process p_i to wait some message from the others before terminating. As a

[4]The exact number of processes is given by the implementation of the $\mathcal{A}_J$ and $\mathcal{A}_G$ algorithms. In any case, such number does not affect the proof and it must be at least one.

consequence $|Q_x| \geq 1$. Let us now observe that a process p_i issuing a get() or a join() operation has no knowledge about the set of process participating in the distributed computation and that updated their states during a preceding update operation. As a consequence, in order to ensure the Get Validity property, p_i needs to be sure that in the set of replies it receives there is at least one process with an updated state and this is obtained with the intersection of two majorities from which the claim follows. $\square_{Lemma\ 3}$

Corollary 1. *Let $\mathcal{P}_{set} = \{\mathcal{A}_J, \mathcal{A}_A, \mathcal{A}_R, \mathcal{A}_G\}$ be a protocol implementing a set object. Any algorithm $\mathcal{A}_x$ (with $x \in \{A, R, J, G\}$) has a terminating condition including the following step:*

$$\textbf{wait until } |Q_x| < \frac{n}{2}.$$

Proof The corollary follows directly from Lemma 3. $\square_{Corollary\ 1}$

Corollary 2. *At any time t, $|A(t)| \geq Q_x > \frac{n}{2}$ (with $x \in \{A, R, J, G\}$).*

Proof The corollary follows from the observation that processes joining the computation (i.e., processes that are not yet active) have no knowledge about the computation and they have no state. As a consequence they are not able to answer to any message before they become active and to avoid them to block the computation they must be a minority. $\square_{Corollary\ 2}$

Informally speaking, previous Lemmas and Corollaries show that the timing assumption (i.e., eventual synchrony) plays an important role in the definition of a distributed protocol implementing a set object. In fact, the uncertainty about the time required from messages to reach their destination and the absence of knowledge about the computation membership require any algorithm, part of the set protocol, to follow a quorum-based approach relying on a majority of active processes.

Lemma 4. *Let $\mathcal{P}_{set} = \{\mathcal{A}_J, \mathcal{A}_A, \mathcal{A}_R, \mathcal{A}_G\}$ be a protocol implementing a set object. It is a necessary condition for the termination of any algorithm $\mathcal{A}_A$ ($\mathcal{A}_R$ respectively) that any process p_i stores information about the current* add() *(*remove() *respectively) operation.*

Proof Let us suppose by contradiction that there exists an algorithm $\mathcal{A}_A$ ($\mathcal{A}_R$ respectively) running at a process p_i that terminates and that never stores information related to the current add() (remove() respectively) operation.

Due to Corollary 1, if $\mathcal{A}_A$ ($\mathcal{A}_R$ respectively) terminates, it means that eventually the condition **wait until** $|Q_x| < \frac{n}{2}$ will become false. Let us recall that initially all the processes starts from the initial state where all the local variables are initialized to the default values. Thus, initially the variable counting the number of replies received Q_x is set to 0.

Let us consider the scenario depicted in Figure 7 with a distributed computation composed of 5 processes. Without loss of generality, let us assume that the add() operation triggered by p_1 is the first operation issued on the set. Due to the presence of churn, processes start to join and the leave the computation. As a consequence, the set of active processes may change along time.

Let $A(t_0) = \{p_1, p_2, p_3, p_4, p_5\}$ be the set of active processes at the time t_0 when p_1 disseminates information about the add(). At time $t' > t_0$, process p_5 leaves the computation and a new process

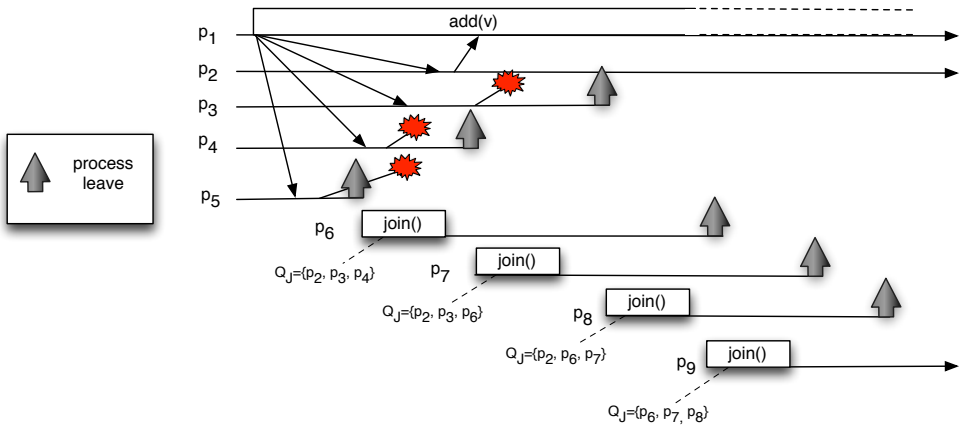


Figure 7: Non-terminating run for protocol $\mathcal{P}_{set}$.

p_6 join. Let us remark that, due to the assumptions on the communication primitives, (i) p_6 is not guaranteed to deliver messages related to the add() operation sent by p_1 and (ii) messages sent from p_5 to p_1 may be lost as p_5 left from the computation.

Note that, this scenario where new processes substituting those in $A(t_0)$ may happen several times as shown in Figure.

Thus, the only chance for the add() operation to terminate is that p_1 receives answers from processes that substituted those in $A(t_0)$. Note that a process $p_j \notin A(t_0)$ can become aware of the add() operation if (i) p_1 informs p_j about the operation, or (ii) some other active process p_k, that updated its state for the effect of the add() operation, forwards information about current running operations.

Let us consider the fist case. As shown in Figure, it may happen that a process $p_j \notin A(t_0)$ leaves before being informed from p_1 and thus it may be not able to answer to the message. Note that, due to the eventual synchrony of the communication primitives, this scenario may happen for any joining process. Considering that the churn is continuous, this may happen an infinite number of times letting the operation not terminating. Therefore, we have a contradiction.

Let us now consider the second case. In order to forward such information to p_j, a process $p_k \in A(t_0)$ needs to store such information locally. Considering that there is no special process in the computation but all the processes execute the same protocol, we have that any active process should store information related to update operations and we have a contradiction.

$\square_{Lemma\ 4}$

Informally, Lemma 4 shows that due to the effect of continuous churn, processes participating in the computation might continuously changes. As a consequence, the set of active processes, that can help the termination of the protocol, changes and the only way for a newcomer to become aware of an operation starting before its arrival is to be informed by someone. To this end, the protocol needs to keep track of the information related to each operation.

Lemma 5. *Let $\mathcal{P}_{set} = \{\mathcal{A}_J, \mathcal{A}_A, \mathcal{A}_R, \mathcal{A}_G\}$ be a protocol implementing a set object. Let p_i be a process running the $\mathcal{A}_A$ algorithm ($\mathcal{A}_R$ respectively) and storing information about the current* add()

*(*remove() *respectively) operation.* p_i *is not able to detect when it can safely delete information about the* add() *operation.*

Proof Due to Lemma 4, while running the algorithm $\mathcal{A}_A$ ($\mathcal{A}_R$ respectively), processes need to store information about the running operation in order to forward them to joining processes and allow the termination. As a consequence, a process p_i can remove information about an add() operation (remove() respectively) when it knows that the operation is completed. This information is generated by the process p_j issuing the operation as it is the only one that knows when the operation terminates. Thus, p_i may know that the operation is terminated if and only if p_j advertises the end of the operation and such a message is delivered by p_i.

However, due to the churn, p_j may disseminate such information and then leave the computation. As a consequence, considering that the communication primitives may loose messages sent from processes leaving the computation, we have that p_i may not receive any message and keep forever the information stored in memory. $\square_{Lemma\ 5}$

Informally, the previous Lemma shows that eventual synchrony does not make possible for a process p_i to understand when an operation is completed unless the issuing process p_j advertises this information. However, due to the churn effect, p_j may leave just after the end of the operation and its advertisement can be lost forcing p_i to keep forever the information stored locally.

Theorem 1. *Let* $\mathcal{P}_{set} = \{\mathcal{A}_J, \mathcal{A}_A, \mathcal{A}_R, \mathcal{A}_G\}$ *be a protocol implementing a set object. There not exists a protocol* $\mathcal{P}_{set}$ *implementing a per-element sequential consistent set object using finite memory.*

Proof The claim follows from Lemma 5 by considering that there may exist an infinite number of processes joining the computation, updating the object and then leaving. As a consequence, an infinite number of operation may be stored locally and never deleted. $\square_{Theorem\ 1}$

6. Implementing Set Objects with Finite Memory

As shown in the previous Section, the impossibility to design a distributed protocol $\mathcal{P}_{set}$ working with finite memory depends on the combined action of churn and eventual synchrony. To overcome this impossibility, it is possible to follow three different approaches:

1. **Stronger timing assumptions.** Assuming a synchronous system where the communication latency is bounded by a known constant δ allows us to simplify the structure of the protocol and to compute upper bounds on the time each operation requires to terminate. As a consequence, it is possible to define a procedure able to delete from the memory all the completed operations keeping the memory size finite and bounded.
2. **Weaker churn model.** A weaker churn model is the one of *quiescent churn* i.e., the one assuming the existence of a time t after which the churn stops long enough to ensure the convergence of the protocol. Under this assumption, it is possible to build a distributed protocol implementing the set object as Lemma 5 does not hold anymore. In fact, there will exists a time after which processes stop to leave (and to join) and thus messages advertising

the end of the operation will not be lost anymore. This makes possible to delete operations from the memory and to use a finite amount of memory. However, this implementation does not allows to bound the amount of memory required. In fact, the amount of memory depends from the number of operations invoked before the stability period and cannot be fixed a priori.

3. **Weaker object specification.** Weakening the specification allows to trade off what the object is able to guarantee and the amount of memory required.

In this paper, we focus our attention on points 1 and 3 as they allow to design a protocol working with bounded memory while point 2 just allows to guarantee that the memory is finite.
In Section 6.1, we will consider a synchronous system and we show that a set object implementation, using finite memory space for local computation, is possible.
In Section 6.2, we will consider an eventually synchronous system and we weaken the specification of the set object by introducing a k-bounded set object and we provide a protocol implementing such an object using finite memory in presence of continuous churn.

6.1. Set Object in a Synchronous System

The distributed system is synchronous in the following sense: processing times of local computations (except for the wait statement[5]) are negligible with respect to communication delays, so they are assumed to be equal to 0. However, messages take time to travel to their destination processes. In the following, we will detail the guarantees offered by the two communication primitives introduced in Section 3 while used in a synchronous system.

Point-to-point communication. The network allows a process p_i to send a message to another process p_j as soon as p_i knows that p_j entered the system. Let us recall that the network is reliable, i.e., it does not loose, create or modify messages. Moreover, the synchrony assumption guarantees that there exists a known upper bound δ' on message transmission delays. More formally, if p_i invokes "send MSG() to p_j" at time t, then p_j receives that message by time $t + \delta'$, if it has not left the system by that time.

Broadcast. It is assumed that the CCS offers a broadcast communication primitive that provides processes with two operations, denoted broadcast MSG() and deliver MSG(). 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 "broadcasted" and "delivered". These operations satisfy the following property:

- Timely delivery: Let t be the time at which a process p_i invokes broadcast MSG(). There is a constant δ ($\delta \geq \delta'$) (known by the processes) such that if p_i 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 the message by time $t + \delta$.

[5] A wait statement is a particular computation step that blocks the process and keeps it waiting for a certain amount of time before going to the next step in the code. While waiting, a process can still deliver messages and process them according to the protocol.

Such a pair of broadcast operations has first been formalized in [12] in the context of systems where process can commit crash failures. It has been extended to the context of dynamic systems in [8]. Let us note that, similarly to the case of point-to-point communication primitive, the constant δ represents an upper bound on the transmission delay of the broadcast. For the ease of presentation, in the following we will use only the constant δ as it is an upper bound for both the point-to-point transmission delay and the broadcast transmission delay.

6.1.1. Per-element Sequential Consistent Protocol

The basic idea behind the set object implementation is to replicate the set and other relevant information (e.g., recent executed operations, pending joins, etc.) over all the processes participating in the set computation. In the following, we will use to term *update operation* to identify an operation that modifies the content of the set object i.e., an update operation can be either an add() or a remove() operation. Moreover, we will call *state of the set* at process p_i the local copy of the set stored by p_i together with a sequence number counting how many update operations have been seen by p_i and the list of recent operations stored by p_i.

When a process wants to update the content of the set, it just asks to all the other processes to perform the operation on its local copy while when it invokes a get() operation it just returns the local copy of the set.

In order to implement a set object (as specified in Section 4.2) satisfying per-element sequential consistency in a distributed system with continuous churn we have to handle the following issues:

- update operations should be persistent despite churn: if an element v is added to the set then v should be in set (and should be returned by any get() operation) until it is removed and if an element v' is removed it should not appear anymore in the set (i.e., it should not be returned by any get() operation) until it is added again;
- concurrency among add() and remove() of the same element v must be handled in a consistent way by all the processes;
- the memory used for local computation should be finite.

The first issue is mainly due to the fact that active processes (maintaining the set) leave and new joining processes have no information about the current content of the set. Thus, if the protocol does not properly master the churn, the set may disappear as the processes having an updated state are not in the computation anymore. To solve this issue we designed a join() operation that basically implements a state transfer (i.e. elements in the set and other relevant information) from active processes to the joining one. In particular, when a process joins, it queries other processes in the distributed computation to obtain their current state. Exploiting the synchrony of the system, the joining process waits for a round trip delay and then updates its local state by processing the information received with states of active processes and then becomes active. In the meanwhile, if it receives requests by other joining processes or it receives messages requiring to update the set, it buffers those messages and will process them as soon as it becomes active.

To enforce per-element sequential consistency, we assign a timestamp to each update operation. In particular,

update operations are implemented by sending an update message to disseminate the element to be inserted/removed, together with the operation type, to all the replicas. When a process receives such an update message, it updates its local copy of the set according to Thomas' *Last-Writer-Wins* rule [30]: updates are applied to the local copy of the set if and only if the corresponding timestamp is greater than the local one (i.e. the sequence number). We have defined an update procedure executed each time that an add() or a remove() is issued. Such procedure is executed atomically and checks if the current operation is new or does not "conflict" with an already executed operation i.e., it checks the type of the operation and updates the local copy of the set it if and only if the timestamp of the current operation (composed by the pair sequence number and process identifier) follows the one stored locally with respect to the total order imposed by the pair sequence number and process identifier.

In order to ensure that the memory used is always finite, we exploit the synchrony of the communication to estimate the maximum time that each operation must be stored at each replicas before it can be garbage collected (cf. procedure garbageCollection()). In particular, an operation can be garbage collected when it has been executed by any active process and there are no more messages related to concurrent update operations traveling in the network (i.e., all the possible conflicts arising by concurrent add(v) and remove(v) operations are resolved). This allows to keep the buffer size limited and makes possible to provide an implementation working with finite memory space.

In the following, we will provide additional details on the implementation of each operation. We will first show the algorithms $\mathcal{A}_A$, $\mathcal{A}_R$ and $\mathcal{A}_G$ implementing respectively the the add(), the remove() and the get() operations and then we will detail the $\mathcal{A}_J$ algorithm implementing the join() operation.

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

- Two variables denoted set_i and sn_i; set_i contains the local copy of the set; sn_i is an integer that represents the sequence number for the current update operation and it is used to generate the timestamp of update operations.

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

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

- Three set variables, denoted $replies_i$, $reply\_to_i$ and $pending_i$, that are used in the period during which p_i joins the computation. $replies_i$ contains the 3-tuples $< set, sn, ops >$ that p_i has received from other processes during its join period, while $reply\_to_i$ contains the identifier of processes that are joining the comptation concurrently with p_i (as far as p_i knows). The set $pending_i$ contains the 4-tuples $< type, val, sn, id >$ each one characterizing an update operation running concurrently with the join.

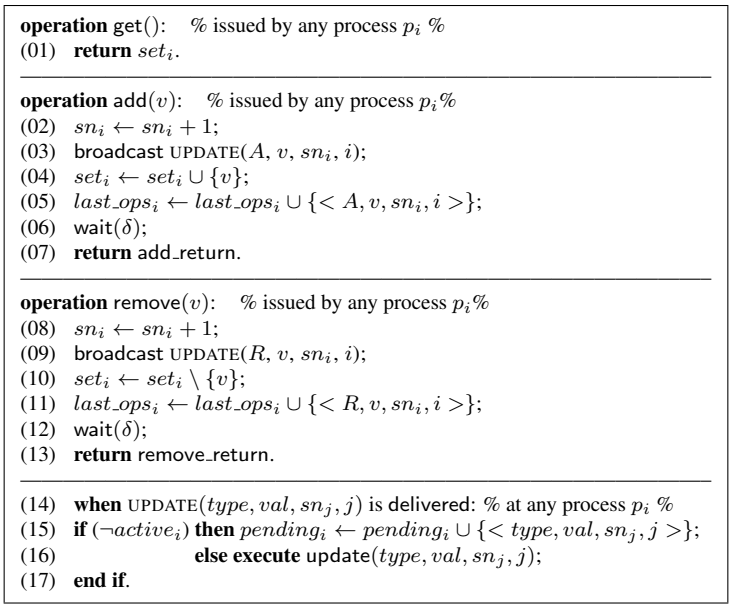


Figure 8: The get(), add() and remove() protocol for a synchronous system (code for p_i)

The get() **operation.** The algorithms $\mathcal{A}_G$ implementing the get() operation is shown in Figure 8. The get is purely local (i.e., fast): it consists in returning the current content of the local variable set_i.

The add(v) **and the** remove(v) **operations.** The algorithms $\mathcal{A}_A$ and $\mathcal{A}_R$ implementing the add(v) and the remove(v) operations are shown in Figure 8. They have a quite similar structure. In order to ensure per-element sequential consistency, update operations have to be executed at each process following the same order provided by a deterministic rule. In the proposed algorithm, the deterministic rule is defined as the total order among the pairs $< sn, id >$ where sn is the sequence number of the operation and id is the identifier of the process issuing the operation.

When p_i wants to add/remove an element v to/from the set, it increments its sequence number sn_i (line 02 and line 08 of Figure 8), it broadcasts an UPDATE($type$, val, sn, id) message (line 03 and line 09 of Figure 8) where $type$ is a flag that identify the type of the update (i.e., A for an add() operation or R for a remove() operation), val is the element 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. Then, p_i executes the operation on its local copy of the set (line 04 and line 10 of Figure 8) and it stores in its $last\_ops_i$ variable the tuple $< type, val, sn, id >$ that identifies such operation (line 05 and line 11 of Figure 8).

Finally, p_i waits for δ time units (line 06 and line 12 of Figure 8) to be sure that all the active processes have received the UPDATE message before returning from the operation (line 07 and line 13 of Figure 8).

When a process p_i receives an UPDATE($type, val, sn_j, j$) message from a process p_j, if it is not active, it puts the current UPDATE message in its $pending_i$ buffer and will process it as soon as it will be active, otherwise it executes the UPDATE() procedure shown in Figure 9.

This procedure is executed atomically by each process and is responsible of enforcing a total

```
procedure update(type, val, sn_j, j) % at any process p_i %
(01) if (sn_j > sn_i)
(02)   then last_ops_i ← last_ops_i ∪ {< type, val, sn_j, j >};
(03)        if (type = A) then set_i ← set_i ∪ {< val, sn_j >};
(04)                      else set_i ← set_i \ {< val, − >};
(05)        end if;
(06)   else
(07)        temp ← {X ∈ last_ops_i | X =< −, val, −, − >};
(08)        if (temp = ∅)
(09)          then last_ops_i ← last_ops_i ∪ {< type, val, sn_j, j >;
(10)               if (type = A) then set_i ← set_i ∪ {< val, sn_j >};
(11)                             else set_i ← set_i \ {< val, − >};
(12)               end if;
(13)          else if ((type = A) ∧ (∄ < R, −, sn, id >∈ temp | (sn, id) > (sn_j, j)))
(14)                 then set_i ← set_i ∪ {< val, sn_j >};
(15)                      last_ops_i ← last_ops_i ∪ {< type, val, sn_j, j >};
(16)               end if;
(17)               if ((type = R) ∧ (∄ < A, −, sn, id >∈ temp | (sn, id) > (sn_j, j)))
(18)                 then set_i ← set_i \ {< val, − >};
(19)                      last_ops_i ← last_ops_i ∪ {< type, val, sn_j, j >};
(20)               end if;
(21)        end if;
(22) end if;
(23) sn_i ← max(sn_i, sn_j).
```

Figure 9: The update() procedure for a synchronous system (code for p_i). The procedure is executed atomically.

order among operations needed to enforce per-element sequential consistency. As first action p_i checks if the sequence number sn_j, corresponding to the current operation, is greater than the one stored locally and if it is so, p_i executes the operation by adding/removing the element to the local copy of the set (lines 02-05).

If not, p_i checks if there exists some operation occurred on the same element val in the set of the last executed operation $last\_ops_i$; if there is not such an operation, p_i executes the current one (lines 06 - 13) otherwise, it checks, according to the type of the operation to be executed, if the two operations are in the right order and if so, p_i executes the operation (lines 14 - 22).

Finally p_i updates it sequence number (line 23) by selecting the highest between its own and the one associated to the operation.

The join() **operation.** The algorithm implementing the join() operation for a set object is shown in Figure 10, and involves all the processes that are currently in the distributed computation (whether active or not).

First p_i initializes its local variables (line 01) and waits for a period of δ time units (line 02); the motivations for such waiting period is explained later. After this waiting period, p_i broadcasts an INQUIRY() message to acquire states of other processes and remains waiting for 2δ time units, i.e., the maximum round trip delay (lines 03-04).

When a process p_j delivers an INQUIRY() message from p_i and it is active, it answers by sending back a REPLY() message containing its current state (composed by its local copy of the set, the current sequence number and the list of recent operation) (line 20), otherwise it buffers the request in the $reply\_to_j$ variable (line 21) and will answer as soon as it will be active (line 15).

When p_i is unblocked by the wait statement, it processes all the received information to deter-

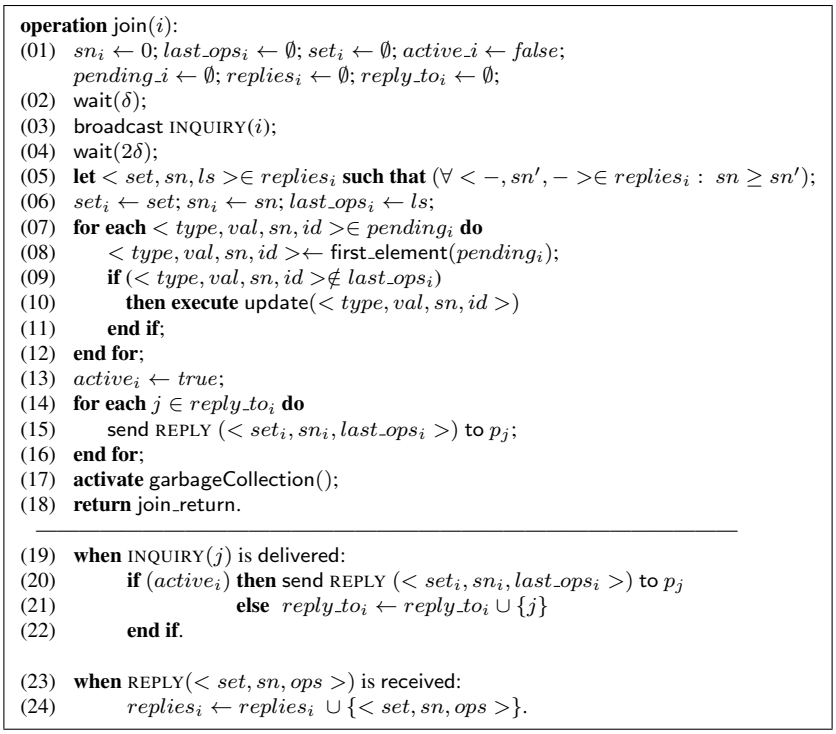
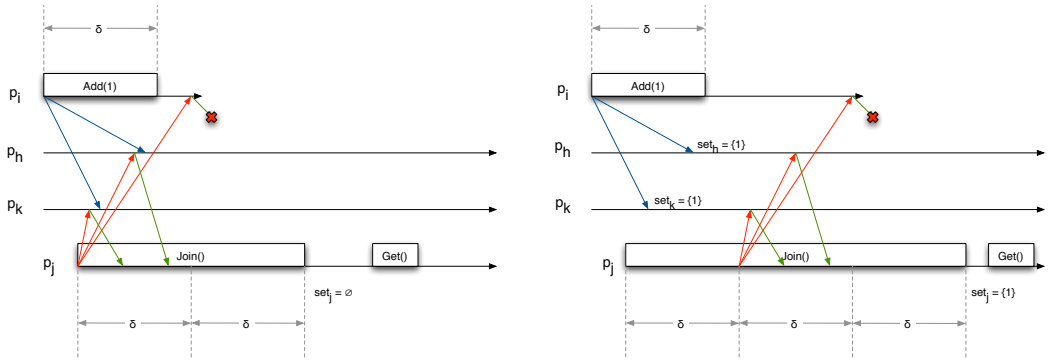


Figure 10: The $\mathcal{A}_J$ algorithm for the join() operation of a set object in a synchronous system (code for p_i).

mine the current content of the set. In particular, it updates its local variables set_i, sn_i and $last\_ops_i$ to the most up-to-date set it has received from active processes (lines 05-06). Moreover, for each UPDATE() message received during the join execution and stored in the $pending_i$ variable, p_i executes the update() procedure (shown in Figure 9 and discussed later), as if the UPDATE() message is just received (lines 09-11). Then, p_i becomes active (line 13), which means that it can answer the inquiries it has received from other processes, and does it if $reply\_to \neq \emptyset$ (line 14). Finally, p_i activates the garbageCollection() procedure (shown in Figure 12 and discussed later) to avoid that the $last\_ops_i$ variable grows to infinite (line 17) and then returns (with the join_return event) to indicate the end of the join() operation (line 18).

Why the wait(δ) statement at line 02 of the join() operation? To motivate the wait(δ) statement at line 02 of Figure 10, let us consider the execution depicted in Figure 11(a). The set is initially empty and the processes p_i, p_h and p_k are the three processes participating in the computation. At time t, p_i invokes an add() operation. Moreover, the process p_j invokes the join() operation just after t.

When p_i invokes the add() operation, it broadcasts an UPDATE() message and waits δ time unit before returning from the operation. Due to the timely delivery property of the broadcast primitive, p_h and p_k deliver UPDATE() message containing the element to be added (i.e., 1) by $t+\delta$. However, since p_j starts the join() operation just after time t, there is no such a guarantee that it will deliver the message as when the message has been generated, p_j was not participating in the computation. Hence, if p_j does not execute the wait(δ) statement at line 02, its execution of the lines 05-12



(a) Without wait(δ) (b) With wait(δ)

Figure 11: Why wait(δ) is required

can provide it with the previous value of the set, namely $\emptyset$, as shown in Figure 11(a) as, in the worst-case scenario, all the active processes may first answer to the INQUIRY() message and then deliver the UPDATE() message. Thus, p_j may become active storing a copy of the set that is not up-to-date. In addition, if p_j issues a get() operation just after the end of join(), it obtains again $\emptyset$, while it should obtain the new set $\{1\}$ (because 1 is the element added by a completed add() operation and there is no remove() concurrent with this get() issued by p_j).

The execution depicted in Figure 11(b) shows that this incorrect scenario cannot occur if p_j is forced to wait for δ time units before inquiring to obtain the last value of the set.

Garbage Collection. Let us remark that the $last\_ops_i$ set variable collects the information related to operations executed on the set. However, considering that messages are always delivered in a bounded time interval, the protocol needs only the information related to recent operations to checks the total order among timestamps and ensure per-element sequential consistency. To avoid this problem it is possible to define a GARBAGECOLLECTION() procedure that periodically removes from the $last\_ops_i$ variable the information related to "old" operation.
The thread managing the garbage collection is shown in Figure 12 and each iteration is executed atomically. It is activated the first time at the end of the join() operation (line 17, Figure 10) and then is always running. Every 2δ time units, it checks the operations not known in the previous period (line 05) and discards the ones already executed (line 06).

6.1.2. Correctness Proofs

In this section, the termination of each operation is first proved (Theorem 2). Then we show that every execution history generated by the protocol is per-element sequential consistent under the assumption that the churn rate is below a certain bound (Theorem 3). We prove the latter theorem through two main steps: (*i*) if every active process maintains at any time an admissible set, the execution history is always per-element sequential consistent (Lemma 10); (*ii*) if the churn is below a certain bound, every get()operation returns an admissible set (Lemma 9). To get the latter result, we need to prove preliminary lemmas stating that if the churn is below a certain bound, a process that issued a join() operation becomes active storing an admissible set, i.e., the current

```
procedure garbageCollection():
(01) temp_i ← ∅; old_ops_i ← ∅;
(02) while(true) every 2δ
(03)     temp_i ← ∅;
(04)     for each op =< type, val, sn, id >∈ last_ops_i do
(05)         if (op ∉ (last_ops_i ∩ old_ops_i)) then temp_i ← temp_i ∪ op;
(06)                                            else last_ops_i ← last_ops_i \ op;
(07)         end if;
(08)         old_ops_i ← temp_i;
(09)     end for;
(10) end while.
```

Figure 12: The garbageCollection() procedure for a set object in a synchronous system (code for p_i). The procedure is executed atomically.

state of the shared object is transferred from active processes already participating in the distributed computation to newcomers. In particular, Lemma 7 proves that, starting from a set of processes having an admissible set a time t_0 and given that the churn rate is below a certain threshold, then every process invoking a join() operation at time t_1 becomes active returning from the operation with an admissible set, while Lemma 8 generalizes this proof to any join() operation.

Theorem 2. *If a process invokes* join() *operation and does not leave the system for at least* 3δ *time units or invokes a* get() *operation or invokes an* add() *operation or a* remove() *operation and does not leave the computation for at least* δ *time units, then it terminates the invoked operation.*

Proof The get() operation trivially terminates. The termination of the join(), add() and remove() operations follows from the fact that the wait() statements at line 02 and line 04 of Figure 10 and at line 06 and line 12 of Figure8 terminates. $\square_{Theorem\ 2}$

In the following Lemma we will prove that if the churn rate is smaller than $\frac{1}{3\delta}$, then there always exists at least one active process in the distributed computation.

Lemma 6. *Let* c *be lesser than* $\frac{1}{3\delta}$*. At any instant of time* t *we have*

$$|A[t, t + 3\delta]| \geq n(1 - 3\delta c) > 0$$

Proof Let us recall that at time t_0 (when the computation starts) all the processes are active (i.e., $|A(t_0)| = n$). Due to assumption on the churn, at time t_1 $n \times c$ processes leave the system and $n \times c$ processes invoke the join() operation. Hence, $|A[t_0, t_0 + 1]| \geq n - (n \times c)$. During the second time unit, $n \times c$ new processes enter the system and replace the $n \times c$ processes that left the system during that time unit. In the worst case, the $n \times c$ processes that left the system are processes that were present at time t_0 (i.e., they are not processes that entered the system at time $t_0 + 1$). So, $|A[t_0, t_0 + 2]| \geq n - 2(n \times c)$. If we consider a period of 3δ time units, i.e., the longest period needed to terminate a join() operation, it follows that $|A[t_0, t_0 + 3\delta]| \geq n - 3\delta(n \times c) = n(1 - 3\delta c)$. Moreover, as $c < \frac{1}{3\delta}$, then $|A[t_0, t_0 + 3\delta]| \geq n(1 - 3\delta c) > 0$. As a consequence, at time $t_0 + 3\delta$ the set of active processes is composed by at least one process (i.e., $|A(t_0 + 3\delta)| \geq 1$). Let us note that, in the worst case, all the $n \times c$ processes issuing a join() operation at time t_1 are now going to

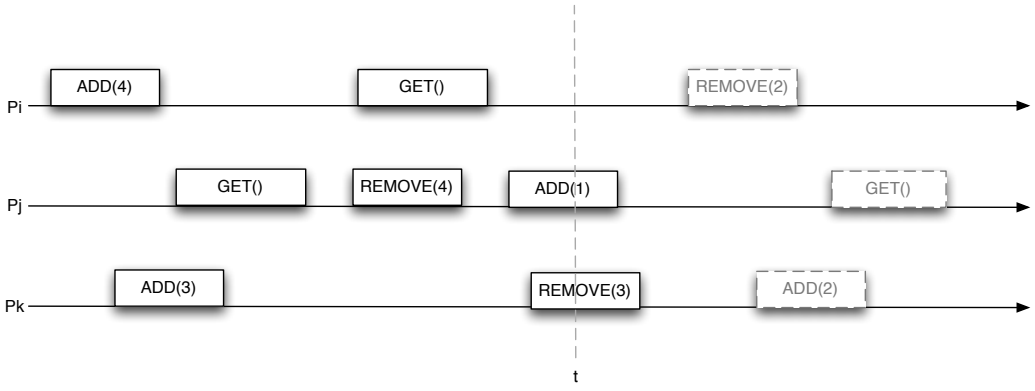


Figure 13: Sub-History $\widehat{H}_t$ at time t of the History $\widehat{H}$

become active Therefore, $|A(t_0 + 3\delta + 1)| \geq 1 + (n \times c) - (n \times c) \geq 1$ (where the terms $+(n \times c)$ represents processes terminating the join() issued at time t_1 and $-(n \times c)$ represents processes leaving at time $t_0 + 3\delta + 1$). It is easy to see that the previous reasoning depends only on (1) the fact that there are n processes at each time t_0, and (2) the definition of the churn rate c. Hence, the previous reasoning can be extended at any time t concluding that $\forall t: \ |A[t, t + 3\delta]| \geq n(1 - 3\delta c)$ and the claim follows. $\square_{Lemma\ 6}$

In order to prove that any get() operation returns an admissible set (Lemma 9), let us first show that every join() operation terminates by storing an admissible set. To this aim, let us introduce the notions of *sub-history $\widehat{H_t}$ of $\widehat{H}$ at time t* and *admissible set of elements at time t*.

Definition 15 (Sub-history $\widehat{H_t}$ of $\widehat{H}$ at time t)**.** *Given an execution history $\widehat{H} = (H, \prec)$ and a time t, the sub-history $\widehat{H_t} = (H_t, \prec)$ of $\widehat{H}$ at time t is the sub-set of $\widehat{H}$ such that:*

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

As an example, consider the history $\widehat{H}$ depicted in Figure 13. The sub-history $\widehat{H}_t$ at the time t is the partial order of all the operations started before t (i.e., H_t contains add$(4)i$, get()i, get()j, remove$(4)j$, add$(1)j$, add$(3)k$ and remove$(3)k$).

Definition 16 (Admissible set of elements at time t)**.** *An admissible set of elements at time t for $\mathcal{S}$ (denoted $\mathcal{V}_{ad}(t)$) is any possible admissible set $V_{ad}(op)$ for an instantaneous*[6] get()*operation op that would be executed at time t.*

As an example, consider the execution of Figure 13. The possible admissible sets at time t are, by definition, all the admissible sets for a "virtual" get() operation op executed instantaneously at

[6] An operation op is instantaneous if and only if $t_B(op) = t_E(op)$.

time t (i.e., $t_B(op) = t_E(op) = t$)[7]. Given the sub-history $\widehat{H}_t$ at time t (represented in Figure 13), it is possible to define the update sub-history $\widehat{U}$ induced by op by removing the operations $\mathsf{get}()i$ and $\mathsf{get}()j$ and the corresponding permutation set is shown in Figure 14.

$$
\begin{aligned}
\Pi_{\widehat{U}} = \{ \quad & \pi_1 = (\mathsf{add}(4)i, \mathsf{add}(3)k, \mathsf{remove}(4)j, op, \mathsf{add}(1)j, \mathsf{remove}(3)k) \\
& \pi_2 = (\mathsf{add}(4)i, \mathsf{add}(3)k, \mathsf{remove}(4)j, op, \mathsf{remove}(3)k, \mathsf{add}(1)j), \\
& \pi_3 = (\mathsf{add}(4)i, \mathsf{add}(3)k, \mathsf{remove}(4)j, \mathsf{add}(1)j, op, \mathsf{remove}(3)k), \\
& \pi_4 = (\mathsf{add}(4)i, \mathsf{add}(3)k, \mathsf{remove}(4)j, \mathsf{add}(1)j, \mathsf{remove}(3)k, op), \\
& \pi_5 = (\mathsf{add}(4)i, \mathsf{add}(3)k, \mathsf{remove}(4)j, \mathsf{remove}(3)k, op, \mathsf{add}(1)j), \\
& \pi_6 = (\mathsf{add}(4)i, \mathsf{add}(3)k, \mathsf{remove}(4)j, \mathsf{remove}(3)k, \mathsf{add}(1)j, op), \\
& \pi_7 = (\mathsf{add}(3)k, \mathsf{add}(4)i, \mathsf{remove}(4)j, op, \mathsf{add}(1)j, \mathsf{remove}(3)k), \\
& \pi_8 = (\mathsf{add}(3)k, \mathsf{add}(4)i, \mathsf{remove}(4)j, op, \mathsf{remove}(3)k, \mathsf{add}(1)j), \\
& \pi_9 = (\mathsf{add}(3)k, \mathsf{add}(4)i, \mathsf{remove}(4)j, \mathsf{add}(1)j, op, \mathsf{remove}(3)k), \\
& \pi_{10} = (\mathsf{add}(3)k, \mathsf{add}(4)i, \mathsf{remove}(4)j, \mathsf{add}(1)j, \mathsf{remove}(3)k, op,) \\
& \pi_{11} = (\mathsf{add}(3)k, \mathsf{add}(4)i, \mathsf{remove}(4)j, \mathsf{remove}(3)k, op, \mathsf{add}(1)j), \\
& \pi_{12} = (\mathsf{add}(3)k, \mathsf{add}(4)i, \mathsf{remove}(4)j, \mathsf{remove}(3)k, \mathsf{add}(1)j, op), \\
\}.
\end{aligned}
$$

Figure 14: Permutation Set induced by op on the Sub-history $\widehat{U}$

Considering all the permutations π_i consistent with $\widehat{U}$ shown above, the four possible admissible sets for op and then possible admissible sets at time t are:

$$
\begin{aligned}
\textit{(i)}\ \ V_{ad}(t) &= \emptyset, \\
\textit{(ii)}\ \ V_{ad}(t) &= \{1\}, \\
\textit{(iii)}\ \ V_{ad}(t) &= \{3\}, \\
\textit{(iv)}\ \ V_{ad}(t) &= \{1, 3\}.
\end{aligned}
$$

In the following Lemma we will show that all the processes that invokes the $\mathsf{join}()$ operation during the fist time unit (i.e., at time t_1) will terminates the operation by storing locally an admissible set. This means that the consistency of the set is preserved while transferring the state to processes arriving during the first time unit. This results will be generalized in Lemma 8 to any $\mathsf{join}()$ operation.

Lemma 7. *Let t_0 be the time at which the computation of a set object $\mathcal{S}$ starts, let $\widehat{H} = (H, \prec)$ be an execution history of $\mathcal{S}$, and $\widehat{H}_{t_1+3\delta} = (H_{t_1+3\delta}, \prec)$ be the sub-history of $\widehat{H}$ at time $t_1 + 3\delta$. Let p_i be a process that invokes $\mathsf{join}()$ on $\mathcal{S}$ at time t_1, if $c < 1/(3\delta)$ then at time $t_1 + 3\delta$ the local copy set_i of $\mathcal{S}$ maintained by p_i will be an admissible set at time $t_1 + 3\delta$.*

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

[7]Note that op is concurrent with $\mathsf{add}(1)j$ and $\mathsf{remove}(3)k$ while it follows $\mathsf{add}(4)i$, $\mathsf{get}()i$, $\mathsf{get}()j$, $\mathsf{remove}(4)j$ and $\mathsf{add}(3)k$.

If set_i is not an admissible set at time $t_1 + 3\delta$ then it is not an admissible set for any operation $op = \mathsf{get}()$ executed instantaneously at time $t_1 + 3\delta$. Considering the sub-history $\widehat{U}$ induced by op on $\widehat{H}$ and the permutation set $\Pi_{\widehat{U}}$, this means that there does not exist a permutation π_i, belonging to the permutation set $\Pi_{\widehat{U}}$ induced by op, such that π_i generates set_i. One of the following conditions holds:

1. There exists an element v, generated by every permutation π_i of the permutation set $\Pi_{\widehat{U}}$, such that v does not belong to set_i (i.e., $\exists\, v : \forall\, \pi_i \in \Pi_{\widehat{U}}\ (\exists\, \mathsf{add}(v) \in\ \pi_i : \mathsf{add}(v) \rightarrow_{\pi_i} op) \wedge (\nexists\, \mathsf{remove}(v) \in \pi_i : \mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op) \wedge (v \notin set_i)$);
2. There exists an element v belonging to set_i such that it can not be generated by any permutation π_i of the permutation set $\Pi_{\widehat{U}}$ (i.e., $\exists\, v \in set_i : \forall \pi_i \in \Pi_{\widehat{U}}\ (\exists\, \mathsf{add}(v), \mathsf{remove}(v) \in\ \pi_i$: $\mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op)$).

Case 1: There exists an element $v \notin set_i$ generated by every permutation $\pi_i \in \Pi_{\widehat{U}}$.

Due to Definition 4.5, if v is generated by every permutation π_i, then in every permutation π_i, (i) $\exists\, \mathsf{add}(v) : \mathsf{add}(v) \rightarrow_{\pi_i} op$ and (ii) $\nexists\, \mathsf{remove}(v)$: $\mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op$.

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

1. If $t_B(\mathsf{add}(v)) \geq t_1$ then p_i is already inside the system when the $\mathsf{add}(v)$ operation starts. Let us note that, since $t_E(\mathsf{add}(v)) < t_1 + 3\delta$ and the $\mathsf{add}()$ operation execution time is bounded by δ, $t_B(\mathsf{add}(v)) \leq t_1 + 2\delta$. Thus, due to the timely delivery property of the broadcast primitive, p_i will receive the update message sent by p_j and will include such operation into the buffer $pending_i$ before time $t_1 + 3\delta$. As a consequence, when at time $t_1 + 3\delta$, p_i will execute lines $07 - 12$ of Figure 10, it will execute the $\mathsf{update}()$ procedure for the buffered $\mathsf{add}(v)$ operation, and thus p_i will insert v in set_i. Since there not exist any $\mathsf{remove}(v)$ operation in π_i between the $\mathsf{add}(v)$ and op, $v \in set_i$ and there is a contradiction.
2. If $t_B(\mathsf{add}(v)) < t_1$ (i.e., $t_B(\mathsf{add}(v)) = t_0$), p_i has no guarantee to receive the update message sent by p_j. However, due to the timely delivery property of the broadcast, every active process between time t_0 and $t_0 + \delta$ will receive this message and it will execute the operation by adding v to its set_j variable. Considering that any update operation is bounded by δ, the wait statement in line 02 assures that such update operation is completed by any active process at time $t_0 + \delta$, i.e., before p_i sends the INQUIRY message. Therefore, any active process that replies to p_i has in its local copy of the set the element v and, due to Lemma 6, there exists at least one process inside the computation that replies to p_i. Upon the reception of such set, p_i will execute line 24 of Figure 10 by including the set_j received (containing v) in its $reply_i$ buffer. Considering that (i) each received set_k will contain v and (ii) there not exists any $\mathsf{remove}(v)$ operation in π_i between the $\mathsf{add}(v)$ and op, this implies $v \in set_i$ contradicting the initial assumption of case 1.

Case 2: There exists an element $v \in set_i$ such that it can not be generated by any permutation $\pi_i \in \Pi_{\widehat{U}}$.

Due to Definition 4.5, if v can not be generated by any permutation π_i, then one of the following

cases can happen:

(2.1) for each π_i, there exists an $\mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op$ or,

(2.2) for each π_i there not exists any $\mathsf{add}(v) \rightarrow_{\pi_i} op$.

Considering that the local variable set_i is empty at the beginning of the join() operation (line 01, Figure 10), if $v \in set_i$ then either (i) v belongs to some set_j received during the join() by p_i from an active process p_j (lines 23 - 24, Figure 10) or (ii) v has been added by p_i while processing the buffer $pending_i$.

Note that, at the beginning of the computation, the value of set_j at each process p_j is an empty set. Thus, if some process p_j sends to p_i a set_j containing v, it follows that p_j has executed either line 03 or line 10 or line 14 of the update() procedure (Figure 9). Such procedure is activated when an UPDATE$(A, v, -, -)$ message is delivered, at a certain time t, to an active process (line 16, Figure 8). Considering that, such UPDATE message is sent by active processes when an add() operation is triggered (line 03, Figure 8) case 2.2 cannot happen; it follows that there exists an $\mathsf{add}(v)$ operation issued before time t such that its invocation time precedes op (i.e., $t_B(\mathsf{add}(v)) < t < t_1 + 3\delta$ and the $\mathsf{add}(v)$ either precedes or is concurrent with op).

As a consequence, there exists at least one permutation π_i, consistent with $\widehat{U}$, where $\mathsf{add}(v) \rightarrow_{\pi_i} op$. Considering the case 2.1, if there exists an $\mathsf{add}(v)$ then there exists also a $\mathsf{remove}(v)$ and, for each permutation π_i in the permutation set, $\mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op$. It follows that $\mathsf{add}(v) \prec \mathsf{remove}(v)$ in $\widehat{H}$.

If $\mathsf{add}(v) \prec \mathsf{remove}(v)$ in $\widehat{H}$ then when a process p_k issued the $\mathsf{remove}(v)$ operation, it has already executed the $\mathsf{add}(v)$ and in particular it has executed line 23 of the update() procedure. Hence, the sequence number attached to the $\mathsf{remove}(v)$ operation will be greater than the one attached to the $\mathsf{add}(v)$. It follows that every active process will execute first the $\mathsf{add}(v)$ and then the $\mathsf{remove}(v)$, and it will execute line 04 of the update() procedure by removing v from their local copies of the set and incrementing its update sequence number. Due to Lemma 6, 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.

Hence, by construction, p_i selects a copy of the set that does not include v because it cannot be generated by any permutation satisfying case 2.1. The same reasoning applies to p_i while processing every element of $pending_i$ buffer. Therefore in both sub-cases the assumption is contradicted.

In both cases we reached a contradiction to the initial assumptions and the claim follows.

$\square_{Lemma\ 7}$

Lemma 8. *Let $\widehat{H} = (H, \prec)$ be the execution history of a set object $\mathcal{S}$, and p_i a process that invokes a* join() *on the set $\mathcal{S}$ at time t. If $c < 1/(3\delta)$ then at time $t + 3\delta$ the local copy set_i of $\mathcal{S}$ maintained by p_i will be an admissible set at time $t + 3\delta$.*

Proof Let suppose by contradiction that the local copy set_i of the set object $\mathcal{S}$, 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 $op = \mathsf{get}()$ executed instantaneously at time $t + 3\delta$. Considering the sub-history $\widehat{U}$ induced by op on $\widehat{H}$ and the permutation set $\Pi_{\widehat{U}}$, this means that there is no permutation π_i, belonging to the permutation set $\Pi_{\widehat{U}}$, such that π_i generates set_i. One of the following conditions holds.

1. there exists an element v, generated by every permutation π_i of the permutation set $\Pi_{\widehat{U}}$, such that v does not belong to set_i (i.e., $\exists\ v : \forall\ \pi_i \in \Pi_{\widehat{U}}\ (\exists\ \mathsf{add}(v) \in\ \pi_i : \mathsf{add}(v) \rightarrow_{\pi_i} op) \wedge (\nexists\ \mathsf{remove}(v) \in \pi_i : \mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op) \wedge (v \notin set_i)$);
2. there exists an element v belonging to set_i such that it can not be generated by any permutation π_i of the permutation set $\Pi_{\widehat{U}}$ (i.e., $\exists\ v \in set_i : \forall \pi_i \in \Pi_{\widehat{U}}\ (\exists\ \mathsf{add}(v), \mathsf{remove}(v) \in\ \pi_i$: $\mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op)$).

Case 1: there exists an element $v \notin set_i$ generated by every permutation $\pi_i \in \Pi_{\widehat{U}}$.
Due to Definition 4.5, if v is generated by every permutation π_i, then in every permutation π_i, (i) $\exists$ $\mathsf{add}(v) : \mathsf{add}(v) \rightarrow_{\pi_i} op$ and (ii) $\nexists\ \mathsf{remove}(v)$: $\mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op$.

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

1. If $t_B(\mathsf{add}(v)) \geq t$ then p_i was already inside the system when the update operation starts and then, due to the timely delivery of the broadcast property, p_i will receive the UPDATE message sent by some process p_j and will include such operation to the buffer $pending_i$. At time $t + 3\delta$, p_i will execute lines $07 - 12$ of Figure 10 and then will execute the $\mathsf{add}(v)$ operation inserting v in set_i. Since there not exist any $\mathsf{remove}(v)$ operation in π_i between the $\mathsf{add}(v)$ and op, v belongs to the set and there is a contradiction to the initial assumption of case 1.
2. If $t_B(\mathsf{add}(v)) < t$, p_i may not receive the 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 $\mathsf{add}(v)$ or has received bot an $\mathsf{add}(v)$ and a $\mathsf{remove}(v)$. Let consider a generic process p_j replying to p_i: it could be active or not at time $t_B(\mathsf{add}(v))$.
   (a) If $p_j \in A(t_B(\mathsf{add}(v)))$ then p_j have executed the update operation and has added v to its local copy of the set. Considering that any update operation is bounded by δ, the wait statement in line 02 assures that such update operation is done by every active process before p_i sends the INQUIRY message. Therefore, any active process that replies to p_i has in its local copy of the set the element v and, due to Lemma 6, there exists at least one process inside the computation from time t to time $t + 3\delta$ that replies to p_i. Upon the reception of such set, p_i will execute line 24 of Figure 10 by including the set_j received (containing v) in its $reply_i$ buffer. At time $t + 3\delta$, 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 set_k will contain v and (ii) there not exists any $\mathsf{remove}(v)$ operation in π_i between the $\mathsf{add}(v)$ and op, v belongs to the set and there is a contradiction (i.e., $v \in set_i$ and $\mathsf{add}(v) \prec op$ and there not exist any $\mathsf{remove}(v)$ in the between).

(b) If $p_j \notin A(t_B(\mathsf{add}(v)))$ then p_j has completed the join after the $\mathsf{add}(v)$ starts (i.e., $t_E(\mathsf{join}(j) > t_B(\mathsf{add}(v)))$. Iterating the reasoning above and considering that, for Lemma 6, there is always at least one active process that replies, we arrive to the situation of Lemma 7: there exist a process p_k such that p_k has received the update from the process issuing the $\mathsf{add}(v)$ operation inserting v in its local copy and then p_k propagates it.
As a consequence, $v \in set_i$ and it is generated by any permutation π_i , thus, we have a contradiction to the assumption of case 1.

Case 2: there exists an element $v \in set_i$ that can not be generated by any permutation $\pi_i \in \Pi_{\widehat{U}}$.

Considering that set_i is empty at the beginning of the $\mathsf{join}()$ operation (line 01, Figure 10), if $v \in set_i$ then either (i) v belongs to some set_j received by some active process p_j (lines 23 - 24, Figure 10) or (ii) v has been added by processing the buffer $pending_i$.

At the beginning of the computation, each process p_j has its variable set_j set to empty. If some process p_j sends to p_i a set_j variable containing v, it follows that p_j has executed either line 03 or line 10 or line 14 of the $\mathsf{update}()$ procedure (Figure 9). Such procedure is activated when an UPDATE message is delivered to an active process (line 16, Figure 8); let t' be the time when p_j delivers the UPDATE$(< A, v, -, - >)$ message and triggers the $\mathsf{update}()$ procedure. Note that, UPDATE messages are sent by active processes when an $\mathsf{add}()$/$\mathsf{remove}()$ operation is triggered (line 03 and line 09, Figure 8).

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

If the $\mathsf{add}(v)$ operation precedes or is concurrent with op then there exists at least one permutation π_i, consistent with $\widehat{U}$ induced by op, where $\mathsf{add}(v) \rightarrow_{\pi_i} op$.

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

If $\mathsf{add}(v) \prec \mathsf{remove}(v)$ in $\widehat{H}$ then when a process p_k issued the $\mathsf{remove}(v)$ operation, it has executed the $\mathsf{add}(v)$ and in particular it has executed line 23 of the $\mathsf{update}()$ procedure. Hence, the sequence number attached to the $\mathsf{remove}(v)$ operation will be grater than the one attached to the $\mathsf{add}(v)$. Therefore every active process will execute first the $\mathsf{add}(v)$ and then the $\mathsf{remove}(v)$, and it will execute line 04 of the $\mathsf{update}()$ procedure by removing v from their local copies of the set.

Due to Lemma 6, 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 7, 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, by construction, p_i selects a copy of the set that does not include v because it cannot be generated by any permutation. The same reasoning applies to p_i while processing every element of $pending_i$ buffer. Therefore the assumption of Case 2 is contradicted.

In both cases we reached a contradiction, hence the claim follows. $\square_{Lemma\ 8}$

The following Lemma proves that the algorithms presented in Section 6.1.1 implements a set object.

Lemma 9. *Let $\mathcal{S}$ be a set object and let op be a* get() *operation issued on $\mathcal{S}$ by some process p_i. If $c < 1/(3\delta)$, the set of elements V returned by op is always an admissible set (i.e., $V = V_{ad}(op)$).*

Proof Let us suppose by contradiction that there exists a process p_i that issues a get() operation op on the set object $\mathcal{S}$ that returns a set of element V not admissible for op. Considering the sub-history $\widehat{U}$ induced by op on $\widehat{H}$ and its the permutation set $\Pi_{\widehat{U}}$, if V is not an admissible set then there not exists any permutation π_i, belonging to the permutation set $\Pi_{\widehat{U}}$ induced by op, such that π_i generates set_i. One of the following cases can occur:

1. there exists an element v, generated by every permutation π_i of the permutation set $\Pi_{\widehat{U}}$, such that v does not belong to set_i (i.e., $\exists\ v : \forall\ \pi_i \in \Pi_{\widehat{U}}\ (\exists\ \mathsf{add}(v) \in\ \pi_i : \mathsf{add}(v) \rightarrow_{\pi_i} op) \wedge (\nexists\ \mathsf{remove}(v) \in \pi_i : \mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op) \wedge (v \notin set_i)$);
2. there exists an element v belonging to set_i such that it can not be generated by any permutation π_i of the permutation set $\Pi_{\widehat{U}}$ (i.e., $\exists\ v \in set_i : \forall \pi_i \in \Pi_{\widehat{U}}\ (\exists\ \mathsf{add}(v), \mathsf{remove}(v) \in\ \pi_i$: $\mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op)$).

Case 1: there exists an element $v \notin set_i$ generated by every permutation $\pi_i \in \Pi_{\widehat{U}}$ and the set V is not admissible. Let p_j be the process issuing the $\mathsf{add}(v)$ operation. Note that, if the $\mathsf{add}(v)$ precedes op in every π_i, it follows that $\mathsf{add}(v)$ precedes op in the execution history $\widehat{H}$. Since there exists the $\mathsf{add}(v)$ operation then there exists also a process p_j issuing such operation that executes the algorithm of Figure 8.

1. If $p_i = p_j$, then p_i has executed lines 04-05 of Figure 8 and has added v to its local copy set_i of the set object. Since there not exists any $\mathsf{remove}(v)$ operation between the $\mathsf{add}(v)$ and op and since the get() operation returns the content of the local copy of the set without modifying it (line 01 Figure 8) then $v \in V$ and we have a contradiction.
2. If $p_i \neq p_j$, then p_j has executed, at some time t, line 03 of Figure 8 by sending an UPDATE (A, v, sn_j, j) message by using the broadcast primitive. Due to the timely delivery property of the broadcast primitive, every process that is active at time t will receives the update up to time $t + \delta$.

   (a) 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 02-05 Figure 8); otherwise it checks if it has in its $last\_ops_i$ an operation already executed that "collides" with the current one (i.e., if there is a $\mathsf{remove}(v)$ of the same element issued by a process with a lower identifier). Since there not exist any $\mathsf{remove}(v)$ operation between the $\mathsf{add}(v)$ and the get(), when process p_i evaluate the condition at line 08 it finds an empty set and the executes lines 10 - 11 by adding v to its local copy of the set (i.e., $v \in set_i$). Since the get() operation returns the content of the local copy of the set without modifying it (line 01 Figure 8) then $v \in V$, V is an admissible set and we have a contradiction.

(b) $p_i \notin A(t)$, it means that p_i is executing or it will execute the join protocol. If p_i is executing the join protocol then it will buffer the update message of p_j, into the $pending_i$ variable, and will execute the update just before becoming active (lines 07-12 Figure 10); p_i will add v to its local copy of the set. Since there not exists any $\mathsf{remove}(v)$ operation between the $\mathsf{add}(v)$ and op and since the $\mathsf{get}()$ operation returns the content of the local copy of the set without modifying it (line 01 Figure 8) then $v \in V$ and we have a contradiction.

If p_i is not joining the system at time t when it will join, p_i asks the local copy of the set to other active processes. Due to Theorem 8, at the end of the join, set_i is admissible and will include v. Even in this case, since there not exist any $\mathsf{remove}(v)$ operation between the $\mathsf{add}(v)$ and op and since the $\mathsf{get}()$ operation returns the content of the local copy of the set without modifying it (line 01 Figure 8) then $v \in set_i$, $V = set_i$ and it is an admissible set and we have a contradiction.

Case 2: there exists an element $v \in set_i$ that can not be generated by any permutation $\pi_i \in \Pi_{\widehat{U}}$ and V is an admissible set.
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.

1. If p_i has added v to set_i as consequence of the join, it means that v is an element of an admissible set at time t of the end of the join (cfr. Theorem 8). If v belongs to an admissible set at time t, it means that v is generated by some permutation π_i belonging to the permutation set $\Pi_{\widehat{U}}$ of the sub-history at time t. It follows that there exists an $\mathsf{add}()$ operation that terminates or is running at time t. Note that, each permutation π_i belonging to the permutation set $\Pi_{\widehat{H}_t}$ is a prefix for at least one permutation belonging to the permutation set $\Pi_{\widehat{H}}$. Since there not exist any $\mathsf{remove}(v)$ operation between the $\mathsf{add}(v)$ and op, v is never removed and we have a contradiction.
2. If p_i has added v to set_i as consequence of an UPDATE message it means that there exists a process p_j that has sent it. An UPDATE message is generated by a process p_j when the $\mathsf{add}()$ operation is issued; this means that there exist an $\mathsf{add}()$ operation that precedes or is concurrent with op. Hence, there exists at least one permutation π_i containing $\mathsf{add}(v)$. Since there not exist any $\mathsf{remove}(v)$ operation between the $\mathsf{add}(v)$ and op in π_i, v is never removed and there is a contradiction.

$\square_{Lemma\ 9}$

Lemma 10. *Let $\mathcal{S}$ be a set object and let $\widehat{H} = (H, \prec)$ be an execution history of $\mathcal{S}$ generated by the algorithm in Figure 10 and Figure 8. If every active process p_i maintains an admissible set, $\widehat{H}$ is always per-element sequential consistent.*

Proof Let us assume by contradiction that there exists an history $\widehat{H} = (H, \prec)$ generated by the algorithms in Figure 10 and Figure 8 such that $\widehat{H}$ is not per-element sequentially consistent. If

$\widehat{H}$ is not per-element sequential consistent then there exist two processes p_i and p_j that admit two linear extensions, respectively $\widehat{S}_i$ and $\widehat{S}_j$, where at least two concurrent operations $op = \mathsf{add}(v)$ and $op' = \mathsf{remove}(v')$ (with $v = v'$) appear in a different order. Without loss of generality, let us suppose that op and op' have no other concurrent operation, op is issued by a process p_h, op' is issued by a process p_k and $t_B(op) < t_B(op')$.
At time $t < t_B(op)$ all the processes have the same value for their sequence numbers[8] and let us suppose that $sn_w = x$ for every p_w. When process p_h issues the $\mathsf{add}(v)$ operation, it increments its sequence number $sn_i = x + 1$ by executing line 02 of Figure 8 and attaches such value to the UPDATE$(A, v, x + 1, h)$ message. We can have two cases: (i) process p_k receives the UPDATE message of p_h before issuing the $\mathsf{remove}(v)$ operation (Figure 15(a)) or (ii) process p_k receives the UPDATE message of p_h just after issuing the $\mathsf{remove}(v)$ operation (Figure 15(b)).

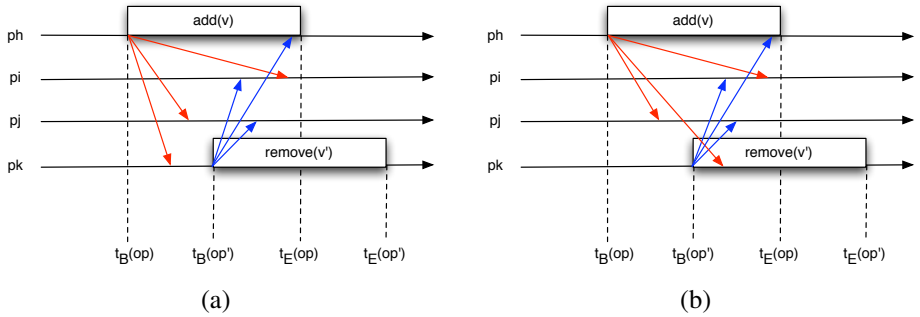


Figure 15: Concurrent execution of $\mathsf{add}(v)$ and $\mathsf{remove}(v)$.

Let us show what happen to p_i and p_j in both cases.
Case i. When p_k receives the 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 03-05) and sets its sequence number to the one received by executing line 23 (i.e., $sn_k = x + 1$). Just after, p_k issues a $\mathsf{remove}(v)$ operation, it increments its sequence number $sn_k = x + 2$ by executing line 02 of Figure 8 and attaches such value to the UPDATE$(R, v, x + 2, k)$ message. Let us consider the scenario depicted in Figure 15(a) where p_i receives first the update message of the $\mathsf{remove}(v)$ sent by p_k and then the update message of the $\mathsf{add}(v)$ sent by p_h while p_j receives first the update message of the $\mathsf{add}(v)$ sent by p_h and then the update message of the $\mathsf{remove}(v)$ sent by p_k.

- **behavior of process** p_i. When p_i receives the 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 03-05 removing the element v (if it is already contained in the set) and storing the tuple $< R, v, x + 2, k >$ in the $last\_ops_i$ set, and then it sets its sequence number to the one received by executing line 23 (i.e., $sn_i = x + 2$). Later, p_i receives the UPDATE$(A, v, x + 1, h)$ message and, since its

[8]Let us recall that the sequence number maintained at each process counts the number of update operations issued on the local copy of the set.

sequence number is now smaller that the received one, it executes lines 07-22. In particular, p_i examines the $last\_ops_i$ buffer and founds an entry (i.e., $< R, v, x+2, k >$) with the value greater than to the one received (line 07) 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 13 is false, then p_i does not execute the update. Note that, not executing the $\mathsf{add}(v)$ is equal to execute $\mathsf{add}(v)$ followed by a $\mathsf{remove}(v)$ operation. Hence, in the linear extension $\widehat{S}_i$ 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 03-05 adding the element 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 23 (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 03-05 removing the element v. Hence, in the linear extension $\widehat{S}_j$ of p_j we have again that $op \rightarrow_{S_j} op'$. Thus, the two operations appear in the same order both in $\widehat{S}_i$ and $\widehat{S}_j$ and this contradicts the initial assumption

Case ii. When p_k issues the $\mathsf{remove}(v)$ operation, it has not yet received the UPDATE messaged sent by p_h and the the two sequence number of both the operation are the same (i.e., $sn_h = sn_k = x+1$). Let us consider the scenario depicted in Figure 15(b) where p_i receives first the update message of the $\mathsf{remove}(v)$ sent by p_k and then the update message of the $\mathsf{add}(v)$ sent by p_h while p_j receives first the update message of the $\mathsf{add}(v)$ sent by p_h and then the update message of the $\mathsf{remove}(v)$ sent by p_k.

- **behavior of process** p_i. When p_i receives the 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 03-05 removing the element 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 23 (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 07-22. In particular, p_i examines the $last\_ops_i$ buffer and founds an entry (i.e., $< R, v, x+1, k >$) with the value equals to the one received (line 07) 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 13 is false, then p_i does not execute the update. Note that, even in this case not executing the $\mathsf{add}(v)$ is equal to execute the $\mathsf{add}(v)$ followed by the $\mathsf{remove}(v)$ operation. Hence, in the linear extension $\widehat{S}_i$ 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 03-05 adding

the element 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 23 (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 07-22. In particular, p_j examines the $last_{ops_j}$ buffer and founds an entry (i.e., $< A, v, x+1, h >$) with the value equals to the one received (07) 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 13 is true, then p_j executes the update removing v. Hence, in the linear extension $\widehat{S}_j$ of p_j we have again that $op \rightarrow_{S_j} op'$. Since the two operations appear in the same order both in $\widehat{S}_i$ and $\widehat{S}_j$, we have a contradiction to the initial assumption.

In both cases we found a contradiction to the initial assumption, then the claim follows.

$\square_{Lemma\ 10}$

We are now in the position to state the following theorem whose proof is a straight application of Lemma 9 and Lemma 10:

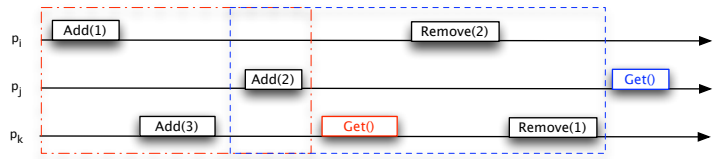
Theorem 3. *Let $\mathcal{S}$ be a set object and let $\widehat{H} = (H, \prec)$ be an execution history of $\mathcal{S}$ generated by the algorithm in Figure 10 and Figure 8. If $c < 1/(3\delta)$, $\widehat{H}$ is always per-element sequential consistent.*

6.2. k-Bounded Set object in an Eventually Synchronous System

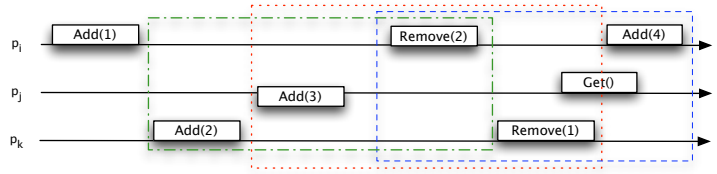
In this section, we are going to consider an eventually synchronous system, i.e., a distributed system where synchrony assumptions hold only after an unknown time t. In order to overcome the impossibility result proved in Section 5, we are going to weaken the specification of the set object by defining a k-bounded set object that is able to work by using finite memory space for local computation and supporting continuous operation invocations (i.e., processes never stop to invoke operations on the object) tolerating continuous churn.

Informally, a k-bounded set object is a set that has limited memory of the execution history and it is able to consider only to the k most recent update operations. In particular, given a get() operation, a k-bounded set behaves as a set having an execution history made by a subset of update operations containing only the k most recent ones. All the other operations are forgotten by the object. Hence, each get() operation defines a "window" on the execution history and operations out of the window are hidden to the k-bounded set as if they are never invoked.

As an example, let us consider the execution history shown in Figure 16(a) for a k-bounded set object where $k = 3$. For each of the two get() operations, the execution history is restricted to the last 3 update operations. If we consider the 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 get() invoked on a set). Considering the second get() invoked by p_j, the set returned will be $\emptyset$, instead of $\{3\}$ that would be returned by the same get() invoked on a set; this is actually due to the add(1) and add(3) operations that are out of the window for such a get() and then they appear as if they are never invoked. 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.



(a)



(b)

Figure 16: 3-bounded set shared object

Let us note that assuming the existence of an upper bound k on number of operations issued on the object, the k-bounded set object falls down in to a set. Thus, any implementation of a k-bounded set object is also an implementation for a set assuming the number of updated operation bounded.

In the following we will first provide the specification of the k-bounded set object and then will show an algorithm implementing such object.

6.2.1. Preliminary Definitions

Due to the concurrency, the meaning of "most recent update operations" may be not well defined. As an example, consider the execution shown in Figure 16(b). Given the get() operation issued by p_j, should the add(4) issued by p_i and the remove(1) issued by p_k be considered as recent operations? Depending on which operations are considered or not, different sets of recent operations are considered (i.e., the three different dotted rectangles in the Figure). To solve this ambiguity, 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 $\Pi_{\widehat{H}}$ of the execution by considering for each permutation its k-cut.

Definition 17 (k-cut permutation induced by op_i)**.** *Given an execution history $\widehat{H} = (H, \prec)$ let $\pi = (op_1, op_2, \ldots, op_n)$ a permutation consistent with $\widehat{H}$. Given an operation op_i of π and an integer k, the k-*cut permutation *induced by op_i on π, denoted $\pi_k(op_i)$, is the sub-set of π ending with op_i and including the k operations that precede op_i in π (i.e., $\pi_k(op_i) = (op_{i-k}, \ldots, op_{i-1}, op_i)$).*

As an example, consider the execution history $\widehat{H}$ shown in Figure 3 and consider the permuta-

$$
\begin{aligned}
\Pi_{3,\widehat{H}}(op) = \{ \quad & \pi_1 = (\mathsf{add}(2)j, \mathsf{remove}(2)i, \mathsf{remove}(1)k, \mathsf{get}()j), \\
& \pi_2 = (\mathsf{remove}(2)i, \mathsf{remove}(1)k, \mathsf{add}(3)i, \mathsf{get}()j), \\
& \pi_3 = (\mathsf{get}()k, \mathsf{add}(2)j, \mathsf{remove}(2)i, \mathsf{get}()j), \\
& \pi_4 = (\mathsf{add}(2)j, \mathsf{remove}(2)i, \mathsf{add}(3)i, \mathsf{get}()j), \\
& \pi_5 = (\mathsf{remove}(2)i, \mathsf{add}(3)i, \mathsf{remove}(1)k, \mathsf{get}()j), \\
& \pi_6 = (\mathsf{get}()k, \mathsf{remove}(2)j, \mathsf{add}(2)i, \mathsf{get}()j), \\
& \pi_7 = (\mathsf{remove}(2)j, \mathsf{add}(2)i, \mathsf{add}(3)i, \mathsf{get}()j), \\
& \pi_8 = (\mathsf{add}(2)i, \mathsf{add}(3)i, \mathsf{remove}(1)k, \mathsf{get}()j), \\
& \pi_9 = (\mathsf{add}(2)i, \mathsf{remove}(1)k, \mathsf{add}(3)i, \mathsf{get}()j), \\
& \pi_{10} = (\mathsf{remove}(2)j, \mathsf{add}(2)i, \mathsf{remove}(1)k, \mathsf{get}()j) \\
\} &
\end{aligned}
$$

Figure 17: 3-cut Permutation Set induced by a get()operation

tion $\pi_1 = (\mathsf{add}(1)i, \mathsf{get}()k, \mathsf{add}(2)j, \mathsf{remove}(2)i, \mathsf{remove}(1)k, \mathsf{get}()j, \mathsf{add}(3)i)$ consistent with $\widehat{H}$. If $op = \mathsf{get}()j$ and $k = 3$, the 3-cut permutation induced by op on π_1 is $\pi_{1_3}(op)$=($\mathsf{add}(2)j$, $\mathsf{remove}(2)i$, $\mathsf{remove}(1)k$, $\mathsf{get}()j$).

Definition 18 (k-cut permutation set induced by op_i)**.** *Given an execution history $\widehat{H} = (H, \prec)$ let $\Pi_{\widehat{H}}$ its permutation set. Given an operation op of H and an integer k, the k-cut permutation set induced by op on $\Pi_{\widehat{H}}$, denoted $\Pi_{k,\widehat{H}}(op)$, is the set of all the k-cut permutations induced by op on each permutation π of $\Pi_{\widehat{H}}$.*

As an example, consider the execution history $\widehat{H}$ depicted in Figure 3 and consider the permutation set $\Pi_{\widehat{H}}$ of the execution history $\widehat{H}$ shown above. If op=$\mathsf{get}()j$ and $k = 3$, the 3-cut permutation set induced by op is shown in Figure 17.

Definition 19 (Admissible set for a get() operation)**.** *Given an execution history $\widehat{H} = (H, \prec)$ of a k-bounded set object, let op = $\mathsf{get}()$ be an operation of H. Let $\widehat{U}_{\widehat{H},op}$ be the sub-history induced by op on $\widehat{H}$ and let k be an integer. An admissible set for op, denoted $V_{ad}(op)$, is any set generated by any permutation π belonging to the k-cut permutation set $\Pi_{k,\widehat{U}_{\widehat{H},op}}$*

As an example, consider the execution history $\widehat{H}$ shown in Figure 3 and its sub-history $\widehat{U}$ induced by the operation op =$\mathsf{get}()j$. Given its 3-cut permutation set $\Pi_{3,\widehat{U}}$, all the possible admissible sets for op are:

- $V_{ad\ 1} = \emptyset$,
- $V_{ad\ 2} = V_{ad\ 4} = V_{ad\ 5} = \{3\}$,
- $V_{ad\ 3} = \{1\}$,
- $V_{ad\ 6} = \{1, 2\}$,
- $V_{ad\ 7} = V_{ad\ 8} = V_{ad\ 9} = \{2, 3\}$,
- $V_{ad\ 10} = \{2\}$.

6.2.2. Eventually Synchronous Distributed System

We consider the eventually synchronous distributed system as presented in Section 5.2. In such system the the two communication primitives (point-to-point channels and broadcast) can be formalized as follows.

Point-to- Point communication: the network allows a process p_i to send a message to another process p_j as soon as p_i knows that p_j is connected to the system. Moreover, there exist a time t and an upper bound δ' such that any message sent at time $t' \geq t$ from p_i, is received up to time $t' + \delta'$ by p_j, if it has not left the system.

Broadcast. As in the synchronous case, it is assumed that the CCS offers a broadcast communication primitive that provides processes with two operations, denoted broadcast MSG() and deliver MSG(). 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 "broadcasted" and "delivered". These operations satisfy the following property:

- Broadcast Eventual Timely Delivery: There is a time t and a bound $\delta > \delta'$ 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 $[t', t' + \delta]$.

Let us note that, similarly to the case of point-to-point communication primitive, the constant δ represents an upper bound on the transmission delay of the broadcast when the synchrony period arrives. For the ease of presentation, in the following we will use only the constant δ as it is an upper bound for both the point-to-point transmission delay and the broadcast transmission delay.

Finally, let us remark that the time t after which the system becomes synchronous and the upper bound δ message delivery times are not known by processes.

6.2.3. The Protocol

The protocol implementing a k-bounded set object is based on the following assumptions:

1. there always exists a majority of active processes participating in the distributed computation (i.e., the churn rate is such that $\forall t, |A(t)| > \frac{n}{2}$);
2. each process remains in the distributed computation for at least 3δ time units i.e., a process p_i invoking the join() operation at time t cannot leave the distributed computation before time $t + 3\delta$.

The first assumption guarantees the termination of the algorithms while the second one guarantees that a process remains in the computation long enough to ensure (when the synchrony assumptions hold) a successful state transfer to joining processes.

The basic idea of the protocol is to follow the quorum-based approach. The execution of each operation involves at least $\lceil \frac{n}{2} \rceil + 1$ active processes (i.e., a *quorum*) that will be witnesses for the operation and each operation is associated with a timestamp (i.e., a sequence number together with the process identifier). Each process p_i keeps locally the same variables used for the synchronous case and, in addition, it stores a partial history containing only k update operations.

More in details, when a process p_i wants to update the content of the k-bounded set (either by adding or removing an element), it asks to other processes to update the object by sending the element, the type of the update (add or remove) and other relevant information needed to preserve the consistency (and discussed in the following). Receiving such update request, any active process p_j will take care of it by (i) updating the partial history in case the request is new and (ii) re-computing the content of the object; in any case, p_j will answer by sending back an acknowledgement for the update. Once p_i received a majority of acknowledgement the operation terminates.

Let us note that, due the temporary asynchrony of the system, the get() operation cannot be executed locally as an update request may arrive to a certain process after that the update operation is complete. Therefore, also the get() operation is executed by involving the participation of a quorum. In particular, when a process p_i wants to obtain the current content of the object, it asks to all the other processes that will answer by sending back the partial history of update operations (if active). When a majority of histories are received, p_i computes the union of such histories, order the update operations in the union according to their timestamps and then computes the content of the object by executing the operations in order and returning the results.

Finally, the join() operation is implemented as a particular type of get() where processes buffer the operation requests received during the execution (i.e., while they are not yet active) and postpone their execution at the end of operation.

Informally, the existence of quorums for each update operation and the dissemination of the partial history of update operations guarantee that there will always exist a witness for each update operation able to transfer this information to joining processes (i.e., for each update operation op in the partial history, the intersection between the quorum of processes acknowledging op and the quorum of processes answering to a join() is non empty). As in the previous case, the consistency of the object is preserved by the total order on the update operations imposed by the timestamps.

Local variables at process p_i. Each process maintains locally the following variables:

- A set variable denoted set_i containing the local copy of the k-bounded set.
- two integers $update\_sn_i$ and get_sn_i that represent the sequence numbers used by p_i to distinguish its successive update (add() and remove()) and get() operations respectively. Such integers, together with the id of the process represent the timestamp of the request.
- a variable $running_i$ that stores the tuple corresponding to the update operation (add() or remove()) currently executed by p_i. It is initialized to a default value $< null, \perp, 0, i >$.
- A set variable $last\_ops_i$, that contain at most k entries, used to maintain an history of recent update operations executed by p_i. Such variable contains 4-tuples $< type, val, sn, id >$ each one identifying an update operation of type $type = \{A \ or \ R\}$ of the element val, with a sequence number sn, issued by a process with identity id.
- A boolean $active_i$, initialized to $false$, that is switched to $true$ just after p_i has joined the computation.

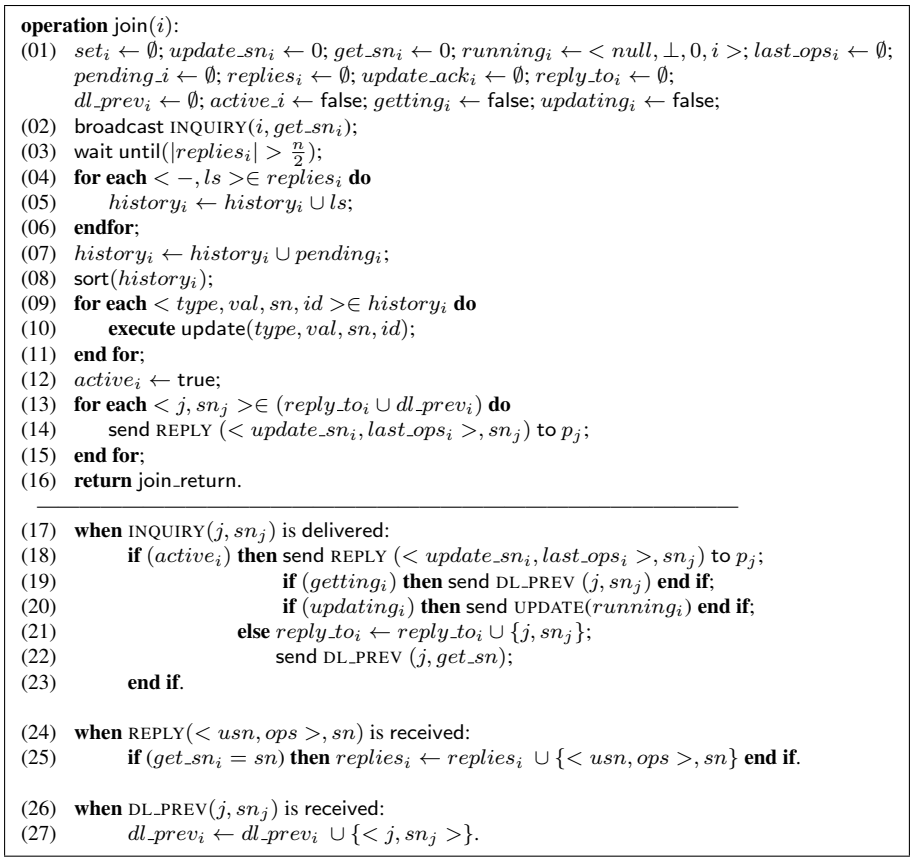


Figure 18: The join() protocol for a k-bounded set object in an eventually synchronous system (code for p_i)

- Two boolean variables $getting_i$ and $updating_i$ whose value is true when p_i is executing respectively a get() operation or an add() or remove() operation.
- 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 computation. 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 computation concurrently with p_i (as far as p_i knows). The set $pending_i$ contains the 4-tuples $< type, val, sn, id >$ 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 (i.e., it terminates 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.
- The $update\_ack_i$ set is used to collect processes that have acknowledged add or remove operations issued by p_i.

The join() **operation.** The code for the join() operation in shown in Figure 18.

After initializing its local variables, p_i broadcasts an INQUIRY(i, get_sn_i) message to inform other processes that it enters the computation and wants to obtain the history of most recent operations (line 02).

When a process p_j receives an INQUIRY(i, sn_i) message if it is active, it answers to p_i by sending back a REPLY$(< update_sn_j, last_ops_j >, sn_i)$ message containing its local variables (line 18-20). In addition, p_j checks its state: in case it is accessing the object by means of a get() operation, it sends also a DL_PREV() message while, in case it is accessing the set by means of either an add() or remove() operation, p_j sends also an UPDATE() message to avoid that p_i misses its update request. If p_i is not active, it postpones its answer until it becomes active (line 21 and line 14) and sends a DL_PREV() message (line 22).

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 25). While, receiving a DL_PREV(j, sn_j) message, p_i adds its content to the set dl_prev_i (line 27), in order to remember that it has to send a reply to p_j when it will become active (lines 13-15).

When p_i has received a majority of replies (line 03), it creates the "global" history of operations by doing the union of all the partial history received so far (lines 04-06). Then, p_i adds operations received during the waiting period to this history (line 07) and orders the obtained list by using the sort() function (line 08): this function simply orders the variable $history_i$ according to the lexicographic order of the pairs (sn, id).

Now, for each operation in the $history_i$ set, p_i executes the UPDATE() procedure (lines 09-11 and discussed later). The aim of such procedure is to (i) keep limited the size of the $last\_ops_i$ variable to k elements by storing the k most recent update operations known by p_i, (ii) keep the set_i variable consistent with the list of operations stored in $last\_ops_i$ and (iii) keep the sequence number updated (further details later in the section).

At the end of the execution of the update() procedure, p_i becomes active (line 12). As a consequence p_i can answer the inquiries it has received during the join() operation and p_i does it if $reply\_to \neq \emptyset$. In addition, p_i sends a reply message also to the processes in its set dl_prev_i (that are supposed to be processes accessing the object to get its content) to prevents them from waiting forever (line 13-15). Finally, p_i returns join_return event to indicate the end of the join() operation (line 16).

The update() **procedure.** The protocol of the update() procedure is shown in Figure 19. Such procedure is triggered by every operation and its aim is to keep updated the local variables of a process p_i according to the most recent k operations. It is executed atomically.

When a process p_i executes the update$(< type, val, sn_j, j >)$ procedure, it first sends an ACK(sn_j, i) message to prevent that process p_i remains blocked forever while executing an update operation (line 01) and then it 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 $last\_ops_i$ variable, the one with the smaller pair (sn, id) (line 02) and then, if it stores less than k operations, it puts the current one in the $last\_ops_i$ variable (line 03). Contrarily, p_i substitutes (if possible) the one with smallest pair (sn, id) with the current one (lines 04-07). Once p_i has the correct set of recent operations, it empties the variable set_i corresponding to the local copy of the object (line 09), sorts the set of

```
procedure update(< type, val, sn_j, j >) % at any process p_i %
(01)  send ACK (sn_j, i) to p_j;
(02)  let < −, −, sn, id >∈ last_ops_i
        such that (∀ < −, −, sn', id' >∈ last_ops_i : (sn, id) ≤ (sn', id'));
(03)  if (|last_ops_i| < k) then last_ops_i ← last_ops_i ∪ {< type, val, sn_j, j >};
(04)                        else if ((sn_j, j) > (sn, id))
(05)                                then last_ops_i ← last_ops_i \ {< −, −, sn, id >};
(06)                                     last_ops_i ← last_ops_i ∪ {< type, val, sn_j, j >}
(07)                             end if
(08)  end if
(09)  set_i ← ∅;
(10)  sort (last_ops_i);
(11)  for each < type, val, sn, id >∈ last_ops_i
(12)                        if (type = A) then set_i ← set_i ∪ {val};
(13)                                      else set_i ← set_i \ {val}
(14)                        end if
(15)  endfor
(16)  update_sn_i ← max(update_sn_i, sn_j).
```

Figure 19: The UPDATE() protocol for an eventually synchronous system (code for p_i)

recent operations according the lexicographic order of pairs (sn, id) (line 10) and then executes the operations contained in the $last\_ops_i$ 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).

The get() **operation.** The pseudo-code of the get() operation is described in Figure 20. A get() operation is actually a simplified version of the join() operation described so far.

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 (line 04)[9]. So, p_i first empties the sets used to store information during the operation (lines 01-03), sets its status $getting_i$ to true (line 03) and then it broadcasts a get request GET(i, get_sn_i).

When a process p_j receives a message GET(i, sn_i), if it is active, p_j replies to p_i by sending back a REPLY$(< update_sn_j, last_ops_j >, sn_i)$ message containing its local variables (line 18). If p_j is not active, it postpones the reply until it becomes active (line 19).

When p_i receives a majority of replies, it 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 06-10). Then p_i orders the history obtained by invoking the SORT function (line 11) and it executes all the operations by invoking the update() procedure (lines 12-14). Finally, p_i sets $getting$ to false (line 15) and then returns the set (line 16).

The add(v) **and** remove(v) **operations.** The code for the add(v) and remove(v) operation is similar and it is shown in Figure 21.

When a process p_i wants to update (i.e., by adding or removing an element to the k-bounded set), it first performs a get()operation to obtain the most updated sequence number (line 01, line 09), it increments the $update\_sn_i$ (line 02, line 10), it empties its $update\_ack_i$ set to store the

[9]The invocation corresponding to the pair $(i, 0)$ is the join() operation issued by p_i.

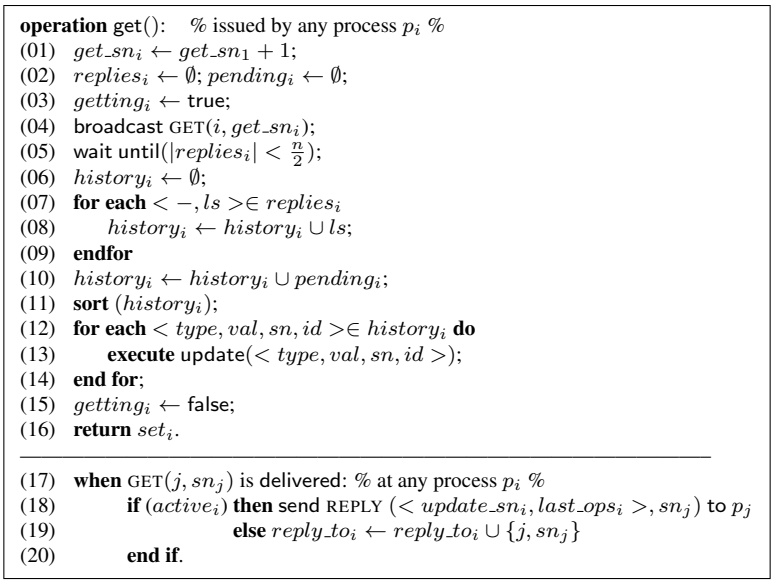


Figure 20: The get() protocol for a k-bounded set object in an eventually synchronous system (code for p_i)

acknowledgements to the current update operation (line 03, line 11), sets its state to $updating$ and stores the 4-tuple corresponding to the current operation (line 04, line 12); then p_i sends an UPDATE() message to perform the current update (line 05, line 13).

When the UPDATE$(< type, val, sn_i, i >)$ message sent from p_i is delivered to some process p_j, it checks if it is active or not. If p_j is not active, it puts the 4-tuple corresponding to the current operation into its $pending_j$ set (line 18) to process the operation at the end of the join() operation when it will be active (line 07, Figure 18). Contrarily, p_j executes the update() procedure for the current operation (line 19) where an ACK() message is sent.

When an 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).

When p_i receives a majority of acknowledgement (line 06, line 14), it resets its state by setting its variable $update_i$ to false and its $running_i$ variable to a default value (line 07, line 15) and finally it returns from the operation (line 08, line 16).

6.2.4. Correctness Proofs

In the following we are going to prove that the algorithms shown in Figures 18 - 21 implements a k-bounded set object with per-element sequential consistency.
In particular, Lemmas 11 - 13 and the following Theorem 4 prove that all the algorithms implementing each operation eventually terminate. Theorem 5 proves that each get() operation returns an admissible set of the object by exploiting the following intermediate results: (i) the timestamps associated to update operations made by the pair $< sn, id >$ (where sn is a sequence number and id is the process identifier) induce a total order among update operations (Lemma 14), (ii) at the end of every join() operation, the variable $last\_ops_i$ stores a k-cut permutation induced by the operation itself (Lemma 15), and (iii) at the end of every get() operation, the variable $last\_ops_i$

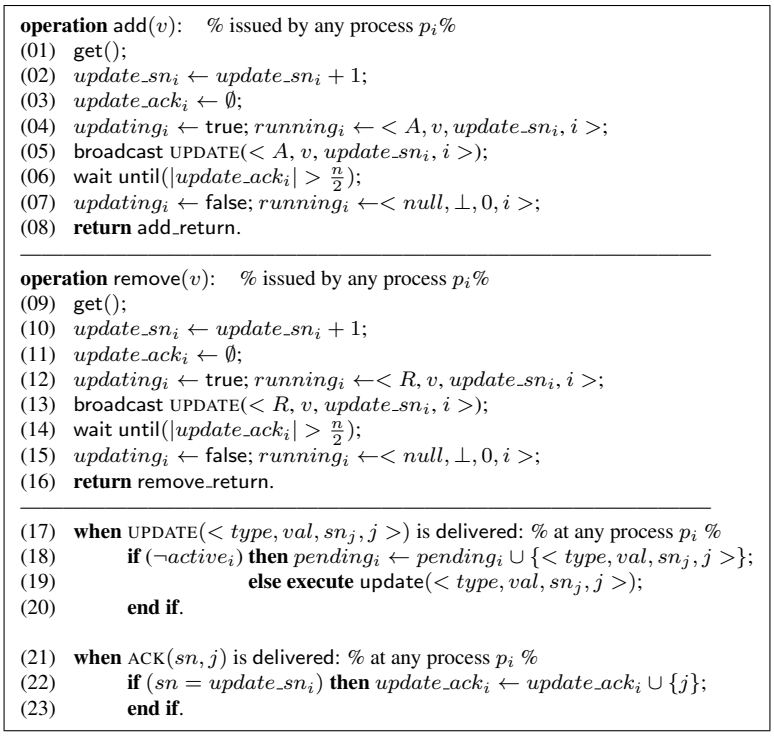


Figure 21: The add() and remove() protocol for a k-bounded set object in an eventually synchronous system (code for p_i)

stores a k-cut permutation induced by the operation itself (Lemma 16). Finally, Theorem 6 proves the per-element based sequential consistency.

Lemma 11. *Let n be the number of processes belonging to the computation at time t_0 and let p_i be a process invoking a* join() *operation. If p_i does not leave the system for at least 3δ time units and at any time t, $|A(t)| > \lceil \frac{n}{2} \rceil$, then p_i eventually returns from the* join() *operation.*

Proof Let us first observe that, in order to terminate its join() operation, a process p_i has to wait until its set $replies_i$ contains, at least, $\lceil \frac{n}{2} \rceil$ elements (line 03, Figure 18). This set is filled in by p_i when it receives the REPLY() messages for the current operation (line 25, Figure 18). A process p_j sends a REPLY() message to p_i if (i) it is active and has received an INQUIRY message from p_i, (line 18, Figure 18), or (ii) it terminates its join() operation and $< i, - > \in reply\_to_j \cup dl\_prev_j$ (lines 13-15, Figure 18).

Let us suppose by contradiction that $|replies_i|$ remains smaller than $\lceil \frac{n}{2} \rceil$. This means that p_i does not receive enough REPLY() carrying the appropriate sequence number. Let t be the time at which the system becomes synchronous and let us consider a time $t' > t$ at which a new process p_j invokes the join operation. At time t', p_j broadcasts an INQUIRY message (line 02, Figure 18). As the system is synchronous from time t, every process present in the system during $[t', t' + \delta]$ receives such INQUIRY message by time $t' + \delta$.

As p_i is not active when it receives p_j's INQUIRY message, p_i executes line 22 of Figure 18 and sends back a DL_PREV message to p_j. Due to the assumption that every process joining the

computation remains inside the computation for at least 3δ time units, p_j receives p_i's DL_PREV and executes consequently line 27 of Figure 18 by adding $< i, - >$ to dl_prev_j. Due to the assumption that there are always at least $\lceil \frac{n}{2} \rceil$ active processes in the system, we have that at time $t' + \delta$ at least $\lceil \frac{n}{2} \rceil$ processes receive the INQUIRY message of p_j, and each of them will execute line 18 of Figure 18 and will send a 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 12, Figure 18). Consequently (lines 13-15) p_j sends a REPLY to p_i as $i \in reply\_to_j \cup dl\_prev_j$. By δ time units, p_i receives that REPLY message and executes line 25, Figure 18. 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 $replies_i$ and terminate its join() operation. $\square_{Lemma\ 11}$

Lemma 12. *Let n be the number of processes belonging to the computation at time t_0. If (i) at any time t, $|A(t)| > \lceil \frac{n}{2} \rceil$ and (ii) each process that invokes a* join() *operation does not leave the system for at least 3δ time units, then a process p_i invoking a* get() *operation and not leaving the computation eventually returns from such operation.*

Proof Since the get() is a simplified case of a join(), the proof is the same of Lemma 11. $\square_{Lemma\ 12}$

Lemma 13. *Let n be the number of processes belonging to the computation at time t_0. If (i) at any time t, $|A(t)| > \lceil \frac{n}{2} \rceil$ and (ii) each process that invokes a* join() *operation does not leave the system for at least 3δ time units then a process p_i that invokes an* add() *operation or a* remove() *operation and does not leave the system eventually returns from such operation.*

Proof Let us first assume that the get() operation invoked at line 01 and line 09 terminates (this is proved in Lemma 12). Before terminating the add() (or remove()) of an element v with an update sequence number usn a process p_i has to wait until its set $update\_ack_i$ contains at least $\lceil \frac{n}{2} \rceil$ elements (line 06 and line 14, Figure 21).

Empty at the beginning of the add(v) operation (line 03, Figure 21) and at the beginning of the remove(v) operation (line 11, Figure 21), this set is filled in when the ACK($usn, -$) messages are delivered to p_i (line 22, Figure 21).

Such an ack message is sent by every process p_j when the update() procedure is activated due to (i) the receipt of an UPDATE message from a process p_i (line 19, Figure 21) or (ii) the termination of a join() operation and $< -, -, usn, i > \in history_j \cup pending_j$ (lines 09 - 11, Figure 18).

Suppose by contradiction that p_i never fills in $update\_ack_i$. This means that p_i misses ACK() messages carrying the sequence number 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_i will receive all the INQUIRY messages sent by processes joining after time t.

When it receives an INQUIRY() message from some joining process p_j, p_i executes lines 18- 21

of Figure 18[10] and sends a REPLY message to p_j as reply to its request and forwards an UPDATE message with the sequence number usn.

Since, after t, the system is synchronous, p_j receives both the REPLY message and the UPDATE message in at most δ time units. When p_j receives the UPDATE message, since it is not yet active, it puts the current update $< -, -, usn, i >$ in the $pending_j$ set (line 18, Figure 21). Due to Lemma 11, p_j will terminate the join executing lines 07-11 of Figure 18 by processing the pending updates (including the one of p_i) and sending the ACK$(ubn, -)$ message to p_i (line 01, Figure 19).

As (1) by assumption a process that joins the system does not leave for at least 3δ time units and (2) the system is now synchronous, such an ACK$(usn, -)$ message is received by p_i in at most δ time units and consequently p_i executes line 22, Figure 21 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 INQUIRY(), UPDATE(), ACK(), the reception of each message triggers the sending of the next one. It follows that p_i eventually receives $\lceil \frac{n}{2} \rceil$ ACK$(usn, -)$ messages and terminates its update operation.

$\square_{Lemma\ 13}$

Theorem 4. *Let n be the number of processes belonging to the computation at time t_0. If (i) at any time t, $|A(t)| > \lceil \frac{n}{2} \rceil$ and (ii) each process that invokes a* join() *operation does not leave the system for at least 3δ time units, then a process p_i that invokes a* join(), get(), add() *or* remove() *operation and does not leave the system eventually returns from its operation.*

Proof It follows from Lemma 11, Lemma 12 and Lemma 13. $\square_{Theorem\ 4}$

Lemma 14. *Let $\widehat{H} = (H, \prec)$ an execution history of a k-bounded set object $\mathcal{S}$. Given the algorithms shown in Figures 18-19, there exists a total order of the pairs (sn, id) that identify each* add(v) *and* remove(v) *operation such that the total order is consistent with the partial order given by the execution history $\widehat{H}$.*

Proof Let op and op' two update operations and let $sn(op)$ and $sn(op')$ the corresponding sequence numbers associated to the operations by the algorithm.

Case 1: $\mathbf{op \prec op' \Rightarrow sn(op) < sn(op')}$. 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. The following may happen:

- $\nexists\ op'' : op \prec op'' \prec op'$: when p_i starts an update operation (both add(v) or remove(v)), it sends an UPDATE$(< -, -, x, i >)$ message and waits until it receives at least $(\frac{n}{2} + 1)$ ACK messages carrying the sequence number x (line 06 and line14, Figure 21). This means that in the system there exist at least $(\frac{n}{2} + 1)$ processes that execute line 16, Figure 19 and

[10]The process p_i that issues an update operation always executes lines 18-20 of Figure 18 because it is always in the active mode.

update their $update\_sn_k$ to a value that is greater equal to x. When p_j starts its operation, it first issues a get() to retrieve the most up-to-date sequence number (line 01 and line 09, Figure 21). During the get()operation, processes sent to p_j their local copies of the list $last\_ops_k$ and to terminate the get(), p_j executes the update() procedure by updating its $update\_sn_j$ variable. Since, at any time t, $|A(t)| < \lceil \frac{n}{2} \rceil$ there exists at least one process that acknowledged the operation of p_i and that sent a sequence number that is at least x. Doing the update, p_j always selects the maximum sequence number and at the end of the get() it has in its $update\_sn_j$ variable a value grater equal than x. Since before sending the UPDATE message p_j increments its sequence number (line 02 and line 10, Figure 21) the operation of p_j will have a sequence number grater that p_i and there is a contradiction.

- $\exists op'', \ldots, op^i : op \prec op'' \prec \cdots \prec op^i \prec op'$: 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: $\mathbf{op||op'}$. 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 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.

$\square_{Lemma\ 14}$

Lemma 15. *Let $\widehat{H} = (H, \prec)$ an execution history of a k-bounded set object $\mathcal{S}$ and let p_i be a process that invokes a* join() *operation at time t. Let op be an instantaneous* get() *operation issued at time t_E(join) and $\widehat{U}$ be the sub-history of $\widehat{H}$ induced by op. Given an integer k, if, at any time t, $|A(t)| > \lceil \frac{n}{2} \rceil$, then at the end of the* join() *operation, p_i maintains in its $last\_ops_i$ variable a k-cut permutation induced by op consistent with $\widehat{U}$.*

Proof In order to terminate the join(), a process p_i has to wait until it receives at least $(\lceil \frac{n}{2} \rceil + 1)$ replies from other processes (line 03, Figure 18). 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 $\mathcal{O}$ be the set of operations started before the end of the join() of p_i.

- $|\mathcal{O}| \leq \mathbf{k}$. Since p_i is the first process that concludes the 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., $A(t_E(\text{join})) \subset A(t_0)$). Therefore, for each update operation op belonging to $\mathcal{O}$, there exists at least one process p_k that has sent an ACK() message for op (line 01, Figure 19), that has inserted a tuple $< -, -, sn, id >$ in its $last\_ops_k$ variable (line 03, Figure 19) and that

has sent a REPLY() message to p_i[11]. When p_i executes lines 04-06 of Figure 18, it obtains an $history_i$ variable containing all the operations terminated before the join() invocation, and possibly some of the concurrent ones. Executing line 08 of Figure 18, p_i will obtain an history totally ordered by the pairs (sn, id) and due to Lemma 14 this total order is consistent with the partial order of the execution history $\widehat{U}$. Since the update() procedure is executed following such total order and considering that all the operations will be stored in the $last\_ops_i$ variable, it is possible to conclude that at the end of the join() operation, the $last\_ops_i$ variable contains a k-cut permutation consistent with $\widehat{U}$.

- $|\mathcal{O}|>$ k. Let consider the worse case where there exists a process p_j that replies to all the $|\mathcal{O}|$ operations. Note that, for the first k UPDATE messages received by p_j, it always executes line 03 of Figure 19 because it has empty slots in its $last\_ops_j$ variable. When p_j receives the $(k+1)$-th UPDATE message, it executes line 04, Figure 19 and checks if the current update is "new" with respect to the ones contained in its $last\_ops_j$ variable, and if it is so, it substitutes the one with the smaller pair (sn, id) with the new one (line 05-06, Figure 19). 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 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., $A(t_E(\mathsf{join})) \subset A(t_0)$). Therefore, for each update operation op belonging to $\mathcal{O}$, there exists at least one process p_k that has sent an ACK() message for op (line 01, Figure 19), that has inserted a tuple $< -, -, sn, id >$ in its $last\_ops_k$ variable (line 03, Figure 19). Considering that an operation op is deleted by the $last\_ops_k$ 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 process that replies to p_i, follows that even in this case at the end of the join() operation, the $last\_ops_i$ variable contains a k-cut permutation consistent with $\widehat{U}$.

Step i. Let p_i be the i-th process that terminates the join() operation. Let $\mathcal{P}$ be the set of processes that send a 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 $last\_ops_k$ variable where, for each new update, they substitute the "older" operation and then, their list represents k-cut permutation consistent with $\widehat{U}$. 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.

□$_{Lemma\ 15}$

Lemma 16. *Let $\widehat{H} = (H, \prec)$ an execution history of a k-bounded set object $\mathcal{S}$ and let p_i be a process that invokes a* get()*operation op. Let $\widehat{U}$ be the sub-history of $\widehat{H}$ induced by op. Given an*

[11]Note that, since the total number of operations is smaller than k, processing the UPDATE message, each process always executes line 03, Figure 19.

integer k, if at any time t, $|A(t)| > \lceil \frac{n}{2} \rceil$, then at the end of the get() *operation, p_i maintains in its $last\_ops_i$ variable a k-cut permutation induced by op consistent with $\widehat{U}$.*

Proof In order to terminate the get(), a process p_i has to wait until it receives at least $(\frac{n}{2} + 1)$ replies from other processes (line 05, Figure 20). 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 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 $last\_ops_k$ 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 get()consistent with $\widehat{U}$.
Each process p_j that became active after t_0, due to Lemma 15, terminates its join() operation maintaing in its local variable $last\_ops_j$ a k-cut permutation consistent with $\widehat{U}$ at time t_E(join) and then for each new update, it substitutes the "older" operation. Hence, also for such processes the list sent to p_i represents a k-cut permutation consistent with $\widehat{U}$ and the Lemma follows. $\square_{Lemma\ 16}$

Theorem 5. *Let $\mathcal{S}$ be a k-bounded set object and let op be a* get() *operation issued on $\mathcal{S}$ by some process p_i. If, at any time t, $|A(t)| > \lceil \frac{n}{2} \rceil$, then the set V returned by op is an admissible set (i.e., $V = V_{ad}(op)$).*

Proof Let us suppose by contradiction that the set V returned by a get()operation op is not an admissible set. The set V is calculated in lines 09-15, Figure 19. Due to Lemma 16, such operations represents a k-cut permutation induced by op on $\widehat{U}$ and let's call π_i such permutation.
If V is not an admissible set for op, it means that it is not generated by any $\pi_i \in \Pi_{k,\widehat{U}(op)}$ and then, one of the following conditions holds:

1. there exists an element v, generated by every permutation π_i of the permutation set $\Pi_{\widehat{U}}$, such that v does not belong to set_i (i.e., $\exists\ v : \forall\ \pi_i \in \Pi_{k,\widehat{U}(op)}\ (\exists\ \mathsf{add}(v) \in\ \pi_i : \mathsf{add}(v) \rightarrow_{\pi_i} op) \wedge (\nexists\ \mathsf{remove}(v) \in \pi_i : \mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op) \wedge (v \notin set_i)$);
2. there exists an element v belonging to set_i such that it can not be generated by any permutation π_i of the permutation set $\Pi_{\widehat{U}}$ (i.e., $\exists\ v \in set_i : \forall \pi_i \in \Pi_{k,\widehat{U}(op)}\ (\exists\ \mathsf{add}(v), \mathsf{remove}(v) \in\ \pi_i$: $\mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v) \rightarrow_{\pi_i} op)$).

Case 1. If there exists an $\mathsf{add}(v)$ in the permutation π_i it means that in the $last\_ops_i$ variable there exists a tuple $< A, v, -, - >$ and then p_i will execute line 12 of Figure 19. Since the $last\_ops_i$ variable is ordered according to the partial order of the execution and since there not exists any tuple $< R, v, -, - >$ in the $last\_ops_i$ ordered variable after $< A, v, -, - >$ it follows that after executing line 12 of Figure 19, p_i will not execute 13 of Figure 19 in the following and then v will be in the set_i variable returned having a contradiction.

Case 2. At line 09 of Figure 19, the variable set_i 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 19 and after it has not executed line 13

of Figure 19. If p_i has executed line 12 and after it has not executed line 13 it means that in the $last\_ops_i$ variable there exists a tuple $< A, v, -, - >$ and there is not any tuple $< R, v, -, - >$ coming after. Due to Lemma 16 all the operations in the $last\_ops_i$ variable represent a k-cut permutation consistent with $\widehat{U}$ and then it follows that there exists an $\mathsf{add}(v) \in \pi_i$ and there is not any $\mathsf{remove}(v) \in \pi_i : \mathsf{add}(v) \rightarrow_{\pi_i} \mathsf{remove}(v)$ and we have a contradiction.

$\square_{Theorem\ 5}$

Theorem 6. *Let $\mathcal{S}$ be a k-bounded set object and let $\widehat{H} = (H, \prec)$ be an execution history of $\mathcal{S}$ generated by the algorithms in Figures 18 - 19. If every active process p_i maintains an admissible k-bounded set, $\widehat{H}$ is always per-element sequential consistent.*

Proof The proof follows immediately by Lemma 14. Since update operations are ordered according to the pair (sn, id), each process executing line 11 of Figure 20 will order all the operations in the same way. $\square_{Theorem\ 6}$

6.2.5. Programming with k-bounded set objects: Eventual Participant Detector

A k-bounded set can be easily used to implement an oracle that returns the list of processes currently part of a group. Such an oracle is called *participant detector*. The notion of participant detector is close to the concept of failure detector and it has been considered in [6, 10] to discover processes currently in the network and to solve the consensus with unknown participants problem.

Specification and Protocol. Due to the eventual synchrony of our model, in this paper we consider an oracle, called *eventual participant detector* that can make mistakes during asynchrony periods. Given a group computation, an eventual participant detector can be characterized by two properties:

- **Eventual Completeness:** Eventually, every process that leaves permanently the group is no more returned by the oracle.
- **Eventual 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. 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 id in the repository by invoking the $\mathsf{add}(id)$ operation on the k-bounded set and repeats periodically such operation. Repeating periodically the $\mathsf{add}(id)$ operation is actually needed because a k-bounded set object keeps memory only of recent update operations.

When a process leaves the group, it invokes the $\mathsf{remove}(id)$ operation and stops to add its identifier in the repository.

When a process wants to have the current membership of the group it executes the $\mathsf{get}()$operation. The pseudo-code of this simple procedure is shown in Figure 22.

Note that, the value k depends both on the size of the group and on the latency of the operations

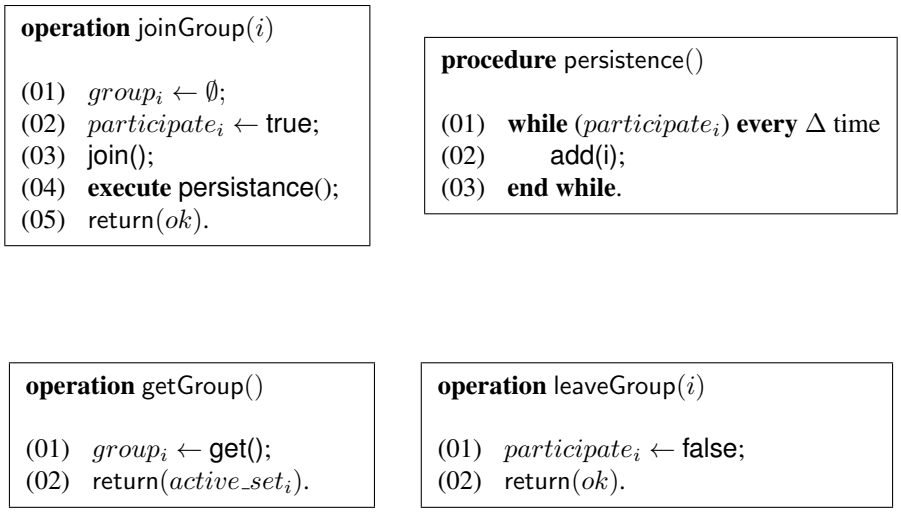


Figure 22: The Eventual Participant Detector Implementation

invoked on the k-bounded set; the higher the value of these parameters are, the higher k must before the implementation to be correct.

Correctness Proofs. In the following, we will show how the algorithm shown in Figure 22 is able to implement an eventual participant detector in an eventually synchronous system where the value of k is appropriately selected. The hard part of the problem is to satisfy the eventual accuracy property. In fact, during asynchrony periods it is not possible to define deterministic bounds on the operations execution times. As a consequence, a fast process p_i could fill in all the k slots of the object with its add(i) operations, while a slow process p_j could be infinitely often overwritten by the fast one. Therefore, it is possible that p_j is never returned by the oracle. However, when the synchrony period arrives, it is possible to compute an upper bound Δt_{op} on the execution time of each k-bounded set operation. Thus, starting from the number of processes N and from the various Δt_{op}, it is possible to compute the minimum value of k that verifies eventual accuracy. Let us remark that if k is not appropriately configured, some process p_j could never be returned by the oracle due to random overwriting of the k-bounded set structure.

Lemma 17. Eventual Completeness. *Eventually, every process that leaves permanently the group is no more returned by the oracle.*

Proof The proof trivially follows by considering that a process p_i leaving the group, stops to add its identifier in the k-bounded set. Due to the continuous churn, every joining process will add its identifier and considering that k is finite, eventually the last add(i) operation executed from p_i will disappear from the object. $\square_{Lemma\ 17}$

Lemma 18. Eventual Accuracy. *Let Δt_A, Δt_R, Δt_G and Δt_J be the upper bounds on the execution time taken respectively by an* add(v) remove(v), get$()$ *and* join$()$ *operations. Let N_{PD} be the*

maximum size of the group and let c_{PD} be the churn rate. If $k > N_{PD} + \big((\Delta + 2\Delta t_A) \times 2cN_{PD}\big)$, then eventually, each process that remains forever in the group is always returned by the oracle.

Proof At the beginning of the computation, at most N_{PD} processes belongs to the group (i.e., $k \geq N_{PD}$). As soon as the churn starts, processes start to join the eventually participant detector executing the joinGroup() operation. In particular, a joinGroup() operation, invoked by a process p_i, triggers first a join() on the k-bounded set and then it activates a periodic invocation of add(i) operations. Let us suppose by contradiction that there exists a process p_i remaining forever in the computation and its identifier is not returned by a get()operation. Let us consider the time t when the system becomes synchronous. After time t, the execution time for each operation executed on the k-bounded set is finite and bounded. Let us consider the time interval I between the beginning of an add(i) and the termination of the next add(i) (i.e., $I = [t, t+\Delta+2\Delta t_A]$). If p_i does not belong to the set returned by a get()operation, it means that between two following add(i) operation, there exist too many operations and i disappear from the k-bounded set. Due to the churn rate, given the interval I, there exist at most $\big((\Delta + 2\Delta t_A) \times 2cN_{PD}\big)$ operations (add()/remove()) that can be issued. Considering that $k > N_{PD} + \big((\Delta + 2\Delta t_A) \times 2cN_{PD}\big)$, and considering that between two following add(i) operations at most $\big((\Delta + 2\Delta t_A) \times 2cN_{PD}\big)$ operations can take place, it means that p_i does not disappear from the k-bounded set and we have a contradiction. $\square_{Lemma\ 18}$

7. Conclusion

Shared objects provide programmers with a powerful tools to design distributed applications on top of complex distributed systems. This paper has investigated under which assumptions a set object can be implemented using finite (and bounded) memory on top of a dynamic distributed system prone to continuous churn.

The paper has presented a consistency condition suited to a set object which is weaker than sequential consistency by exploiting the semantic of the set object and allowing, at the same time, concurrent readings to return the same set in absence of other operations. The paper has also shown the impossibility of defining a protocol implementing a set object using a finite memory on top of a non-synchronous distributed system prone to continuous churn. To overcome the impossibility we worked along two orthogonal directions: strengthening the system model and weakening the object specification. Thus, we firstly presented a protocol working in a synchronous system prone to continuous churn along with its correctness proof. Secondly, along the line of weakening the object specification, we introduced a weaker form of set, namely k-bounded set, that "approximates" a set as k tends to infinity (i.e., it behaves as a set when $k = \infty$). We presented a distributed protocol implementing a k-bounded set that requires a majority of active processes to be up at the same time.

We strongly believe that this paper makes a first step towards the definition of generic abstract data types (e.g., maps, list, queues etc.) for dynamic distributed systems as (i) set object can be used itself as basic building blocks and (ii) it highlighted some constraints on the design of distributed protocols in message passing model, following by the combination of eventual synchrony and continuous churn that are common to any abstract data type implementation.

References

[1] M.K. Aguilera. A pleasant stroll through the land of infinitely many creatures. *ACM SIGACT News*, 35(2):36–59, 2004.

[2] M.K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *Journal of the ACM (JACM)*, 58(2):7, 2011.

[3] M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, and P.W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[4] R. Baldoni, S. Bonomi, A.M. Kermarrec, and M. Raynal. Implementing a register in a dynamic distributed system. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 639–647. IEEE, 2008.

[5] R. Baldoni, S. Bonomi, and M. Raynal. Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. *IEEE Transactions on Parallel and Distributed Systems*, 23(1):102–109, January 2012.

[6] D. Cavin, Y. Sasson, and A. Schiper. Consensus with unknown participants or fundamental self-organization. *Ad-Hoc, Mobile, and Wireless Networks*, pages 630–630, 2004.

[7] C. Delporte-Gallet and H. Fauconnier. Two consensus algorithms with atomic registers and failure detector ω. In *Proceedings of the 10th International Conference on Distributed Computing and Networking*, ICDCN '09, pages 251–262, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] R. Friedman, M. Raynal, and C. Travers. Two abstractions for implementing atomic objects in dynamic systems. In *Proceedings of the 9th international conference on Principles of Distributed Systems*, OPODIS'05, pages 73–87, Berlin, Heidelberg, 2006. Springer-Verlag.

[9] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[10] F. Greve and S. Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *Dependable Systems and Networks (DSN), 2007 IEEE/IFIP International Conference on*. IEEE Computer Society, 2007.

[11] R. Guerraoui, R. R. Levy, B. Pochon, and J. Pugh. The collective memory of amnesic processes. *ACM Transactions on Algorithms*, 4(1):12:1–12:31, March 2008.

[12] V. Hadzilacos and S. Toueg. Reliable broadcast and related problems. *Distributed Systems*, pages 97–145, 1993.

[13] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[14] S.Y. Ko, I. Hoque, and I. Gupta. Using tractable and realistic churn models to analyze quiescence behavior of distributed protocols. In *Reliable Distributed Systems, 2008. SRDS'08. IEEE Symposium on*, pages 259–268. IEEE, 2008.

[15] M. J. Kosa. Time bounds for strong and hybrid consistency for arbitrary abstract data types. *Chicago Journal of Theoretical Computer Science*, 1999.

[16] F. Kuhn, S. Schmid, J. Smit, and R. Wattenhofer. A blueprint for constructing peer-to-peer systems robust to dynamic worst-case joins and leaves. In *Quality of Service, 2006. IWQoS 2006. 14th IEEE International Workshop on*, pages 12–19, June 2006.

[17] F. Kuhn, S. Schmid, and R. Wattenhofer. A self-repairing peer-to-peer system resilient to dynamic adversarial churn. In *Proceedings of the 4th International Conference on Peer-to-Peer Systems*, IPTPS'05, pages 13–23, Berlin, Heidelberg, 2005. Springer-Verlag.

[18] D. Leonard, V. Rai, and D. Loguinov. On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):26–37, 2005.

[19] Z. Li and M. Parashar. Comet: A scalable coordination space for decentralized distributed environments. In *Proceedings of the Second International Workshop on Hot Topics in Peer-to-Peer Systems*, HOT-P2P '05, pages 104–112, Washington, DC, USA, 2005. IEEE Computer Society.

[20] N. Lynch and A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, pages 173–190, 2002.

[21] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[22] S. W. McLaughry and P. Wycko. T spaces: The next wave. In *Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences-Volume 8 - Volume 8*, HICSS '99, pages 8037–, Washington, DC, USA, 1999. IEEE Computer Society.

[23] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. *Distributed Computing*, pages 164–178, 2000.

[24] A. Murphy, G. Picco, and G. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, ICDCS '01, Washington, DC, USA, 2001. IEEE Computer Society.

[25] H. Roh, M. Jeon, Jin-Soo Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distributed Computing*, 71(3):354–368, March 2011.

[26] A. Schiper and M. Raynal. A suite of formal definitions for consistency criteria in distributed shared memories. In *ISCA Proceedings of the International Conference PDCS, Dijon France*, pages 125–130, 1996.

[27] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[28] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[30] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.

[31] S. Tucci Piergiovanni and R. Baldoni. Connectivity in eventually quiescent dynamic distributed systems. In *Third Latin-American Symposium on Dependable Computing (LADC)*, pages 38–56. Springer, 2007.

[32] S. Voulgaris, D. Gavidia, and M. Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[33] G. T.J. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.

[34] J. Zhou, C. Zhang, H. Tang, J. Wu, and T. Yang. Programming support and adaptive checkpointing for high-throughput data services with log-based recovery. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 91–100. IEEE, 2010.