# k-bounded Set Objects in Eventually Synchronous Distributed Systems with Churn and Continuous Accesses

Roberto Baldoni, Silvia Bonomi
Sapienza Università di Roma
Via Ariosto 25
I-00185 Roma, Italy
{baldoni, bonomi}@dis.uniroma1.it

Michel Raynal
IRISA, Univeristé de Rennes 1
Campus de Beaulieu
F-35042 Rennes Cedex, France
raynal@irisa.fr

## ABSTRACT

This paper introduces a "k-bounded set object", namely a shared object with limited memory that allows processes to add and remove values as well as take a snapshot of its content. The interest of the k-bounded set lies in the fact that it can be used to program useful abstractions for dynamic distributed systems, such as an eventual participant detector.

## Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: System and Software—*distributed systems*; D.4.2 [**Operating Systems**]: Storage Management—*distributed memories*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*

## General Terms

Reliability, Theory

## Keywords

$k$-bounded set object, Continuous accesses, Churn, Dynamic distributed system, Infinite arrival model, Eventually Synchronous system.

## 1. INTRODUCTION

One of the main aspects of à-la-google distributed computing is to help non-stop processing of data objects, possibly residing on a large number of machines, deployed on top of a message-passing distributed system (e.g. datacenters). These data are continuously and frequently updated and retrieved by hundreds or thousands of users (either processes or software modules) for applicative, profiling, mining or accounting purposes. The traffic accessing such data objects is continuous and may reach hundreds of thousands of requests per second (data object accesses are non-quiescent). Additionally, these objects are implemented in a setting with an high degree of self-management due to the huge number of software and hardware components. As a consequence, data objects must tolerate continuous arrivals and departures of processes due to failures, maintenance procedures, etc. Said differently, the object implementation must resist to continuous churn (churn is non-quiescent). Hence, there is the need to define suitable data structures that support concurrent accesses guaranteeing, in a deterministic way, a degree of consistency which is adequate to solve a given problem and whose implementation is both done using finite memory and working in the presence of continuous churn and without assumption on a bound on the number of accesses. An example of such objects is the set object [3]. A set object is a shared object storing a (possibly empty) finite set of values. A process can acquire the content of the set object through a *get* operation while it can add (remove) an element to the set object through an *add* (*remove*) operation. It has been shown in [4] that the implementation of such set object requires infinite memory in non synchronous settings when the set object is under continuous both churn and accesses. This impossibility result comes from the following simple observation: considering the domain of the values stored in the set arbitrary but finite, there is the need to store all the execution history[1] in order to get the precise state of the set during a get operation [4].

This paper thus introduces a weaker form of set called *$k$-bounded set object*, which is implementable in an eventually synchronous distributed system with churn, continuous accesses and finite memory. Informally, a $k$-bounded set object is a set that has limited memory of the execution history. In particular, given a get()operation, a $k$-bounded set behaves as a set where the execution history is limited only to the $k$ most recent update operations. In this way, $k$ can be used as a parameter to trade in-memory availability on the node, used to implement the object, with the need to approximate precisely a set object. We show additionally how a $k$-bounded set can be used to implement interesting abstractions such as *participant detectors* in dynamic distributed systems [5].

**Roadmap.** After discussing the related work in Section 2, Section 3 describes the system model. Section 4 introduces the $k$-bounded set abstraction and then describes an implementation. At the end of this section, how to use a $k$-bounded set to build an eventual participant detector is shown. Section 5 concludes the paper.

## 2. RELATED WORK

**Regular Registers in Dynamic Distributed Systems.** In [12], the authors deal with the implementation of an atomic R/W object where the churn is abstracted by the notion of system reconfiguration. A reconfiguration could

[1] Given a get()operation, its result depends from all the operations invoked in its "past".

Copyright is held by the author/owner(s).
ACM X-XXXXX-XX-X/XX/XX.

happen every time there is a process joining or leaving the system. To be valid a reconfiguration requires processes to agree upon an unique sequence of configuration changes. This agreement implies the need of consensus and thus cannot be implemented in a fully asynchronous system. In [1] Aguilera et al. show that an atomic R/W object can be realized without consensus and, thus, on a fully asynchronous distributed system provided that the number of reconfiguration operations is finite and thus the churn is quiescent [2]. Finally in [2] 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.** In [2], we implemented a regular register in an eventually synchronous distributed systems with continuous churn while we showed in [4] 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, such a dimension does not play any role because the register does not have to store anything else that the last written value. This is not true for a set that has to be able to reconstruct all the update operation history. If the update operations are continuous, this implies that the set implementation needs infinite memory.

Let us note that a weaker notion of set, namely weak set, has been introduced by Delporte and Fouconnier in [7]. A weak set is an object representing a restricted form of set that does not include remove operations and that can be implemented, under some specific assumptions on the number of processes and the domain of the values, by using a *finite number of atomic registers*.

# 3. THE SYSTEM MODEL

**Distributed System.** The distributed system is composed, at each time, by a bounded number of processes that communicate by exchanging messages through point-to-point channels or by means of a broadcast primitive. The passage of time in the distributed system is measured through a global fictional clock (whose domain is the set of integers) not accessible from processes. Processes are uniquely identified (with their indexes), has finite memory space for local computation, and they may join and leave the system at any point in time. The processing times of local computations are negligible with respect to communication delays, so they are assumed to be equal to 0. Contrarily, messages take time to travel to their destination processes. The distributed system is *eventually synchronous*, that is after an unknown but finite time, it behaves synchronously [6, 8]. Similarly to [9], the communication primitives are characterized by the following property:

Eventual timely delivery: there exists an integer $\delta$ and a a time $t$ such that any message sent (broadcast) at some time $t' \geq t$, is received (delivered) by time $t' + \delta$ to the processes that are in the system during the interval $[t', t' + \delta]$.

It is important to remark that the time $t$ exists but it is not known by processes.

**Distributed Computation.** A distributed computation is composed, at each time, by a subset of processes of the distributed system. A process $p$, belonging to the system, that wants to participate to the distributed computation has to execute a join() operation. Such an operation, invoked at some time $t$, is not instantaneous: it consumes time. But, from time $t$, the process $p$ can receive and process messages related to the computation.

A process leaves the computation in an implicit way. When it does, it leaves the computation forever and does no send computation messages anymore. From a practical point of view, if a process wants to re-enter the computation, it has to enter it as a new process (i.e., with a new name). We assume that a process does not crash during the distributed computation (i.e. it does not crash from the time it joins the system until it leaves)[3]. The set of processes that actively participate to the computation is defined as follows.

DEFINITION 1. *A process is* active *from the time it returns from the* join() *operation until the time it leaves the system.* $A(t)$ *denotes the set of processes that are active at time* $t$*, while* $A([t_1, t_2])$ *denotes the set of processes* $p$ *such that* $p \in A(\tau)$ *for each time* $\tau$ *in the interval* $[t_1, t_2]$.

**Continuous Churn Model.** The dynamicity of the joins and leaves of the processes is captured by the system parameter called *churn*. We consider here the *churn rate*, denoted $c$, defined as the percentage of the nodes that are "refreshed" at every time unit ($c \in [0, 1]$). This means that, while the number of processes remains constant (equal to $n$), in every time unit $c \times n$ processes leave the system and the same number of processes join the system (where $n$ is the number of processes inside the distributed computation). It is shown in [11] that this assumption is fairly realistic for several classes of applications built on top of dynamic systems.

**Continuous accesses.** In our model, active processes do not stop invoking operations on the shared object (accesses to the object are non-quiescent). This implies that the number of operations issued on the shared object during an execution cannot be bounded.

**The set object S.** A set object $\mathcal{S}$ is an object shared among processes, used to store values. Without loss of generality, we assume that (i) $\mathcal{S}$ contains only values taken from an arbitrary finite domain (i.e. a subset of integers) and (ii) at the beginning of the computation $\mathcal{S}$ is empty.

A set object $\mathcal{S}$ can be accessed from processes through three operations: add(), remove() and get();

• the *add* operation, denoted add($v$), takes an input parameter $v$ and returns a confirmation that the operation has been executed. 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 $v$ and returns a confirmation that the operation has been executed. 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 the current content of $\mathcal{S}$ (i.e., all the values added and not removed) without modifying its content.

The formal specification and a discussion on consistency criterium for the set object can be found in [3].

**Impossibility Result.** In [4], it has been shown that it is not possible to define an automata for the set object that uses finite memory if (i) the accesses to the set are con-

[2]This assumption has been also employed in [13] and in [12].

[3]Actually in the proof we just need processes do not fail during the execution of add and remove operations.

tinuous, (ii) the churn is continuous and (iii) the system is eventually synchronous.

# 4. $K$-BOUNDED SET OBJECT

Due to previous impossibility result, the specification of the set object is weakened and the notion of *$k$-bounded set object* is introduced. Informally, a $k$-bounded set object is a set that has limited memory of the execution history. In this section, we first define formally the $k$-bounded set object, then we provide an algorithm for an eventually synchronous system and we show an interesting application of the $k$-bounded set: an oracle called *eventual participant detector*.

## 4.1 $k$-bounded set specification

Informally, given a get()operation, a $k$-bounded set behaves as a set where the execution is limited only to the $k$ most recent update operations[4] defining thus a "window" on the execution history.

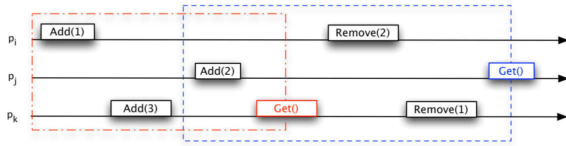



**Figure 1: Examples of 3-bounded set wrt two get()operations.**

As an example consider the execution shown in Figure 1 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 (operations out of the window appear as never invoked); the set returned by the the get()invoked by $p_k$ is $\{1, 2, 3\}$ (as the one that would be returned by the same get()invoked on a set) while the set returned by the get()invoked by $p_j$ 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 get()).
More formally, we can say that a protocol implements a $k$-bounded set object if

- **Termination:** each operation invoked on the $k$-bounded set eventually terminates;
- **Validity:** each get()operation invoked on a $k$-bounded set returns an admissible set.

In the following we provide a set of definitions to formally define when the set returned from a get()operation is an admissible set.

**Basic Definitions.** Every operation issued on the object is not instantaneous and it can be characterized by two events occurring at its boundary: an *invocation* event and a *reply* event. These events occur at two time instants (invocation time and reply time). According to these time instants, it is possible to state when two operations are concurrent with respect to the real time execution.
Given two operations $op$ and $op'$, and their invocation times $t_B(op)$ and $t_B(op')$ and return times $t_E(op)$ and $t_E(op')$, we say that *$op$ precedes $op'$* ($op \prec op'$) iff $t_E(op) < t_B(op')$. If $op$ does not precede $op'$ and $op'$ does not precede $op$ then they are *concurrent* ($op||op'$). Considering all the operations invoked on the object and the previous precedence relation, 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:

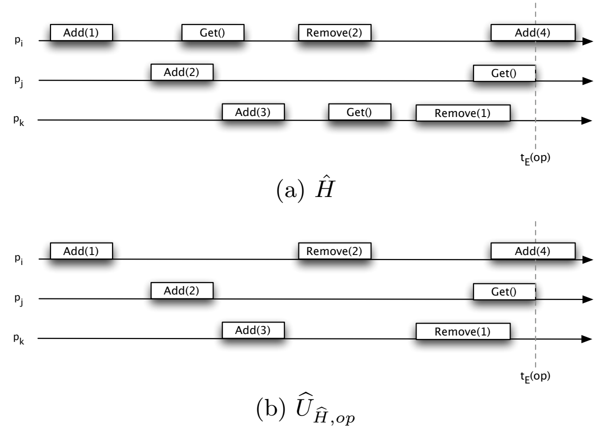
[4]With the term *update operation* we denote both an add() or a remove() operation.

Definition 2 (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 $\prec$.*

As a get()operation does not modify the content of a $k$-bounded set, in the following, the admissible values for a get()operation will be defined by considering the sub-history of the execution containing only the operations that update the object (i.e. add() and remove() operations) and the considered get(). To this aim, let us introduce the concept of *update sub-history induced by a* get() *operation.*

Definition 3 (Update sub-history). *Let $\widehat{H} = (H, \prec)$ be the execution history of a $k$-bounded set object. The update sub-history of $\widehat{H}$ induced by a* get()*operation $op$, denoted as $\widehat{U}_{\widehat{H},op} = (U, \prec)$, is defined as follows:*
- *$U = \{o \in H | (o = \mathsf{add}(v) \ \vee \ o = \mathsf{remove}(v)) \wedge t_B(o) < t_E(op)\} \cup \{op\}$;*
- *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 $\widehat{U}_{\widehat{H},op}$.*

Note that, due to continuous accesses to the $k$-bounded set object, the execution history can contain an unbounded number of operations. However, given a get()operation $op$, the update sub-history induced by $op$ is always composed by a finite number of operations. As an example, Figure 2(a) shows the execution history $\hat{H}$ of a $k$-bounded set and Figure 2(b) shows the update sub-history induced by the




(a) $\hat{H}$

(b) $\hat{U}_{\hat{H},op}$

**Figure 2: Execution history and update sub-history of a $k$-bounded set object.**

get()operation *op* issued by $p_j$.
In the following, when it is clear from the context which are the execution history $\widehat{H}$ and the operation *op* the updated sub-history refers to, the notation $\widehat{U}$ is used instead of $\widehat{U}_{\widehat{H},op}$.
Let us remark that an update sub-history (and in general each finite execution history) is a partial order of operations. Therefore, several linearizations of the operations, differing one from the other for the order given to concurrent operations, can be defined. As an example, consider the update sub-history induced by the get()operation *op* invoked from $p_j$ shown in Figure 2(b); there exist two different linearizations of the operations, namely $l_1 =$ add$(1)i$, add$(2)j$, add$(3)k$, remove$(2)i$, remove$(1)k$, add$(4)i$, get$()j$[5] and $l_2 =$ add$(1)i$, add$(2)j$, add$(3)k$, remove$(2)i$, add$(4)i$, remove$(1)k$, get$()j$, producing the same set.
In the following we introduce the concepts of *consistent permutation* to identify each of these linearizations, *set generated by a permutation* to define the set of values resulting from the execution of such operation order and *permutation set* to identify all the consistent permutations following from a given finite execution history (or sub-history).

Definition 4 (Permutation $\pi$ Consistent with $\widehat{H}$ ). *Given a finite execution history $\widehat{H} = (H, \prec)$, a* permutation $\pi$ *of all the operations belonging to H is* consistent *with $\widehat{H}$ if, for any pair of operations $op, op'$ in $\pi$, op precedes $op'$ in $\pi$ whenever op precedes $op'$ in $\widehat{H}$.*

Definition 5 (Set Generated by a Permutation). *Let $\widehat{H} = (H, \prec)$ be an execution history and let op be a* get()*operation of H. Given the update sub-history $\widehat{U}_{\widehat{H},op} = (U, \prec)$ induced by op on $\widehat{H}$, let $\pi$ be a permutation consistent with $\widehat{U}_{\widehat{H},op}$, then the set of values V generated by $\pi$ for op contains all the values v such that:*
- $\exists$ add*(v)* $\in \pi$ : add*(v)* $\prec$ *op and*
- $\nexists$ remove*(v)* $\in \pi$ : add*(v)* $\prec$ remove*(v)* $\prec$ *op.*

As an example, consider the get()operation *op* issued by $p_j$ in the execution history $\hat{H}$ and the corresponding update sub-history $\hat{U}$ shown in Figure 2. One of the possible permutations consistent with $\hat{U}$ is $\pi =$ (add$(1)i$, add$(2)j$, add$(3)k$, remove$(2)i$, remove$(1)k$, get$()j$, add$(4)i$)[6] and the corresponding set of values $V$ generated by $\pi_1$ is $V = \{3\}$.

Definition 6 (Permutation set). *Let $\widehat{H} = (H, \prec)$ be the a finite execution history of the shared object. A* permutation set *of $\widehat{H}$, denoted as $\Pi_{\widehat{H}}$, is the set of all the permutations $\pi$ that are consistent with $\widehat{H}$.*

As an example, consider the update sub-history $\widehat{U}$ shown in Figure 2. The permutation set $\Pi_{\widehat{U}}$ of the update sub-history $\widehat{U}$ is shown in Figure 3(a).
Note that, the $k$-bounded set cares about the most $k$ recent operations and thus the concepts of permutation and $k$-permutation set are specified to the notions of *k-cut permutation* and *k-cut permutation set induced by a* get()*operation.*

[5]Let us recall that an update sub-history is a finite history of a given execution.
[6]With the notation *op()id* it is represented the operation *op* issued by the process with identifier *id*

$\Pi_{\hat{H}} = \{$
$\pi_1 =$ (add$(1)i$, add$(2)j$, add$(3)k$, remove$(2)i$, remove$(1)k$, get$()j$, add$(4)i$)
$\pi_2 =$ (add$(1)i$, add$(2)j$, add$(3)k$, remove$(2)i$, remove$(1)k$, add$(4)i$, get$()j$)
$\pi_3 =$ (add$(1)i$, add$(2)j$, add$(3)k$, remove$(2)i$, get$()j$, add$(4)i$, remove$(1)k$)
$\pi_4 =$ (add$(1)i$, add$(2)j$, add$(3)k$, remove$(2)i$, get$()j$, remove$(1)k$, add$(4)i$)
$\pi_5 =$ (add$(1)i$, add$(2)j$, add$(3)k$, remove$(2)i$, add$(4)i$, get$()j$, remove$(1)k$)
$\pi_6 =$ (add$(1)i$, add$(2)j$, add$(3)k$, remove$(2)i$, add$(4)i$, remove$(1)k$, get$()j$)
$\}$

(a) Permutation Set

$\Pi_{3,\hat{H}}(op) = \{$
$\pi_1 =$ (add$(3)k$, remove$(2)i$, remove$(1)k$, get$()j$)
$\pi_2 =$ (remove$(2)i$, remove$(1)k$, add$(4)i$), get$()j$)
$\pi_3 =$ (add$(2)j$, add$(3)k$, remove$(2)i$, get$()j$)
$\pi_4 =$ (add$(3)k$, remove$(2)i$, add$(4)i$, get$()j$)
$\pi_5 =$ (remove$(2)i$, add$(4)i$, remove$(1)k$), get$()j$)
$\}$

(b) 3-Permutation Set

**Figure 3: Permutation Set and 3-permutation set of the History $\widehat{U}$ of Figure 2.**

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 $\Pi_{\widehat{U}}$ of the update sub-history by considering for each permutation its $k$-cut.

Definition 7 ($k$-cut permutation induced by $op_i$). *Given a finite 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 $\pi$ and an integer k, the* $k$-cut permutation *induced by $op_i$ on $\pi$, denoted as $\pi_k(op_i)$, is the sub-set of $\pi$ ending with $op_i$ and including the k operations that precede $op_i$ in $\pi$ (i.e. $\pi_k(op_i) = (op_{i-k}, \ldots, op_{i-1}, op_i)$).*

As an example, consider the update sub-history $\widehat{U}$ shown in Figure 2 and consider the permutation $\pi_1 =$ (add$(1)i$, add$(2)j$, add$(3)k$, remove$(2)i$, remove$(1)k$, get$()j$, add$(4)i$) consistent with $\widehat{U}$. If $op =$get$()j$ and $k = 3$, the 3-cut permutation induced by $op$ on $\pi_1$ is $\pi_{1_3}(op)$=(add$(3)k$, remove$(2)i$, remove$(1)k$, get$()j$).

Definition 8 ($k$-cut permutation set induced by $op_i$). *Given a finite 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 as $\Pi_{k,\widehat{H}}(op)$, is the set of all the k-cut permutations induced by op on each permutation $\pi$ of $\Pi_{\widehat{H}}$.*

Consider the update sub-history $\widehat{U}$ depicted in Figure 2 and the permutation set $\Pi_{\widehat{U}}$ of the history $\widehat{U}$ shown above. If $op =$get$()j$ and $k = 3$, the 3-cut permutation set induced by $op$ is shown in Figure 3(b).
We are now in the position to specify a validity condition for the set returned by any get()operation issued on a $k$-bounded set object.

Definition 9 (Admissible set for a get()operation). *Given an execution history $\widehat{H} = (H, \prec)$ of a k-bounded set object, let op be a* get()*operation of H. Let $\widehat{U}_{\widehat{H},op}$ be the*

*update sub-history induced by op on $\widehat{H}$ and let k be an integer. An admissible set for op, denoted as $V_{ad}(op)$, is any set generated by any permutation $\pi$ belonging to the k-cut permutation set of $\Pi_{k,\widehat{U}_{\widehat{H},op}}$.*

As an example, consider the execution history $\widehat{H}$ shown in Figure 2 and its update sub-history $\widehat{U}$ induced by the operation *op* =get()$j$. Given its 3-cut permutation set $\Pi_{3,\widehat{U}}$, all the possible admissible sets for *op* are $V_{ad\ 1} = \{3\}$, $V_{ad\ 2} = \{4\}$, $V_{ad\ 3} = \{3\}$, $V_{ad\ 4} = \{3, 4\}$, $V_{ad\ 5} = \{4\}$.

## 4.2 A distributed protocol implementing $k$ - bounded set

This section describes a protocol implementing a $k$ - bounded set object, based on message exchange.

Our algorithm assumes that (i) at each time unit, $\lceil n/2 \rceil$ active processes belongs to the distributed computation (where $n$ is the number of processes inside the distributed computation) and (ii) each process that invokes the join operation remains in the distributed computation for at least $3\delta$ time units from the invocation time.

Each process $p_i$ maintains locally a copy of the $k$-bounded set together with two sequence numbers: one is associated to the last get()operation performed and the other is associated to the last update operation it knows. Moreover, it also maintains the list of most recent update operations it has executed. Each operation *op* is characterized by a 4-uples $< type, val, sn, id >$ where $type = \{A \text{ or } R\}$[7] represents the operation type of the value $val$, with a sequence number $sn$, issued by a process with identity $id$. $p_i$ also maintains a state variable, namely $active_i$ initialized to false, that is switched to true as soon as $p_i$ has terminated the join operation, i.e. when $p_i$ becomes *active*.

To simplify the algorithm presentation, we introduce a procedure, namely UPDATE, that given a 4-uple representing an operation *op*, updates all the local variables. In particular, it is responsible for the management of the buffer where the last $k$ operations are stored. Additional details and the complete pseudo-code of the algorithm can be found in [4].

**The join()operation.** The join()algorithm involves all the processes present in the distributed computation (either they are active or not).

The basic idea of this algorithm is to gather information about recent operations from all the active processes belonging to the distributed computation. To this aim, a joining process $p_i$ broadcasts an INQUIRY message to inform processes part of the computation that it wishes joining the $k$-bounded set computation and thus wants to gather the history of most recent operations.

Each process $p_j$, delivering an INQUIRY message, behaves differently depending on its state. If $p_j$ is active, it answers to $p_i$ by sending back a REPLY message containing the list of recent update operations it stores. In addition, if $p_j$ is concurrently executing an operation, it sends one further message: a DL_PREV message in case of a get() operation, to prevent $p_j$ to be blocked, or an UPDATE() message in case of a modification of the set, to speed up the add/remove operation. On the contrary, if $p_j$ is not active, it postpones its reply until it becomes active and just sends a DL_PREV() message.

[7] $A$ and $R$ stay respectively for add() and remove().

Delivering a DL_PREV message from a process $p_j$, $p_i$ adds $p_j$ to the set of processes "reading" from the set, in order to remember that it has to send a reply to $p_j$ when it will become active.

Delivering a REPLY message, $p_i$ checks if such a messages is a reply to its inquiry and if it is so, it buffers the list of recent operations it contains. As soon as a majority of reply messages have been delivered, $p_i$ orders the sets of recent update operations collected. The sort function simply orders the operations using the lexicographic order on the pairs $(sn, id)$. Now, for each of these operations, $p_i$ execute the UPDATE procedure, obtaining a copy of the $k$-bounded set with the last $k$ recent operations. Then, $p_i$ becomes active, which means that it can answer the inquiries it has received from other processes. Moreover, $p_i$ sends a reply message also to the processes from which it has received a DL_PREV message, to prevents them from waiting forever. Finally, $p_i$ returns *ok* to indicate the end of the join() operation.

**The get()operation.** A get() operation is a simplified version of the join(). 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. So, $p_i$ broadcasts a get request and waits for a quorum. Then $p_i$ recomputes and order its history by making the union of all the partial history received so far and the operations received during the waiting period. Afterwards $p_i$ executes all the operations by invoking the UPDATE procedure and returns the set.
When a process $p_j$ receives a get request, if it is active, it answers $p_i$ by sending back a REPLY message containing its local variables, otherwise $p_i$ postpones its answer until it becomes active.

**The add($v$) and remove($v$) operations.** The structure of the two protocols is basically the same and in the following text the term *update* operation is used as synonym for both the operations.
When a process $p_i$ wants to execute an update, it first performs a get()operation, in order to obtain the most updated sequence number, updates the local variables and then it sends an UPDATE() message to perform the current update and waits until it receives an ack message from a majority of processes. Once $p_i$ is unblocked from the wait statement, it returns from the operation.

When an UPDATE($< type, val, sn_i, i >$) message is delivered to some process $p_j$, if it is not active, it puts the 4-uple corresponding to the current operation into a pending budder and will process the operation at the end of the join(), once it will be active, otherwise, $p_j$ executes the UPDATE procedure for the current operation and it sends back an ACK message for the operation.

When an ACK 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.

**Correctness Proof.**

Due to the lack of space, we provide here only the main Theorem. The correctness proofs can be found in [4].

THEOREM 1 (Termination). *Let $n$ be the number of processes belonging to the computation at time $t_0$. If (i) $|A(t)| > \lceil \frac{n}{2} \rceil$ $\forall t$ and (ii) each process that invokes a* join() *operation*

*does not leave the system for at least $3\delta$ time units then a process $p_i$ that invokes a* join(), get(), add() *or* remove() *operation and does not leave the system eventually terminates its operation.*

THEOREM 2 (Validity). *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 $|A(t)| > \lceil \frac{n}{2} \rceil$ $\forall t$, the set $V$ returned by $op$ is an admissible set (i.e. $V = V_{ad}(op)$).*

### 4.3 Programming with the $k$-bounded set object: 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 [5, 10] to discover processes currently in the network and to solve the CUP (consensus with unknown participants) problem.
Due to the eventual synchrony of our model, in this paper we consider an oracle, called *eventual participant detector* that can make mistakes during the 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 add($id$) operation on the $k$-bounded set and repeats periodically such operation. Repeating periodically the 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 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 get()operation.
In [4], we show an implementation of the eventual participant detector working in an eventually synchronous environment and we prove its correctness defining the minimum value of $k$ that is able to satisfy the oracle specification, i.e., to ensure both eventual completeness and eventual accuracy. In particular, we prove that the value $k$ must be greater than a certain threshold depending both on the churn, the size of the group and on the latency of the operations invoked on the $k$-bounded set.

## 5. CONCLUSION

Realizing a set object in the context of a non-synchronous setting with continuous accesses and churn requires infinite memory [4]. This motivated the introduction of a weaker form of set, namely $k$-bounded set, that approximates a set as $k$ tends to infinity and becomes a one-bit register when $k$ is equal to one. We described a distributed algorithm implementing a $k$-bounded set that requires a majority of active processes to be up at the same time. Let us finally remark that, despite continuous churn, the paper assumed that the size of the distributed system be constant. As a future work, we are planning to extend the churn model to consider distributed computation where the system size might vary in a given range.


## 6. ACKNOWLEDGEMENTS

This work is partially supported by the EU STREP project SM4All and by the EU project SOFIA.


## 7. REFERENCES


[1] Aguilera M. K., Keidar I., Malkhi D., Shraer A. Dynamic atomic storage without consensus *in proceedings of 28$^{th}$ Annual ACM Symposium on Principles of Distributed Computing 2009*, 17-25

[2] Baldoni R., Bonomi S., Kermarrec A.M., Raynal M., Implementing a Register in a Dynamic Distributed System. in *Proc. 29th IEEE Int'l Conference on Distributed Computing Systems*, IEEE Press, pp. 639-647, 2009.

[3] Baldoni R., Bonomi S., Raynal M., Value-based Sequential Consistency for Set Objects in Dynamic Distributed Systems *Proceedings of Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference.*

[4] Baldoni R., Bonomi S., Raynal M., Set Objects in Eventually Synchronous Distributed Systems with Churn and Continuous Accesses *MIDLAB Tec. Report 7/2010* http://www.dis.uniroma1.it/˜ midlab/articoli

[5] Cavin D., Sasson Y., Shiper A. Consensus with Unknown Participants or Fundamental Self-Organization *In Proceedings of the 3rd International Conference on AD-HOC Networks & Wireless (ADHOC-NOW)* 2004.

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

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

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

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

[10] Greve, F. and Tixeuil, S. Knowledge Connectivity vs. Synchrony Requirements for Fault-Tolerant Agreement in Unknown Networks *Proceeding of 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN):, pp. 82-91

[11] Ko S., Hoque I. and Gupta I., Using Tractable and Realistic Churn Models to Analyze Quiescence Behavior of Distributed Protocols. *Proc. 27th IEEE Int'l Symposium on Reliable Distributed Systems*, 2008.

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

[13] Tucci Piergiovanni S. and Baldoni R. Connectivity in Eventually Quiescent Dynamic Distributed Systems *Third Latin-American Symposium on Dependable Computing*, LNCS, 38-56, 2007.