# Asynchronous Group Membership with Oracles

Kal Lin and Vassos Hadzilacos

University of Toronto
kal@cs.utoronto.ca
vassos@cs.utoronto.ca

**Abstract.** We present a model of distributed systems intended for the description of group membership services. The model incorporates a generalization of failure detectors [9], which we call *oracles*. Oracles provide information about processes that may be included into or excluded from the group. Based on this model, we provide a specification of a group membership service in asynchronous systems augmented with oracles. We also present an algorithm that implements such a service provided that the information supplied by the oracles is of sufficient quality.

## 1 Introduction

Several researchers have argued that the design and implementation of complex distributed systems, especially those for which fault-tolerance and high availability are primary goals, is facilitated by the use of "process groups" (for an overview see [16]). Speaking very informally, a process group consists of a collection of processes that cooperate in carrying out a common task — such as maintaining a database, supporting users who are collaborating in a complex design, or carrying out a massively parallel computation. In such applications, processes benefit from multicast capabilities, so that they can efficiently disseminate information about their own state to the entire group.

For multicasts to be meaningful, processes must have a fairly well coordinated view of the group's membership. One complicating factor is that the membership of the group changes over time. This is certainly due to failures and recoveries; it may also be due to other, application-specific, reasons: Subscribers to a database change over time, the participants in the complex design change as the design progresses through different phases, and nodes whose spare cycles are used for a massively parallel computation may drop in or out of the task as their load (and consequently the availability of spare cycles) changes. These requirements call for some system-level facilities that allow the (ever-changing) members of the group to have consistent views of the (ever-changing) membership. A second complicating factor is that often the maintenance of a consistent view of the group's membership must be accomplished in *asynchronous* distributed systems; that is, systems where the speeds of (non-faulty) processes and the delays of messages are finite but unbounded.

Many systems have been developed that provide a group membership service. The first of these systems is the Isis toolkit [6]. Others include the Highly

Available System [10], Amoeba [15], Transis [2], Horus [19], Relacs [5], Newtop [12], and Totem [3]. Group membership services can be classified in one of two categories: *primary partition* and *partitionable*. A primary-partition group membership service ensures that at each (logical) time there is only one view of the group, by restricting changes in the membership to only one partition.[1] A partitionable group membership service allows multiple independent views to coexist at the same (logical) time in different partitions. In this paper, we focus exclusively on primary-partition services.

Despite the existence of many systems that offer a group membership service, there is no satisfactory rigorous and abstract specification for that service in the context of *asynchronous* systems that encompasses both *safety* and *liveness* requirements. Cristian [11] gave such a specification for *synchronous* systems. Ricciardi and Birman [18] gave a specification for asynchronous systems but this specification contains serious flaws, as pointed out by Enceaume et al. [4]. In particular, the liveness property of the specification is too strong, and it is not at all obvious how to weaken it so as to ensure that the resulting property is both useful to applications using the service and implementable. De Prisco et al. [17] gave a rigorous I/O automata-based specification for group membership in asynchronous systems but, as the authors expressly acknowledge, their specification encompasses only safety properties.

Guerraoui and Schiper [14] propose a solution to Group Membership that has similarities to our approach. They advocate using Consensus to solve a variety of agreement problems such as Group Membership — but also Nonblocking Atomic Commitment, View Synchronous Communication and Atomic Multicast — in the context of asynchronous systems with failure detectors. The scope of their paper, therefore, is wider than ours, which addresses only Group Membership. Their specification of Group Membership, however, is incomplete.

The difficulty of formulating a useful, yet implementable, specification for (primary-partition) group membership services in asynchronous systems is, perhaps, not surprising if the matter is considered in the light of the well-known impossibility result by Fischer, Lynch and Paterson [13]. This result states that in asynchronous systems where (even) one process can crash, any algorithm that satisfies the safety property that correct processes should never reach inconsistent decisions, must violate the liveness property that each correct process should eventually reach a decision. It is true that there are differences between the sort of agreement that must be reached in the Consensus problem (to which this impossibility result applies), and the sort of agreement that must be achieved by (primary partition) group membership services. Chandra et al., however, showed that these differences do not render the Fischer, Lynch and Paterson result inapplicable: (even) a very weak type of group membership service is susceptible to a similar impossibility result [8].

In view of this, it is reasonable to try resolving the problem of formulating a useful and implementable specification of group membership by bringing

[1] Roughly speaking, a partition is a maximal set of processes that can communicate with one another.

to bear the approaches which allowed researchers to get around the Fischer-Lynch-Paterson impossibility result for Consensus. The use of unreliable failure detectors, pioneered by Chandra and Toueg [9], seems especially well-suited to the task: One of the reasons why the membership of a group changes is that processes fail. We can use information provided by (unreliable) failure detectors both to determine the membership of the group (this helps in formulating a useful specification) and to achieve agreement among the surviving members of the group (this helps in designing an implementation for the specification).

Considerable modifications to the original model of failure detectors are required, however. In part, these are due to the fact that the set of processes in the system is no longer fixed but changes over time. Furthermore, as we stated earlier, failures (and recoveries) are only *one* cause of changes in the membership of a group. We would like our model to be general enough that we can also describe other causes. A difficulty we must overcome here is that these causes are application-specific while we want our model to be application-independent.

In this paper, we introduce a model based on an abstraction we call *oracles*. Each process has access to an *include oracle* and to an *exclude oracle*, from which the process can obtain information about processes it should include into or exclude from the group. The information provided by oracles regarding which processes to include or exclude incorporates whatever (possibly imperfect) information the oracles have about failures in the system, but may also incorporate application-specific information. For example, in a database application a process might be included because its owner subscribed to the database; in a massively-parallel computation a process might be excluded because it runs on a node that suddenly became heavily loaded and has no spare cycles. We do not model the *reasons* for inclusion or exclusion — merely the fact. This makes the model application-independent. Oracles, and the model of computation, are described in Section 2.

Using this model, we present a rigorous and abstract specification for a group membership service in asynchronous systems. Our specification addresses both safety and liveness requirements. It is described in Section 3.2.

Finally, we present an algorithm to implement this specification. The algorithm is based on Chandra and Toueg's rotating coordinator algorithm [9], and is given in Section 4. Due to space limitations, the proof of correctness of the algorithm is omitted.

## 2 The Model

The system consists of an unbounded set of processes, $\Pi = \{p_1, p_2, p_3, \ldots\}$, connected by reliable links. The unbounded set of processes $\Pi$ represents the set of processes that *may* participate in the computation. In the course of any particular computation only *some* of these processes will actually participate. Informally speaking, we say that a process $p$ is *participating* if $p$ is executing steps of the protocol and some process (possibly $p$) wishes $p$ to join the computation[2].

To simplify the model, we assume the existence of a fictional global clock, whose output is the set of positive integers denoted by $\mathcal{T}$. The processes do not have access to this clock.

To model the set of processes that actually participate in a computation, and the duration of their participation, we define a participation pattern. Formally, a *participation pattern* is a function $PP : \mathcal{T} \rightarrow 2^{\Pi}$ where intuitively $PP(t)$ denotes the set of processes that are participating in the computation at time $t$. We restrict the participation pattern so that there is at most one (possibly infinite) interval of time during which a process is participating. More precisely, $\forall p \in \Pi,\ \forall t, t', t'' \in \mathcal{T},\ t < t' < t'',\ (p \in PP(t) \wedge p \notin PP(t')) \Rightarrow p \notin PP(t'')$.

We define the set of *persistently participating* processes in a participation pattern as $Persist(PP) = \{p : \exists t \in \mathcal{T},\ \forall t' \geq t,\ p \in PP(t')\}$ .

### 2.1 Oracles

Every process has access to an *exclude oracle* and an *include oracle*. Each process may query its oracles at any time to obtain a set of processes that the oracle wishes to exclude from or include into the group. An exclude oracle may wish to exclude a process because it believes (as a failure detector might) that the process is no longer participating in the protocol. It may also wish to exclude a process for application-specific reasons. Similarly, an include oracle may wish to include a process because it believes that the process has recovered, or because of application-specific reasons. We do not assume that oracles are "perfect"; they reflect possibly imperfect (and possibly contradictory) information available to the processes. Thus, the mere fact that a process $p$ is in some process $q$'s exclude (or include) oracle, does not necessarily mean that $p$ will be excluded from (or included into) the group. However, as it will become clearer when we present the liveness properties of our Group Membership specification in Section 3, if the information of oracles is persistent and non-contradictory, then the group's membership must conform to the information of the oracles.

Oracles — in particular exclude oracles — are intended as a generalization of failure detectors [9, 7]. It is tempting to define oracles in a similar manner. There are, however, some differences that complicate somewhat the definition. In particular, the set of processes in the system is no longer fixed; it varies as processes join or leave the group. In fact, the set of *potential* processes in the system may be infinite. Consider one of the most interesting failure detectors, the so-called *eventually strong* failure detector. This is defined in terms of the following two properties: (1) all faulty processes must eventually be suspected permanently by all correct processes in the system (eventual strong completeness); and (2) some correct process must eventually cease being suspected by all correct processes in the system (eventual weak accuracy).

[2] Consider two processes, $p$ and $q$. Process $p$ is participating in the computation but experiences a failure and stops participating without executing another step. Process $q$ is participating in the computation and executes all its steps and terminates. We model both $p$ and $q$ as no longer participating.

In our new setting, "all correct processes in the system" is a huge (possibly infinite) set. Furthermore, most processes in this set are irrelevant to the group: they will never become members of it. It makes no sense to require the exclude oracle to exclude the many processes that may fail but which are not part of the group. More importantly, the existence of *some* process in the system that is eventually not excluded by any correct one is small comfort: we want to be assured that such a process will exist in the group, not in the entire system at large. In other words, we would like to state the completeness and accuracy properties relative to the set of processes in "the group". The trouble is that there is no such thing as "the group" yet: we are trying to define a formal model in which to describe "the group"!

We resolve this apparent circularity by defining the behavior of the exclude oracle in terms of a parameter which specifies a set of processes, and is intended to be "the current membership of the group". The exclude oracle, formally speaking, must be prepared to respond to a query where the set of processes specified in this parameter could be any set of processes whatsoever. In fact, processes are restricted to querying the exclude oracle using the current membership of the group as the actual parameter in the course of an execution. (This will become clearer when we define "well-formed runs" in the discussion of computations.) With these remarks in mind to help guide our intuition, we are now ready to formally define exclude and include oracles.

An *exclude oracle history* is a function $XOH : \Pi \times \mathcal{T} \times 2^{\Pi} \rightarrow 2^{\Pi}$. $XOH(p, t, \varrho)$ is the set of processes that the exclude oracle of process $p$, at time $t$, believes should be excluded. The parameter $\varrho$ is intended to represent the current set of processes in the group. Thus, informally, an exclude oracle history represents a possible behavior of the exclude oracle. An *exclude oracle* $XO$ is a function that maps a participation pattern $PP$ to a set of exclude oracle histories $XO(PP)$. This definition reflects the intuition that the behavior of the exclude oracle depends, in part, on the participation pattern. The participation pattern, however, does not completely determine the behavior of the exclude oracle[3] — hence the exclude oracle maps the participation pattern to a *set* of possible behaviors.

An *include oracle history* is a function $IOH : \Pi \times \mathcal{T} \rightarrow 2^{\Pi}$. $IOH(p, t)$ represents the set of processes that the include oracle of process $p$ believe should be included into the group at time $t$, and is precisely the set it will return if it is queried by $p$ at that time. An *include oracle* $IO$ is a function that maps a participation pattern $PP$ to a set of include oracle histories $IO(PP)$. Note that the parameter $\varrho$ used in the definition of an exclude oracle history does not apply here. $\varrho$ is intended to represent the set of processes currently in the group; it does not contain processes that are trying to join the group but have not yet succeeded in doing so.

In order to circumvent the impossibility result of [8], we must place some restrictions on the behavior of the oracles. These restrictions are the analogues of the eventual completeness and accuracy properties of the eventual strong

[3] Application semantics and inaccuracies are also reflected in the behavior of the exclude oracle.

failure detector mentioned earlier, adapted to the setting of exclude oracles. We say that an exclude oracle $XO$ is an *Eventually Strong XO* if the following two properties are satisfied:

*Property 1 (XO Eventual Completeness).* Eventually every persistently participating process in the current group excludes forever every process in the current group that isn't persistently participating.

$$\forall PP,\ \forall XOH \in XO(PP),\ \forall \varrho \in 2^{\Pi},\ \exists t \in \mathcal{T},\ \forall t' \geq t,$$
$$\forall q \in \varrho \cap (\Pi - Persist(PP)),\ \forall p \in \varrho \cap Persist(PP),\ q \in XOH(p, t', \varrho)$$

*Property 2 (XO Eventual Accuracy).* If the current group contains a persistently participating process then there is a time after which some persistently participating process in the current group is never excluded by any persistently participating process in the current group.

$$\forall PP,\ \forall XOH \in XO(PP),\ \forall \varrho \in 2^{\Pi}\ (\varrho \cap Persist(PP) \neq \emptyset \Rightarrow \exists t \in \mathcal{T},$$
$$\exists q \in \varrho \cap Persist(PP),\ \forall t' \geq t,\ \forall p \in \varrho \cap Persist(PP),\ q \notin XOH(p, t', \varrho))$$

We define an include oracle $IO$ as an *Eventually Strong IO* if the following properties are satisfied:

*Property 3 (IO Eventual Completeness).* Eventually every persistently participating process permanently includes every persistently participating process.

$$\forall PP,\ \forall IOH \in IO(PP),\ \exists t \in \mathcal{T},\ \forall t' \geq t,$$
$$\forall q \in Persist(PP),\ \forall p \in Persist(PP),\ q \in IOH(p, t')$$

*Property 4 (IO Eventual Accuracy).* Eventually no process includes a process that is not persistently participating.

$$\forall PP,\ \forall IOH \in IO(PP),\ \exists t \in \mathcal{T},\ \forall t' \geq t,$$
$$\forall q \in \Pi - Persist(PP),\ \forall p \in \Pi,\ q \notin IOH(p, t')$$

### 2.2 Views

The term "view" is traditionally used to refer to the current membership of the group (at some process). It is appealing to define a view simply as a set of processes, but upon reflection this turns out to be inadequate. To see why, consider the following situation: Suppose that the membership of the group is originally $\{p, q\}$. A process $r$ is then added, so that the membership becomes $\{p, q, r\}$. Process $r$ is then removed, and the membership becomes $\{p, q\}$, again. We would like to distinguish the initial view from the final view. Yet, considered as sets, the two views are identical.

This example leads us to define views as identifiers in some set (for the sake of concreteness, $\mathbf{Z}^+$). We assume that a function can "decode" each identifier $v$ by mapping it into the set of processes that makes up the group's membership when the view is represented by $v$. Formally, a *view* is an identifier from the set $\Psi = \mathbf{Z}^+ \cup \{nil, fin\}$. The special view $nil$ (the null view) represents that no view has been installed, and the special view $fin$ (the final view) represents the end of a process's participation in the group. We define $\hat{\Psi} = \Psi - \{nil, fin\}$. The "decoding" function is $Members : \Psi \rightarrow 2^{\Pi}$, where $Members(v)$ is the set of processes in the view $v \in \Psi$. We assume that $Members(nil) = Members(fin) = \emptyset$. Note that, in general, it is possible to have two views $v, v' \in \hat{\Psi}$ such that $v \neq v'$, but $Members(v) = Members(v')$. If $p \in Members(v)$, we say that view $v$ *contains* process $p$.

We assume that the state of a process includes information about the most recently installed view. We define $ViewState(\sigma)$ to be the view corresponding to state $\sigma$. $ViewState(\sigma)$ is $nil$ if no view from $\hat{\Psi}$ has been installed and $fin$ if the process is no longer participating after having installed a view from $\hat{\Psi}$.

### 2.3 Computations

A *configuration* is a pair $\langle s, M \rangle$ where $s$ is a function that maps a process $p$ to its local state and $M$ is a set of messages (intuitively, the set of messages sent and not yet received). An *algorithm* is a collection of deterministic automata, one per process. In a single *step* of a given algorithm, a process $p$ may perform one of the following actions:

- $\langle send, p, m \rangle$ send the message $m$;
- $\langle recv, p, m \rangle$ receive the message $m$;
- $\langle excl, p, \varrho, \gamma \rangle$ query the exclude oracle of process $p$ about set $\varrho$ with result $\gamma$;
- $\langle incl, p, \delta \rangle$ query the include oracle of process $p$ with result $\delta$;
- $\langle change, p, \sigma \rangle$ change the state of process $p$ to $\sigma$.

If a process $p$ has state $\sigma$ then a step $e$ by $p$ is *applicable* in configuration $C = \langle s, M \rangle$ if $p$'s automaton stipulates that the step $e$ can be taken in state $\sigma$ and, in addition, $m \in M$ for a receive step $e = \langle recv, p, m \rangle$. $e(C)$ denotes the configuration resulting from the application of step $e$ to $C$.

A *schedule* $S$ of an algorithm $A$ is a sequence of steps of $A$. $S[i]$ is the $i^{th}$ step of the sequence. A schedule $S$ of an algorithm $A$ is applicable to a configuration $C$ if and only if (a) $S$ is the empty schedule, or (b) $S[1]$ is applicable to $C$ and $S[2]$ is applicable to $S[1](C)$ and $S[3]$ is applicable to $S[2](S[1](C))$, etc.

An initial configuration $I = \langle s, M \rangle$ is *valid* if $M = \emptyset$ and there exists a unique $v^* \in \hat{\Psi}$ such that for each process $p$ the state function $s$ provides: $ViewState(s(p)) = v^*$ if $p \in Members(v^*)$ and $ViewState(s(p)) = nil$ otherwise. We define $Input(I)$ to be $v^*$.

Let $A$ be an algorithm, $XO$ be an exclude oracle, and $IO$ be an include oracle. A *run* of $A$ with $XO$ and $IO$ is a tuple $R = \langle PP, XOH, IOH, I, S, T \rangle$ where $PP$ is a participation pattern, $XOH \in XO(PP)$ is an exclude oracle history, $IOH \in IO(PP)$ is an include oracle history, $I$ is an initial configuration of $A$, $S$ is an infinite schedule of $A$ that is applicable to $I$, and $T$ is an infinite sequence of increasing time values (indicating when each step in $S$ occurred according to the fictional global clock). We define $ViewRun(p, i, R)$ to be $ViewState(s(p))$, where $s$ is the process state function of the configuration after the first $i$ steps of schedule $S$ starting from initial configuration $I$ in the run $R$; we define $ViewSet(p, i, R) = Members(ViewRun(p, i, R))$.

A run, $R$, is *well-formed* if $\forall i \in \mathbf{Z}^+$ and $\forall p \in \Pi$ it satisfies the following properties:

- $I$ is a valid initial configuration;
- if $S[i] = \langle excl, p, \varrho, \gamma \rangle$ then $\varrho = ViewSet(p, i, R)$[4] and $\gamma = XOH(p, T[i], \varrho)$;
- if $S[i] = \langle incl, p, \delta \rangle$ then $\delta = IOH(p, T[i])$;
- if $p \in Persist(PP)$ then $p$ takes an infinite number of steps in $S$;
- every message sent to a process $p \in Persist(PP)$ is eventually received;
- if $p \notin PP(T[i])$ then $XOH(p, T[i], \varrho) = \emptyset$ for any $\varrho$;
- if $p \notin PP(T[i])$ then $IOH(p, T[i]) = \emptyset$;
- if $p \notin PP(T[i])$ and $\exists i' < i$ such that $ViewRun(p, i', R) \in \hat{\Psi}$ then $\forall i'' \geq i, ViewRun(p, i'', R) = fin$.

From now on, we consider only well-formed runs.

# 3 The Specification

In this section, we give a specification that captures the basic notions of Group Membership. We say that algorithm $A$ solves Group Membership (GM) with oracles $XO$ and $IO$ if every run $R = \langle PP, XOH, IOH, I, S, T \rangle$ of $A$ satisfies the safety properties and liveness properties described in this section.

## 3.1 Safety Properties

The first three safety properties, which we call "validity properties" require that the group's membership does not change unless there is a good reason. First we explain the motivation for such properties. Some previously proposed specifications for group membership allow two kinds of undesirable behavior [4]: Capricious removal causes processes to expel others from the group for no reason; collective suicide causes processes to leave the group for no reason. Such behavior is undesirable because it allows for trivial (and useless) implementations of the group membership service, which force the group to be the empty set for no reason! Our specification avoids this pitfall by insisting that processes be removed from or added to the group *only* if the include or exclude oracles justify the change. Notice that unlike other specifications, in which changes to the membership are caused by processes executing "join" and "leave" events which are under their own control, the output of the include and exclude oracles is *not* under the process' control. More precisely, we require the following properties:

*Property 5 (Exclude Validity).* If $p$ installs view $v \in \hat{\Psi}$ and, later, view $v' \in \hat{\Psi}$ where $v$ contains $q$ and $v'$ does not contain $q$ then $q$ was previously in the output of some exclude oracle.

*Property 6 (Final View Validity).* If $p$ installs $fin$ then $p$ was previously in the output of some exclude oracle or $p$ is not persistently participating.

[4] As mentioned in the discussion leading up to the definition of an exclude oracle, the exclude oracle may only be queried about the current group.

*Property 7 (Include Validity).* If $p$ installs view $v \in \hat{\Psi}$ where $v \neq Input(I)$ and $v$ contains $q$ then $q$ was in the output of some include oracle.

The validity properties require that the group's membership should conform to the information given by the oracles. Ideally, one would like the members of the group at any time to reflect the *participation pattern*. However, the include and exclude oracles are the only means that processes have at their disposal to "sense" the participation pattern. Thus, if these oracles provide very inaccurate information about the participation pattern, the membership of the group may be far from the ideal. This is not a weakness of our specification. On the contrary, we believe that it is an advantage of our approach that it can meaningfully capture situations where the information about the state of the group is highly inaccurate due to unanticipated delays, inappropriately short settings of timeout timers or a variety of other reasons that can arise in practice.

The next two safety properties, which we call "integrity properties" formalize the semantics of the special views $nil$ and $fin$.

*Property 8 (Null View Integrity).* If $p$ installs a view $v \neq nil$ then $p$ cannot subsequently install $nil$.

*Property 9 (Final View Integrity).* If $p$ installs $fin$ then $p$ does not subsequently install another view.

Final View Integrity prohibits a process from leaving a group and then joining it again. In practice, the same "computational entity" can repeatedly leave and rejoin the group under a different name from $\Pi$.

The last safety property, which we call "agreement", stipulates that the sequence of views installed by the different processes fit together in a global sense: there is a single sequence of views such that each process installs a sequence of views that is a "window" to this one sequence.

*Property 10 (Agreement).* There exists a sequence $G = \langle v_0, v_1, v_2, \ldots \rangle$ of distinct views $v_i \in \hat{\Psi}$ such that $v_0 = Input(I)$ and the sequence of views from $\hat{\Psi}$ installed by any process $p$ is a contiguous subsequence of $G$.

### 3.2 Liveness Properties

The validity properties state that the membership does not change without good reason. Conversely, the liveness properties state that if a compelling reason does exist then the membership of the group will change. We say that a property $P$ holds *almost always* in a run if there is a time after which the property $P$ always holds.

*Property 11 (Exclude Progress).* Let $p$ be a persistently participating process that installs some view. If almost always $p$ excludes process $q$ and almost always all processes do not include $q$ then almost always $p$'s view does not contain $q$.

*Property 12 (Include Progress).* Let $p$ be a persistently participating process that installs some view. If almost always $p$ includes $q$ and $q$ never installs $fin$ then almost always $p$'s view contains $q$.

Property 11 states that if there is good reason to exclude a process $p$ from the group and no good reasons to include $p$ in the group then $p$ must eventually be dropped from the group. Property 12 states a similar progress requirement for inclusion into the group.

*Property 13 (Quiescent Progress).* If almost always all processes do not exclude $q$ and almost always all processes do not include $q$ then almost always all views contain $q$ or almost always no view contains $q$.

Property 13 states that the views eventually become stable with respect to processes that the oracles are silent about. Suppose the current view at process $p$ contains $\{p, q, r\}$. $r$ stops participating. $p$'s oracle requests the removal of $r$ successfully and the next view at $p$ contains $\{p, q\}$. All the oracles then give imperfect information and request the inclusion of $r$. Our specification allows the next view at $p$ to contain $\{p, q, r\}$. As long as the oracles are colluding to exclude and then include $r$, our specification allows this to happen repeatedly. When the oracles become silent with respect to $r$, Property 13 requires that the views become stable with respect to $r$.

*Property 14 (Include Notification).* If $p$ installs $v \in \hat{\Psi}$ and $v$ contains a persistently participating process $q$, then $q$ eventually installs $v$.

Property 14 requires that processes that have been included into the group, actually participate in the group.

Several of the properties in our specification refer to the behavior of the oracles. This is a departure from the way in which Consensus is specified in the analogous situation where we have failure detectors rather than oracles. There, the specification makes reference to the failure pattern (the analogue to our participation pattern) but not to the behavior of the failure detectors (the analogue to our oracles). Why not specify Group Membership with reference only to the participation pattern? The reason is the difference in the nature of the two problems. In Group Membership processes must compute something (the new view) that is related to the participation pattern. Their ability to do so depends on the oracles, which are the only means processes have for observing the participation pattern. It is therefore natural that the problem specification refers to the oracles. In contrast, what processes compute in Consensus is not in any way related to the failure pattern. It therefore stands to reason that the problem should be specifiable without reference to the failure detectors, which are the only means the processes have for observing the failure pattern. By requiring stronger properties of the oracles it is possible to limit and even to completely eliminate the reference to the oracles in the specification of Group Membership. The resulting specification, however, is meaningful only for restricted classes of oracles. We felt that the loss of generality is too high a price to pay for the gain in specification simplicity.

## 4 The Algorithm

In our model with an infinite number of processes and a computation that is "long-lived", it is possible that an infinite number of processes may stop participating in the computation. However, if too many processes stop participating simultaneously our algorithm will not be able to make progress. Thus we require a minimal level of participation to satisfy the liveness part of the specification. The safety properties are never violated. In the case of our group membership algorithm, we require a majority of processes in any view to be currently in the participation pattern. Formally, $\forall i \in \mathbf{Z}^+,\ \forall p \in \Pi,\ (PP(T[i]) \neq \emptyset \wedge ViewSet(p, i, R) \neq \emptyset \Rightarrow |ViewSet(p, i, R) \cap PP(T[i])| > |ViewSet(p, i, R)|/2)$.

The algorithm assumes that the exclude oracle is an eventually strong XO (Properties 1 and 2) and that the include oracle is an eventually strong IO (Properties 3 and 4).

In our pseudo-code, we use the notation $xvar \leftarrow XO$ to indicate that variable $xvar$ gets the value of the local exclude oracle when queried about the set of processes corresponding to the current view. The notation $ivar \leftarrow IO$ indicates the variable $ivar$ gets the value of the local include oracle.

We assume all messages that arrive at a process are kept in a local message pool. The pool can be searched by any task of a process for messages satisfying certain criteria. We assume that all messages have a tuple form. We define an *retrieve* primitive to extract messages from the pool.

The *retrieve* primitive takes as an argument a tuple consisting of two types of parameters: "extract" parameters and "pattern" parameters. Parameters that are preceded with a ? are extract parameters and parameters that are not preceded by a ? are pattern parameters. $retrieve\langle a_1, a_2, \ldots, a_n\rangle$ returns $false$ if there is no message in the pool that consists of $n$ parameters where each of the pattern parameters in the argument to *retrieve* matches in position and value with the parameters of the message in the pool. $retrieve\langle a_1, a_2, \ldots, a_n\rangle$ returns $true$ if there is such a message. Further, each of the extract parameters in the argument to *retrieve* is assigned the value of the corresponding parameter in the message. If more than one such message exists in the pool, one is selected arbitrarily.

### 4.1 View Consensus

At the heart of our GM algorithm lies a variant of the well-known Consensus problem, that we call View Consensus (VC). In this section, we define the problem and give an algorithm that solves it. VC is defined in terms of two primitives, VC-propose$(v, v')$ and VC-decide$(v, v'')$ where $v$, $v'$ and $v''$ are views. Intuitively, a process invokes VC-propose$(v, v')$ to propose $v'$ as a possible view to succeed $v$. If a process executes VC-decide$(v, v'')$ it has decided on $v''$ as the view to succeed $v$. We use the notation VC-propose$(v, -)$ and VC-decide$(v, -)$ to indicate that the second parameter can be any view. VC is specified by the following properties:

*Property 15 (VC-Uniform Agreement).* If $p$ executes VC-decide$(v, v')$ and $q$ executes VC-decide$(v, v'')$ then $v' = v''$.

*Property 16 (VC-Termination).* If a process $p$ executes VC-propose$(v, -)$ where $p \in Members(v)$ and $p$ does not stop participating while in the view $v$ then $p$ executes VC-decide$(v, -)$.

*Property 17 (VC-Uniform Integrity).* Every process that executes VC-decide$(v, -)$ does it at most once.

*Property 18 (VC-Uniform Validity).* If $p$ executes VC-decide$(v, v')$ some process previously executed VC-propose$(v, v')$.

In Figure 1 we use the rotating coordinator algorithm of Chandra and Toueg to implement VC.

### 4.2 Group Membership Service

In Figure 2, we give a protocol to satisfy the specification for GM. The protocol runs through three phases. Phase A is only applicable to processes that are not part of the initial group. These processes wait until a message of type $J$ arrives with the first view to be installed. Once this message is retrieved, processes proceed to phase B. Processes in the initial group proceed directly to phase B.

In phase B of the protocol, each process tries to build, in the variable $tryset$, a set of processes to propose as the next view. Each process does this by querying its local include and exclude oracles. Any process that is successful in building a new $tryset$ sends its $tryset$ to all members of the current view. By the end of phase B, each member of the current view has some $tryset$ (whether it was sent by some other member of the current view or successfully built by querying the oracles) to propose for View Consensus.

Phase C begins with View Consensus for the current view with the proposal created in phase B. The process then waits for the resulting new view from View Consensus. If the current process is a member of the new view then the process installs the new view. Otherwise, the process installs $fin$. We keep track of all processes that have been removed from the group in the variable $dropped$. These processes are prevented from being included again during phase B. At the end of phase C, messages of type $J$ are sent to any new processes joining the group (which are still in phase A). Finally, the processes that have not installed $fin$ at the end of phase C, begin again at phase B.

The protocol satisfies some common notions of Group Membership that are not part of the GM specification. For example, the protocol satisfies Self Inclusion[5] which is not required by the specification. Our model assumes that each of the views installed contain a majority of participating processes. A natural modification to the algorithm is to ensure the protocol only proposes $tryset$'s of sufficient size to accurately reflect the assumptions of the model.

## 5 Concluding Remarks

By using oracles, we have abstracted, into well defined modules, the reasons a process might be included into or excluded from a group by a group membership

[5] If $p$ installs a view $v \in \hat{\Psi}$ then $p \in Members(v)$.

Every process $p$ executes the following:

```
cobegin
procedure VC-propose (v, v')
    n ← |Members(v)|                                  ▷ no. of processes in the view
    est ← v'                                          ▷ the current estimate
    tsp ← 0                                           ▷ timestamp est was adopted
    rnd ← 0                                           ▷ the current round
    while p has not executed VC-decide(v, −) do
        rnd ← rnd + 1
        coord ← the (rnd mod n) + 1 member of Members(v)
        send ⟨P1, v, p, rnd, est, tsp⟩ to coord       ▷ PHASE 1: send estimates
        if p = coord then                             ▷ PHASE 2: gather estimates
            wait until [any ⌈(n + 1)/2⌉ processes q: retrieve
            ⟨P1, v, q, rnd, ?rep, ?reptsp⟩]
            msgs[rnd] ← the set of messages retrieved above
            t ← max reptsp : ⟨−, −, −, −, −, reptsp⟩ ∈ msgs[rnd]
            est ← select one rep : ⟨−, −, −, −, rep, t⟩ ∈ msgs[rnd]
            send ⟨P2, v, coord, rnd, est⟩ to Members(v)
        exc ← ∅                                       ▷ PHASE 3: adopt an estimate
        wait until [retrieve ⟨P2, v, coord, rnd, ?newest⟩ or coord ∈ (exc ← XO)]
        if coord ∉ exc then
            est ← newest
            tsp ← rnd
            send ⟨P3, v, p, rnd, ACK⟩ to coord
        else
            send ⟨P3, v, p, rnd, NAK⟩ to coord
        if p = coord then                             ▷ PHASE 4: count acks
            wait until [any ⌈(n+1)/2⌉ processes q: retrieve ⟨P3, v, q, rnd, ACK⟩ or
            ⟨P3, v, q, rnd, NAK⟩]
            if ⌈(n + 1)/2⌉ ACK messages retrieved above then
                send ⟨D, v, p, est⟩ to p

task VC-decide
    while true do
        if retrieve ⟨D, ?v, ?q, ?newview⟩ then
            if p has not executed VC-decide(v, −) then
                send ⟨D, v, q, newview⟩ to Members(v)
                VC-decide(v, newview)
coend
```

**Fig. 1.** View Consensus

service. Further research is needed into more refined specifications of oracles for specific applications, as well as implementations of such application-specific oracles.

In this paper, we did not discuss group communication primitives (multicasts). This can be an extensive topic with a wide variety of interesting ordering

Every process $p$ executes the following:

$v = nil$ ▷ *the current view*
**if** $p$ is in the initial view **then**
    $v =$ initial view
$dropped \leftarrow \emptyset$ ▷ *all processes removed*
**while** $v = nil$ **do** ▷ *PHASE A: first view*
    **if** retrieve $\langle J, ?v, ?dropped \rangle$ **then**
        install($v$)
**while** $v \neq fin$ **do** ▷ *PHASE B: build tryset*
    $tryset \leftarrow Members(v)$ ▷ *set possible in next view*
    **while** $tryset = Members(v)$ **do**
        $joins \leftarrow IO - dropped - Members(v)$ ▷ *the set of processes to add*
        $quits \leftarrow (XO - IO) \cap Members(v)$ ▷ *the set of processes to drop*
        $tryset \leftarrow (Members(v) \cup joins) - quits$
        **if** retrieve $\langle T, v, ?sometryset \rangle$ **then**
            $tryset \leftarrow sometryset$
    send $\langle T, v, tryset \rangle$ to $Members(v)$
    $v' \leftarrow$ view encoding such that $Members(v') = tryset$ and $v' > v$
    VC-propose($v, v'$) ▷ *PHASE C: view consensus*
    **wait until** [some $newv$: $p$ has executed VC-decided($v, newv$)]
    $oldv \leftarrow v$
    **if** $p \in Members(newv)$ **then**
        $v \leftarrow newv$
    **else**
        $v \leftarrow fin$
    install($v$)
    $dropped \leftarrow dropped \cup (Members(oldv) - Members(newv))$
    send $\langle J, newv, dropped \rangle$ to $Members(newv) - Members(oldv)$

**Fig. 2.** Group Membership Service

and delivery requirements. We believe this group membership specification will provide a good basis upon which to study multicasts and are currently working on the issue.

In our model, we assume reliable links. We plan to investigate the work of Aguilera, Chen and Toueg [1] on intermittent connectivity with respect to our model and algorithms.

**Acknowledgments**

We would like to thank Rachid Guerraoui, Wai-Kau Lo, André Schiper and the anonymous reviewers for their useful comments.

## References

1. M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. Technical Report TR97-1631, Department of Computer Science, Cornell University, May 1997.

2. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: a communication sub-system for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.
3. Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. In *ACM Transactions on Computer Systems 13 4*, pages 311–342, Nov. 1995.
4. E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report 95-1534, Department of Computer Science, Cornell University, Aug. 1995.
5. O. Babaoglu, R. Davoli, L.-A. Giachini, and M. G. Baker. Relacs: a communications infrastructure for constructing reliable applications in large-scale distributed systems. In *BROADCAST Project*. Department of Computer Science, University of Newcastle upon Tyne, UK, 1994.
6. K. P. Birman. The process group approach to reliable distributed computing. In *Communications of the ACM*, pages 36–53, Dec. 1993.
7. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving Consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
8. T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charon-Bost. On the impossibility of group membership. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996.
9. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.
10. F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. In *Distributed Computing 4(4)*, pages 9–16, July 1987.
11. F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, Apr. 1991.
12. P. Ezhilchelvan, R. Macêdo, and S. Shrivastava. Newtop: a fault-tolerant group communication protocol. In *Proceedings of the 15thInternational Conference on Distributed Computing Systems*, June 1995.
13. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
14. R. Guerraoui and A. Schiper. The generic consensus service. Technical Report TR 98/282, Ecole Polytechnique Fédérale de Lausanne, July 1998. Revised version appeared in *Proceedings of the 26th IEEE International Symposium on Fault-Tolerant Computing (FTCS-26)*, Sendai, Japan, June 1996.
15. M. F. Kaashoek and A. S. Tanenbaum. Group communication in the amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computer Systems*, pages 222–230, May 1991.
16. D. Powell. Special issue on group communication. *Communications of the ACM*, pages 50–97, Apr. 1996.
17. R. D. Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic view-oriented group communication service. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, pages 227–236, 1998.
18. A. Ricciardi and K. Birman. Process membership in asynchronous environments. Technical report, Department of Computer Science, Cornell University, Apr. 1994. available by anonymous ftp from ftp.cs.cornell.edu in pub/aleta/AsyncMembService.ps.
19. R. van Renesse, K. P. Birman, R. Cooper, B. Glade, and P. Stephenson. The horus system. In *Reliable Distributed Computing with the Isis Toolkit*, pages 133–147, 1993.