Primary Component Asynchronous Group Membership as an Instance of a Generic Agreement Framework*

Fabíola GREVE Michel HURFIN Michel RAYNAL Frédéric TRONEL
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{fgreve,hurfin,raynal,ftronel}@irisa.fr

Abstract

Group-based computing is becoming more and more popular when one has to design a middleware able to support reliable distributed applications. This paradigm is made of two basic services, namely, a group membership service and a group communication service. More generally, a group is a set of processes cooperating to carry out a common task (e.g., copies of a replicated server, participants in a transaction or users in a CSCW*-based application). Due to the desire of new processes to join the group, to the desire of a group member to leave it, or to process crashes, the composition of a group can evolve dynamically. The set of processes that currently implements the group is called the current view of the group.*

This paper addresses the specification and the implementation of a primary component group membership service. "Primary component" means that the specification imposes to have a single view at any time. The paper first proposes a specification for the problem. Then it presents a protocol that implements that specification in asynchronous distributed systems equipped with failure detectors. This primary component group membership protocol is obtained as an appropriate instantiation of a general agreement framework.

Keywords*: Asynchronous Distributed System, Group Membership Problem, Primary Component, Partitionable System, Process Crash.*

1 Introduction

Group-Based Computing is a powerful paradigm which aims at facilitating the design and implementation of fault-tolerant distributed applications and services. This paradigm is offered to distributed middleware designers by several systems ([12] describes some of them). Group-based computing actually allows middleware designers to construct reliable services or implement reliable computations on top of unreliable distributed systems. Indeed, replication is a key to provide a higher degree of fault-tolerance in distributed systems.

A *Group-Based Computing* facility is made of two parts, namely, a *Membership* service and a *Group Communication* service. The membership service provides processes with the current composition of the group. This is a main attribute of the state of the group. (The other attributes of the state of the group are related to the service or the computation the group is processing.) The group composition evolves according to the desire of processes to join the group or to leave it, and to the occurrence of crashes of current members of the group. The current group composition is usually named a *view*.

There are two types of membership services: *Primary Component* and *Partitionable*. A primary component membership service ensures that at any time the group is implemented by a single view. From a user point of view, this means the membership service ensures that the set of views is totally ordered. A partitionable membership service allows different views of the same group to coexist (*concurrent* views). In that case, the processes of each view behave as if they were the only ones that are currently implementing the group.

The aim of a group communication service is to provide application processes with communication primitives that are well-suited to the group computing paradigm. The main primitive offered by this service is a *Reliable Multicast* facility allowing a process to send messages to all group members. Basically, this primitive aims at rendering transparent the group composition to the users of the service offered by the group. Usually, order guarantees (e.g., total order, causal order) are associated with this multicast primitive [3]. As the group composition can evolve dynamically, the statement of a meaningful specification and the design of a correct implementation of a group communication service are far

*This work is supported by grants from Alcatel, France Télécom (98 1B 123) and CNPQ/Brazil (200.323-97).

from being trivial [2, 11, 14].

The group membership problem has been introduced and solved for the first time by Cristian [6] in the context of synchronous distributed systems. The work on the membership problem in asynchronous systems has been pioneered by the Isis system [3]. Unfortunately, its specification was incomplete [2]. The asynchronous group membership problem was later proved to be impossible to solve without additional assumptions (on the detection of crashed processes) [5]. The interested reader will find nice surveys on the membership problem in the asynchronous distributed systems context in [14]. He will also find discussions on the specification of membership services in [10, 11].

In this paper we are interested in the specification of a *Primary Component Group Membership* (PCGM) service and in the design of a protocol that implements it. The paper is made of six sections. Section 2 presents the underlying system model. Section 3 provides a specification for the PCGM problem. It appears that this problem cannot be solved in purely asynchronous distributed systems [5]. So, to solve it, we need to weaken the problem definition, and/or strengthen the system with additional assumptions. Those assumptions are related to the failure detection. They are discussed in Section 4 where Chandra-Toueg's *Unreliable Failure Detector* concept is introduced. Then, Section 5 proposes a primary component membership protocol that works in asynchronous distributed systems equipped with Chandra-Toueg's failure detectors. If the failure detector is perfect (it never makes a mistake), the protocol solves the membership problem as defined in Section 3. If the failure detector is not perfect, the protocol solves a weakened version of the problem. This shows there is an intrinsic tradeoff between the quality of the failure detector and the membership problem approximation that is solved. Finally, Section 6 concludes the paper.

The reader will find in [8] an expanded version of this paper, including more details on the implementation.

2 Distributed System Model

Group We consider groups that have a state. This means that if all processes that currently implement the group crash, then the group crashes [13]. At any time, the current group membership, as perceived by a member of this group, is referred as its current $view$. A view v is made of two fields: $v.id$ (its identity) and $v.members$ (its composition). (Specific assumptions on these fields will be made in Section 3 which defines the group membership problem.) Without loss of generality, in the following we assume there is a single group.

Processes The computing part of the system is made of an unbounded set of processes $\Pi = \{p_1, p_2, \dots\}$. This is the set of processes that may join the group. So, at any given time, only a subset of Π participate in the group computation. There is no bound on the relative process speed. So, processes are asynchronous.

With respect to the group, a process can invoke two operations, namely *join* and *leave*. A process joins the group by invoking the first one, and leaves it by invoking the latter. It is assumed that a process is member of the group at most once. So, the action of leaving the group is definitive. After it has invoked the *join* operation, a process p_i knows it belongs to the group when it is delivered a message *install*(v), where v is a view such that $i \in v.members$. Finally, after it has invoked the *join* operation, and before invoking the *leave* operation, a process can crash (premature halt). Crashes are assumed to be definitive (i.e., a crashed process does not recover).

The fact that a process cannot recover within the group (crashes are definitive) and that a process joins the group at most once, is not a real drawback. Actually, a process that exited from the group by invoking the *leave* operation or by crashing, can come back under a fresh identity.

Communication Network Each pair of processes is connected by a channel. There is no bound on message transfer delays. Moreover, there is neither message alteration, nor creation of spurious messages, nor message losses. So, the communication is reliable but asynchronous.

3 The Primary Component Group Membership Problem

Informally, the *Primary Component Group Membership* (PCGM) problem consists in providing the processes that currently implement the group with a view that is consistent with the past history of the group, namely, the *join* and *leave* operations that have been already executed, and the crashes of processes that were members of the group.

The properties defining the PCGM problem can be decomposed in two sets, namely safety properties and a liveness property. The safety properties maintain the consistency of the group membership, while the liveness property ensures that it evolves according to the will of processes (join and leave) and to the system behavior (process crashes). When p_i delivers *install*(v), we say "p_i installs view v".

Safety Properties The safety specification is made up of six properties.

- Validity property. If a process p_i installs a view v, then $i \in v.members$. Moreover, a process installs a view at most once. This property states that a view is only installed by its members. It is sometimes named "self-inclusion" property.

- Total Order property. The set of views installed by processes is totally ordered[1].

  Notation - Let V denote this sequence of views, and let $v \in V$. By definition, if v is the k-th view of V, then $v.id = k$. If v is not the last view (if any), $succ(v)$ denote the view that is immediately after v in this sequence. Moreover, $succ^{+}(v)$ denotes the set of all the successors of v.

- Initial View property. There is an initial view (first element of V) whose members are predefined. These initial member processes are de facto in the first view, without having to invoke the *join* primitive. The initial states of these processes collectively define the initial state of the group.

- Agreement property. $\forall\ p_i$, let V_i be the sequence of views installed by p_i. Then, V_i is a subsequence of contiguous elements of V. This property means that the sequences of views installed by processes are globally consistent (they could have been seen by an external "idealized" observer, its observation being V).

- Justification property. Let v be any view such that $succ(v)$ exists. Then:

  - $v.members \neq succ(v).members$.
  - $\forall j \in (succ(v).members \setminus v.members) : p_j$ has invoked *join*.
  - $\forall j \in (v.members \setminus succ(v).members) : p_j$ has invoked *leave* or has crashed.

  The last property states that the progress from a view to the next one has to be justified by some cause (a *join*, a *leave* or a crash).

- State Transfer property. $\forall v, v'$ such that $v' = succ(v)$, then $\exists i \in v.members \cap v'.members$ such that p_i installs v'. The group crashes when all the processes that currently implement the group crash. This property is necessary when the group is not stateless. It aims at allowing the processes of the next view to reconstruct a consistent state of the (service/computation implemented by the) group. This comes from the fact that processes belonging to two consecutive views are able to convey state information from a view to the next one.

[1] At any time, there is a single current view. Hence the name *Primary Component* given to the current view.

Liveness Property The liveness specification consists of the following property, which states that if something (*join*, *leave* or crash) happens, then the membership is updated accordingly.

- Termination property.

  - If p_i invokes *join*, then either $\exists v : p_i$ installs v, or p_i eventually crashes.
  - If p_i invokes *leave* or crashes while it is in view v, then $\exists v' : v' \in succ^{+}(v) \ \wedge \ i \notin v'.members$.

4 How to Solve the Problem

4.1 An Impossibility Result

As previously noted, it has been shown in [5] that the PCGM problem is impossible to solve in asynchronous distributed systems. Intuitively, this is due to the fact that there is no way to distinguish a crashed process from a slow process or from a process with which communication are very slow. Therefore, it is impossible to define a protocol that ensures both the justification property and the termination property of the PCGM problem. This impossibility result is of the same nature as the impossibility to solve the consensus problem in asynchronous distributed systems [7].

A way to circumvent this impossibility consists in weakening the problem and/or in strengthening the underlying asynchronous distributed system. Such an approach has been successfully used to solve other problems such as the *Consensus*, *Atomic Broadcast*, *Atomic Multicast*, or the *Atomic Commitment* problems.

4.2 Unreliable Failure Detectors

Following the approach introduced in [4], we consider the distributed systems is augmented with a failure detector. A failure detector can be seen as an oracle that provides hints on crashed processes. Formally, a failure detector is defined by two properties, namely a *Completeness* property and an *Accuracy* property. The completeness property is on the actual detection of crashes; the accuracy property restricts the mistakes a failure detector can make. In this paper, we are interested in the *Strong Completeness* property [4], which states that eventually, every process that crashes is permanently suspected by every correct process. Besides, we consider the following accuracy properties [4]: i) *Perpetual Strong Accuracy* - no correct process is suspected before it crashes. ii) *Eventual Weak Accuracy* - there is a time after which some correct process is never suspected. Combined with the completeness property, these accuracy properties define respectively, two particular classes of failure detectors [4]: i) $\mathcal{P}$: The class of *Perfect* failure detectors. Those failure detectors never make mistakes. ii) $\Diamond\mathcal{S}$: The class of *Eventually Strong* failure detectors. Those failure detectors can make an arbitrary number of mistakes[2].

The strong completeness property can be realized by the use of "I am alive" messages and timeouts. An accuracy property can only be approximated in a fully asynchronous distributed system. It can be insured only in systems that satisfy some synchrony assumptions.

In this paper we consider the output of a failure detector is limited to the processes that have executed a *join* and have not yet executed a *leave*. So, a process that crashes before invoking *join*, or after having left the group is irrelevant from the failure detection point of view.

4.3 Towards a PCGM Protocol

Tradeoff If the asynchronous distributed system is equipped with a perfect failure detector, it is relatively easy to solve the PCGM problem. Yet, extra synchrony assumptions have to be considered to implement perfect failure detectors. As indicated in the Introduction, there is the following intrinsic tradeoff: the stronger the underlying failure detector is, the closer to PCGM the problem we can solve is. The next section proposes a protocol that solves a weakened version of the PCGM problem (called the WPCGM problem). This weakening consists in replacing "crashed process" by "process suspected to have crashed" in the justification property specified in Section 3. This protocol assumes that the system is only equipped with a failure detector of the class $\Diamond\mathcal{S}$, and that, during the lifetime of a view, a majority of its members do not crash.

Best Effort Property To be useful the protocol must make its best for the current view to reflect the real membership of the group. This means the protocol has not to systematically define a new view each time a group member "only suspects" another group member to have crashed. This *Best Effort* property has to be considered as a "first class" property of any membership protocol [14].

[2] $\Diamond\mathcal{S}$ is the weakest class of failure detectors that allows to solve the Consensus problem [4].

5 A Weakened Primary Component Group Membership Protocol

5.1 A WPCGM Protocol

The proposed WPCGM protocol is described in Figure 1. It is based on an underlying building block (line 8), namely, a general agreement framework (GAF). This building block is discussed in Section 5.2.

As far as the WPCGM protocol is concerned, a process p_i manages three local variables: its current view VM_i, and two sets, J_i and L_i. VM_i is equivalent to the set $v.members$ introduced in Section 3. J_i is the set of processes that p_i currently knows they want to enter the group. L_i is the set of processes that p_i currently knows they want to exit the group. These sets are meaningful only when p_i belongs to a view.

The behavior of a process p_i can be decomposed in three parts: an initialization part, a view management part and a JOIN/LEAVE message management part. We examine each of them in the following.

Initialization (lines 1-5): A process p_i can be an initial member of the group (line 1). In the other case, a process p_i may require to enter the group. This is done by sending a JOIN message (line 2) to the current members of the group. Such a message has to be received by at least one non crashed member of the current view. Then, the process effectively enters the group when it receives a notification message carrying a view including it (line 3). Then, p_i initializes its sets J_i and L_i to $\emptyset$ (line 5), and launches two subtasks, $T1$ and $T2$, respectively.

View management Task $T1$ (lines 6-12). Process p_i first installs the current view (line 7), and immediately launches, in its background, the computation of the next view. This is done by invoking (line 8) the GAF sub-protocol. This sub-protocol is an underlying building block appropriately instantiated to solve the membership problem.

When the execution of the GAF sub-protocol terminates (line 8), each process of the current view (VM_i) is provided with three sets of processes, namely, J, L and S. J is a set of processes that want to enter the group, L a set of processes that want to leave the group, and S a set of processes suspected of having crashed. According to these values, each non-crashed process p_i of the current view updates its local variables VM_i, J_i and L_i (lines 9-10), and notifies the new members on their membership to the current view (line 11). Then, if p_i is still a group member, it keeps on executing the view management protocol (lines 6-12). In the other case, it exits the WPCGM protocol.

(1) **if** ($p_i \in initial\_view$) **then** $VM_i \leftarrow initial\_view$;
(2) **else send** JOIN$< p_i >$;
(3) **wait until receive** (ACCEPT$< VM > and\ p_i \in VM$);
(4) $VM_i \leftarrow VM$; **endif**
(5) $J_i \leftarrow \emptyset; L_i \leftarrow \emptyset$;
cobegin
task $T1$:
(6) **while** ($p_i \in VM_i$) **do**
(7) $Install(VM_i)$;
(8) $< J, L, S > \leftarrow$ GAF(); {*agreement on view modifications via GAF*}
(9) $VM_i \leftarrow (VM_i \cup J) \setminus (L \cup S)$; {*compute new view*}
(10) $J_i \leftarrow (J_i \setminus J)$; $L_i \leftarrow (L_i \setminus (L \cup S))$;
(11) **broadcast** ACCEPT$< VM_i >$ **to** $p_j \in J$; {*notify acceptance of new members*}
(12) **endo**
task $T2$:
(13) **while** ($p_i \in VM_i$) **do**
(14) **upon reception of** $Valid$(JOIN$< p_j >$) **do**
(15) **broadcast** JOIN$< p_j >$ **to** $VM_i \setminus \{p_i\}$; $J_i \leftarrow J_i \cup \{p_j\}$; **enddo**
(16) **upon reception of** $Valid$(LEAVE$< p_j >$) **do**
(17) **broadcast** LEAVE$< p_j >$ **to** $VM_i \setminus \{p_i\}$; $L_i \leftarrow L_i \cup \{p_j\}$; **enddo**
(18) **endo**
coend

Figure 1. The WPCGM Protocol

Message management Task $T2$ (lines 13-18). While p_i belongs to the current view, it updates its sets J_i and L_i according to the JOIN/LEAVE messages it receives. When a JOIN message is received, p_i has to check whether this message is not an old one. So, a JOIN$< p_j >$ message is filtered through the predicate $Valid$ that accepts it only if p_j has not previously been a group member. Similarly, a LEAVE$< p_j >$ message is taken into account if and only if the process p_j belongs to the current view.

In order to prevent deadlock, when a group member p_i receives a JOIN/LEAVE message, it reliably broadcasts that message to all the current group members. This is done by forwarding the received message (lines 15 and 17) before updating its own data structure.

The correctness of this protocol relies on the GAF underlying building block. This block is discussed in the next section.

5.2 GAF: A Consensus-Based Approach

This section provides the reader with a short explanation on the way GAF does work. More details on the framework and a correctness proof of it can be found in [9]. The GAF framework defines a very general pattern from which solutions to particular agreement problems can be instantiated. A particular instantiation is obtained by an appropriate definition of five versatility parameters[3].

[3] Instantiations of GAF that solve the Weak Atomic Commitment, the Consensus problem, and the Atomic Broadcast problem are described in [9]. Here, by providing a GAF instantiation that solves the WPCGM problem, we actually extend the domain of problems that can be solved by instantiating GAF.

The GAF framework and its parameters are described in Figure 2 and Table 1, respectively. More precisely, GAF is a general agreement framework based on the well-known Chandra-Toueg's consensus protocol [4] (denoted CT). This protocol assumes an underlying failure detector of the class $\diamond\mathcal{S}$, and requires that a majority of processes participating in the consensus do not crash during its execution.

Parameter	Description
GET	Return the initial value to be proposed by a process
$\prec$	Define a partial order relation among input values
$\mathcal{F}$	Compute an output value from a set of input values
ACCEPTABLE	Validate the output decision
EXCUSED	Check if a process proposal is no longer necessary

Table 1. The GAF Versatility Parameters

In CT, each process proposes a value and all non-crashed processes decide on a value in such a way that (1) there is a single decided value, and (2) the decided value is a value that has been initially proposed by a process. The GAF framework makes "generic" Chandra-Toueg's consensus protocol by allowing it to be customized to solve particular agreement problems. These improvements concern the possibility for a process to change the value it has previously proposed (function GET), the computation of the decided value according to the set of proposed values (function $\mathcal{F}$), a checking to decide whether the computed decided value is an acceptable value (function ACCEPTABLE), and the allowed lack of the value of a process in some circum-

```
Framework GAF
begin
(1)  r_i ← 0; new_round_i ← true; est_i ← ⊥; ts_i ← 0; est_from_i ← [⊥, ⊥, ··· , ⊥];
(2)  while (true) do % The loop is from line 2 until line 46 %
(3)      if (new_round_i) % Initialize the round variables of p_i %
(4)        then new_round_i ← false; r_i ← r_i + 1; c ← COORD(r_i); phase1_begin_i ← true;
(5)             if (i = c) % Initialize the round coordinator variables %
(6)               then received_from_i ← ∅; tsm_i ← 0; phase2_end_i ← false; accept_i ← ∅; reject_i ← ∅
(7)      endif   endif;
(8)      if (ts_i = 0) % The value proposed by the upper layer application can be changed %
(9)        then est_from_i[i] ← GET(); % Get a new proposal %
(10)            if (est_i ≺ est_from_i[i]) then est_i ← est_from_i[i]; phase1_begin_i ← true endif
(11)     endif;
(12)     if (phase1_begin_i) then send(ESTIMATE< p_i, r_i, est_i, ts_i >) to p_c; phase1_begin_i ← false endif;
(13)     if a message m (as defined below) has been received
(14)       then case m of
(15)            • m =DECISION< j, est >    {m is from any p_j}
(16)              send(DECISION< i, est >) to all except {p_i, p_j}; return(est)
(17)            • m =NEW_ESTIMATE< c, r, new_est > such that r = r_i    {m is from p_c}
(18)              if (ACCEPTABLE(new_est))
(19)                then est_i ← new_est; ts_i ← r_i; send(VOTE< i, r_i, ack >) to p_c % p_i accepts new_est %
(20)                else send(VOTE< i, r_i, nack >) to p_c % p_i refuses new_est %
(21)              endif;
(22)              if (i ≠ c) then new_round_i ← true endif
(23)            • m =ESTIMATE< j, r, est, ts > such that r = r_i    {m is from any p_j to p_c (i = c)}
(24)              if not(phase2_end_i)
(25)                then received_from_i ← received_from_i ∪ {j};
(26)                     if (tsm_i < ts) then tsm_i ← ts; new_est_i ← est endif;
(27)                     if ((tsm_i = 0) and not(est ≺ est_from_i[j]))
(28)                       then est_from_i[j] ← est; new_est_i ← F(est_from_i) endif;
(29)                     if ( (|received_from_i| ≥ ⌈(n + 1)/2⌉) and (∀ p_j : j ∈ received_from_i or EXCUSED(j)) )
(30)                       then send(NEW_ESTIMATE< i, r_i, new_est_i >) to all; phase2_end_i ← true
(31)              endif   endif;
(32)            • m = VOTE< j, r, answer > such that r = r_i    {m is from any p_j to p_c (i = c)}
(33)              if (answer = ack)
(34)                then accept_i ← accept_i ∪ {j}; % The coordinator p_i counts the positive acknowledgments %
(35)                     if (| accept_i |= ⌈(n + 1)/2⌉)
(36)                       then send(DECISION< i, est_i >) to all except {p_i}; return(est_i)
(37)                     endif
(38)                else reject_i ← reject_i ∪ {j} % The coordinator p_i counts the rejections %
(39)              endif;
(40)              if (| accept_i ∪ reject_i |= ⌈(n + 1)/2⌉) then new_round_i ← true endif % Deadlock prevention %
(41)            endcase
(42)     endif;
(43)     if (¬(new_round_i) and (i ≠ c) and (p_c ∈ suspected_i))
(44)       then send(VOTE< p_i, r_i, nack >) to p_c; new_round_i ← true
(45)     endif
(46) endo
end
```

Figure 2. A General Agreement Framework (GAF)

stances (function EXCUSED). The $\prec$ relation allows to decide whether the new proposal of a process brings new information.

As indicated, Chandra-Toueg $\Diamond\mathcal{S}$-based consensus protocol [4] is the framework skeleton. Indeed, the framework generates CT-like protocols: each of them is based on the *rotating coordinator* paradigm and proceeds in consecutive asynchronous rounds until a decision is reached (execution of the **return** statement at line 16 or 36). At a given time the value of the variable r_i is equal to p_i's current round number. Each round is coordinated by a predetermined process that tries to impose a decision value. When considering a round r, the "current" coordinator is the process p_c whose identity is defined by the function COORD (line 4). Each process p_i manages a local variable est_i that represents its current estimate of the final decision value. A timestamp ts_i is associated with this value. Both values are updated as the protocol progresses and converges to the final decision value. During a round r, the cooperation between processes is based on a centralized communication scheme: each message (except the DECISION messages) is either sent to or received from the coordinator. Moreover, a message (except DECISION) sent during a round r can only be taken into account (lines 17, 23 and 32) by a process currently executing the same round. A round spans several while loop executions (lines 2-46). Each round is divided into four phases (those are the four phases of CT).

The first phase (lines 3-12): In this phase, each process sends to the current coordinator its own estimation of the final value (line 12). At the beginning of each round, the protocol checks if the upper layer application is allowed to provide a new input value (line 9 - function GET) and tests if the new value is more significant than the previous one (line 10 - order relation $\prec$).

The second phase (lines 23-31): This phase is only executed by the coordinator. It gathers estimates sent by processes during the first phase. As soon as a majority[4] of values have been collected (line 29: $|received\_from_i| \geq \lceil (n+1)/2 \rceil$) and if it turns out that no other estimate messages have to be gathered (line 29 - function EXCUSED), the coordinator computes (at most once per round) a new estimation that will be proposed to all the processes (line 30). The new estimate is either selected among the received estimates (line 26) or computed by applying a function to the set of gathered informations (line 28 - function $\mathcal{F}$).

The third phase (lines 17-22 and lines 43-45): In this phase, each process waits for a new proposition from the coordinator, and either suspects it to have crashed (line 43) or receives the new proposition (line 17). In the former case, a process sends a negative acknowledgment to the coordinator and runs into the next round. In the latter case, after validate the estimation (line 18 - function ACCEPTABLE), it either refuses it by sending a negative acknowledgment to the coordinator (line 20) or adopts it by sending a positive acknowledgment (line 19).

The fourth phase (lines 32-40): This phase is only performed by the coordinator. It waits for a majority of acknowledgment messages. If it receives only positive acknowledgments, it reliably broadcasts a decision message (lines 36 and 16). Otherwise, the coordinator proceeds to the next round (line 40). Note that an estimation is irremediably locked as soon as a majority of processes have sent a positive acknowledgment to the coordinator: after, no other value can be selected to be the final decision.

5.3 Instantiating the Framework to Solve the WPCGM Problem

The WPCGM protocol (Figure 1) has always in its background an execution of the GAF protocol whose aim is to detect changes in its current view and to define accordingly the new view. GAF decides as soon as it has computed an acceptable new view. The way GAF computes a new view is intimately related to the instantiation of its parameters.

[4]The coordinator is assured to receive at least a majority of estimations, because a majority of processes are correct by assumption.

The function GET: The GET function main goal is to define the input provided by p_i to the GAF agreement protocol. These inputs have to be of "sufficiently good" quality, in order the result (namely, a triple $< J, L, S >$) computed by GAF achieves the *Best Effort* requirement.

A first implementation of the GET function is the value returned by the triple $< J_i, L_i, suspected_i >$ (where $suspected_i$ is the current value provided to p_i by the underlying failure detector). However, this function definition does not provide the GAF protocol with the *Quiescence* property [1]. In our context, we interpret this property as follows: eventually, processes stop exchanging messages when no acceptable decision value can be output.

When GAF is instantiated with such a GET function, it is possible that all input values be equal to $\emptyset$ (no modification of the membership is suggested by the processes). When this occurs, processes execute extra rounds and continue to query their GET function and exchange messages until an acceptable value can be decided. A simple way to assure the quiescence property consists in ensuring that any value proposed by a process be a value that can be accepted as a GAF output value(Figure 3).

The technical report [8] provides a third definition of the GET function that limits the risk of false suspicions by providing a "better"approximation of the set of suspected processes.

Function GET()
begin
(1) % wait for a meaningful value %
    wait until $((J_i \cup L_i \cup suspected_i) \neq \emptyset)$
(2) $return(J_i, L_i, suspected_i)$
end

Figure 3. GET**() Function (Quiescent)**

The $\prec$ relation: The aim of the $\prec$ relation is to express the fact that some values are more significant than others. By definition, the less significant value is denoted $\perp$ and is equal to $< \emptyset, \emptyset, \emptyset >$. This value indicates that no modification has to be applied to the current view. To take into account a new value only if it carries "more information" than the previous ones, the $\prec$ relation is defined in the following way.

$$\forall (< J, L, S >, < J', L', S' >) :$$
$$< J, L, S > \prec < J', L', S' > \Longleftrightarrow (J \cup L \cup S) \subset (J' \cup L' \cup S')$$

The function $\mathcal{F}$: This function contributes to fine-tuning the value decided by GAF when defining the new view. It computes the tuple $< J, L, S >$ in the following way. When we consider joins and leaves, GAF has to take into account as much propositions as possible, so it returns the union of all the gathered sets of Joins and Leaves. When we consider suspicions, GAF lets in S only the processes that are suspected by every other process participating in the consensus execution. This ensures that a process can not be removed if it is not suspected at least by a majority of the current view members.

The functions ACCEPTABLE and EXCUSED: These functions have simple instantiations when GAF is used to solve the WPCGM problem. ACCEPTABLE(v) returns $true$ when v is not equal to $\perp$. EXCUSED(p_j) returns $true$ if p_j is suspected to have crashed. The function $\mathcal{F}$ is applied only when (1) a majority of estimate values has been collected and (2) all the non-suspected processes have proposed a value.

Thus, the GAF framework allows to select an instantiation depending on the properties that have to be ensured.

6 Conclusion

This paper addressed the specification and the implementation of a primary component group membership service. "Primary component" means that the specification imposes to have a single view at any time. The proposed primary partition group membership service has been built from a general agreement framework instantiated with appropriate parameter values. From a run-time point of view, as soon as a view has been determined and installed by its members, those process members launch in their background the computation of the next view. This computation takes into account the desire of processes that want to leave or join the group, and the set of current members that are suspected to have crashed.

References

[1] Aguilera M., Chen W. and Toueg S., Using the Hearbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks. *Theoretical Computer Science*, 220:3-30, 1999.

[2] Anceaume E., Charron-Bost B., Minet P. and Toueg S., On the Formal Specification of Group Membership Services. *Tech. Report 95-1534*, Cornell University, 1995.

[3] Birman K., The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37-53, 1993.

[4] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(1):225–267, 1996.

[5] Chandra T.D., Hadzilacos V., Toueg S. and Charron-Bost B., On the Impossibility of Group Membership. *Proc. 15th ACM Symp. on Princinciples of Distributed Computing - PODC*, pp. 322-330, 1996.

[6] Cristian F., Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4:175-187, 1991.

[7] Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.

[8] Greve, F., Hurfin M., Raynal M., Tronel F., Primary Component Asynchronous Group Membership as an Instance of a Generic Agreement Framework. *Tech. Report* 3856 INRIA, France, 2000. http://www.irisa.fr/EXTERNE/bibli/pi/2000/1292.

[9] Hurfin M., Macêdo R., Raynal M., and Tronel F., A General Framework to Solve Agreement Problems *Proc. 18th IEEE Symp. on Reliable Distributed Systems - SRDS*, pp. 56-65, 1999.

[10] Kal L. and Hadzilacos V., Asynchronous Group Membership with Oracles. *Proc. of the 13th Symp. on Distributed Computing - DISC*, LNCS #1693, pp. 87–100, 1999.

[11] Neiger G., A New Look at Membership Services. *Proc. 15th ACM Symp. on Princinciples of Distributed Computing - PODC*, pp. 331-340, 1996.

[12] Powell D. (Guest Editor). Special Issue on Group Communication. *Communications of the ACM*, 39(4), April 1996.

[13] Raynal M. and Tronel F., Group Membership Failure Detection: A Simple Protocol and its probabilistic Analysis. *Distributed Systems Engineering Journal*, 6(3):95-102, 1999.

[14] Vitenberg R., Keidar I., Chockler G.V. and Dolev D., Group Communication Specifications: A Comprehensive Study. *Tech. Report*, MIT-LCS-TM-593a, 1999.