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

Fabiola Greve, Michel Hurfin, Michel Raynal, Frédéric Tronel

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

Fabíola Greve, Michel Hurfin, Michel Raynal et Frédéric Tronel

N˚3856

Janvier 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3856--FR+ENG

RENNES

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

Fabíola Greve, Michel Hurfin, Michel Raynal et Frédéric Tronel

Thème 1 — Réseaux et systèmes
Projet ADP

Rapport de recherche n ° 3856 — Janvier 2000 — 17 pages

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. A group is a set of processes that cooperate in carrying out a common task. 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.

Key-words: Asynchronous Distributed System, Failure Detector, Group Membership Problem, Partitionable System, Primary Component, Process Crash, State Transfer, Unreliable Network.

(Résumé : tsvp)

This work has been partially supported by a CNET-FRANCE TELECOM grant 98 1B 123 and by CNPq/Brazil grant 200323-97.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Gestion de la composition d'un groupe grâce à un framework résolvant des problèmes d'accord

Résumé : Le concept de groupe est de plus en plus utilisé pour concevoir des couches logicielles permettant de supporter l'exécution d'applications réparties fiables. Ce paradigme recouvre deux services de base, à savoir, un service de gestion de la composition du groupe et un service de communication de groupe. Un groupe est un ensemble de processus coopérant à la réalisation d'une tâche commune. Étant donné que de nouveaux processus peuvent souhaiter rejoindre le groupe, que des membres du groupe peuvent souhaiter le quitter ou connaître des défaillances de type panne franche, la composition du groupe doit pouvoir évoluer de façon dynamique. L'ensemble des processus qui constituent le groupe à un instant donné est appelé la vue courante du groupe.

Cet article s'intéresse aux problèmes de la spécification et du développement d'un service de gestion de la composition du groupe ne tolérant qu'une partition primaire (une seule vue à un instant donné). Le papier propose tout d'abord une spécification du problème. Ensuite, il présente un protocole qui satisfait cette spécification dans un système réparti asynchrone équipé de détecteurs de défaillances. Le service de gestion de la composition du groupe via une partition primaire est obtenu en spécialisant de manière appropriée un framework résolvant des problèmes d'accord.

Mots-clé : Système réparti asynchrone, détecteur de défaillances, gestion de la composition d'un groupe, système partitionnable, partition primaire, panne franche, transfert d'état, réseau non-fiable.

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 ([21] 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 (such views are named *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 [4]. It is important to note that, 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 from being trivial [2, 3, 20, 26].

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 [7]. 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.

2 Distributed System Model

Group.
Informally, 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). The membership of a group can change according to the desire of its current members to leave the group, to the desire of processes that want to enter the group, or to the underlying system behavior (crashes, disconnection, etc) [4]. We consider groups that have a state. This means that if all processes that currently implement the group crash, then the group crashes [23].

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 $\Pi = \{p_1, p_2, \ldots\}$. 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.

3.1 Definition

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 the membership 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.

  This 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.

- 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 $\exists v' : v' \in succ^+(v) \ \wedge \ i \notin v'.members$.

  This property states that if something (*join*, *leave* or crash) happens, then the membership is updated accordingly.

3.2 Related Works

The group membership problem has been introduced and solved for the first time by F. Cristian [8]. His specification and solution were considering synchronous distributed systems. The work on the membership problem in asynchronous systems has been pioneered by the Isis system [4, 24]. 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) [7]. The interested reader will find nice surveys on the membership problem in the asynchronous distributed systems context in [3, 26]. He will also find a comparison between synchronous and asynchronous memberships in [9], and discussions on the specification of membership services in [17, 20].

4 How to Solve the Problem

4.1 An Impossibility Result

As previously noted, it has been shown in [7] that the PCGM problem is impossible to solve in asynchronous distributed systems. Intuitively, this is due to the fact that in asynchronous distributed systems, 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 [11].

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 the *Consensus* problem [5], the *Atomic Broadcast* problem [5], the *Atomic Multicast* problem [12], or the *Atomic Commitment* problem [13, 22].

4.2 Unreliable Failure Detectors

Following the approach introduced in [5, 6], 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 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.

Completeness and accuracy properties.

In this paper, we consider the following completeness property [5]:

- Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.

Among the accuracy properties defined by Chandra and Toueg [5] we consider here the two following ones:

- Perpetual Strong Accuracy: No correct process is suspected before it crashes.
- Eventual Weak Accuracy: There is a time after which some correct process is never suspected.

Classes of failure detectors.
Combined with the completeness property, these accuracy properties define the following classes of failure detectors [5]:

- $\mathcal{P}$: The class of *Perfect* failure detectors. This class contains all the failure detectors that satisfy the strong completeness property and the perpetual strong accuracy property. Those failure detectors never make mistakes.
- $\diamond\mathcal{S}$: The class of *Eventually Strong* failure detectors. This class contains all the failure detectors that satisfy the strong completeness property and the eventual weak accuracy property. 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 [18]. It can be insured only in systems that satisfy some synchrony assumptions [10].

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.

[2] $\diamond\mathcal{S}$ is the weakest class of failure detectors that allows to solve the Consensus problem [5]. $\diamond\mathcal{S}$-based Consensus protocols are described in [5, 15, 19, 25].

Best Effort Property.
Of course, 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 [26].

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 9), 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 1-6).

  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 6), and launches two subtasks, $T1$ and $T2$, respectively.

- View management: Task $T1$ (lines 7-14). Process p_i first installs the current view (line 8), and immediately launches, in its background, the computation of the next view. This is done by invoking (line 9)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 9), 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 10-12), and notifies the new members on their membership to the current view (line 13). Then, if p_i is still a group member, it keeps on executing the view management protocol (lines 7-14). In the other case, it exits the WPCGM protocol.

- Message management: Task $T2$ (lines 15-22). 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

```
(1)   if (p_i ∈ initial_view) then VM_i ← initial_view;
(2)   else send JOIN< p_i >;
(3)        wait until receive (ACCEPT< VM > and p_i ∈ VM);
(4)        VM_i ← VM;
(5)   endif
(6)   J_i ← ∅; L_i ← ∅;

cobegin
task T1:
(7)   while (p_i ∈ VM_i) do
(8)      Install(VM_i);
(9)      < J, L, S > ← GAF(); {agreement on view modifications via GAF}
(10)     VM_i ← (VM_i ∪ J) \ (L ∪ S); {compute new view}
(11)     J_i ← (J_i \ J);
(12)     L_i ← (L_i \ (L ∪ S));
(13)     broadcast ACCEPT< VM_i > to p_j ∈ J; {notify acceptance of new members}
(14)  endo

task T2:
(15)  while (p_i ∈ VM_i) do
(16)     upon reception of Valid(JOIN< p_j >) do
(17)           broadcast JOIN< p_j > to VM_i \ {p_i};
(18)           J_i ← J_i ∪ {p_j}; enddo
(19)     upon  reception of Valid(LEAVE< p_j >) do
(20)           broadcast LEAVE< p_j > to VM_i \ {p_i};
(21)           L_i ← L_i ∪ {p_j}; enddo
(22)  endo
coend
```

Figure 1: The WPCGM Protocol

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 17 and 20) 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 A Consensus-Based Approach: The General Agreement Framework

As indicated, the proposed WPCGM protocol is based on a general agreement framework (GAF) that we have developed in [16]. This 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].

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 [5] (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.

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

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	Verify when a process proposal is not necessary anymore

Table 1: The GAF Versatility Parameters

This section provides the reader an explanation on the way GAF does work. More details on the framework and a correctness proof of it can be found in [16]. We assume the reader is familiar with Chandra-Toueg $\diamond\mathcal{S}$-based consensus protocol [5]. As indicated, CT 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 (this variable is modified at lines 1 and 4). Each round is coordinated

[3]Instantiations of GAF that solve the Weak Atomic Commitment, the Consensus problem, and the Atomic Broadcast problem are described in [16]. 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.

```
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)

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. When the protocol starts, est_i, initialized to $\perp$, is timestamped 0 (line 1). This value is 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). Accordingly, the variable new_round_i is used to indicate that a new round has to be started. Each round is divided into four phases (those are the four phases of CT).

• **Phase 1:** In the first phase (lines 3-12), each process sends to the current coordinator its own estimate of the final value (line 12). This message called ESTIMATE contains four values: the identity of the sender (p_i), its current round number (r_i), its current estimate value (est_i) and the associated timestamp (ts_i). The boolean $phase1\_begin_i$ is used by p_i to know if it has to (re)start phase 1. As in CT, this phase is always executed at the beginning of a round: the boolean $phase1\_begin_i$ is set to *true* at line 4 when a new round is launched. This phase is also executed each time a new value proposed by p_i can be taken into account: the boolean $phase1\_begin_i$ is set to *true* at line 10 when the upper layer application is allowed to provide a more significant input value (see the descriptions of both the function GET (line 9) and the order relation $\prec$ (line 10)).

• **Phase 2:** Since all the ESTIMATE messages are exclusively sent to the current coordinator, the second phase (lines 23-31) of a round is executed only by the coordinator. The boolean $phase2\_end_i$ is used by the coordinator p_i to know if it has already completed this phase (line 24). This variable is initialized to $false$ when a new round, coordinated by p_i, starts (line 6). It is set to *true* when the coordinator ends the second phase by broadcasting a new estimate value (line 30) During this phase, the coordinator p_i first gathers estimates sent by processes during the first phase. Each time the current coordinator p_i receives an ESTIMATE message from a process, it executes three main actions:

(a) p_i adds the identity of the sender to the set $received\_from_i$. This set contains the identities of the processes from which p_i has received an estimate during the current round (lines 6 and 25). Thanks to this variable, the coordinator knows if a majority of estimates has been collected ($|received\_from_i| \geq \lceil (n+1)/2 \rceil$) and moreover it knows the identities of the processes from which it has not yet received an ESTIMATE message.

(b) Then the coordinator updates the value of the variable new_est_i called herein its new estimate. If it turns out that no other estimate messages have to be gathered, the new estimate 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 the function $\mathcal{F}$ to the set of gathered informations (line 28).

More precisely, p_i keeps track (by the mean of the variable tsm_i) of the maximal timestamp received during the current round (lines 6 and 26). If the coordinator has received at least one estimate whose timestamp is greater than zero, the value of new_est_i is set to an estimate whose timestamp is the greatest one (line 26). Otherwise, all the estimates it has received were timestamped zero, and they have been saved in an array est_from_i (line 28). In that case the value of new_est_i is the result returned by applying the function $\mathcal{F}$ to the array est_from_i (line 28, see the description of the parameter $\mathcal{F}$). Note that, for any k, the value of the variable $est\_from_i[k]$

is the most significant zero timestamped estimate[4] which has been sent by p_k and received by p_i or the default value $\perp$ (line 1) if no value has been received.

(c) The gathering of estimate values ends when enough estimates have been received by the current coordinator (lines 29-31). The coordinator is assured of receiving at least a majority of estimates, because a majority of processes is correct by assumption. Depending on the problem to solve, it can be necessary to collect more values (line 29, see the description of the parameter EXCUSED). Then, the coordinator proposes its new estimate by sending it to all processes (line 30). The message NEW_ESTIMATE broadcast by the coordinator contains three fields: the identity of the sender ($p_i = p_c$), its current round number (r_i) and the new estimate (new_est_i).

• **Phase 3:** In the third phase (lines 17-22 and lines 43-45) each process p_i waits for the receipt of a new estimate from the coordinator. Either p_i suspects the coordinator of having crashed (line 43), or p_i receives the new estimate (line 17). In the former case, a process sends a negative acknowledgment "NACK" to the coordinator (line 44). In the latter case, it either refuses the new estimate (by sending a negative acknowledgment "NACK" to the coordinator: line 20 and see the description of the function ACCEPTABLE) or adopts it (by sending a positive acknowledgment "ACK": line 19). If the new estimate is adopted, p_i updates the timestamp associated with its current estimate to the current value of the round counter (line 19). A message VOTE sent to the coordinator (lines 19, 20 or 44) contains three fields: the identity of the sender (p_i), its current round number (r_i) and a positive or negative acknowledgment (ACK or NACK).

• **Phase 4:** The fourth phase (lines 32-40) is performed only by the coordinator. It waits for a majority of acknowledgment messages. The variables $accept_i$ and $reject_i$ are used only by p_i when it is the current coordinator. Initialized to $\emptyset$ at the beginning of a round (line 6), these sets contain process identities: $accept_i$ (resp., $reject_i$) contains the processes that have sent a positive acknowledgment ACK (resp. a negative acknowledgment NACK) to the current coordinator, namely p_i.

If the current coordinator receives positive acknowledgments from a majority of processes (line 35), it reliably broadcasts a message DECISION which contains the decision value (lines 36 and 16). *Reliable Broadcast* guarantees that (1) all correct processes deliver the same set of messages, (2) all messages broadcast by correct processes are delivered, and (3) no spurious messages are ever delivered [5]. Otherwise, the current coordinator p_i proceeds to the next round (line 40).

As previously indicated, this framework extends CT. More precisely, the original CT algorithm can be obtained by considering the instantiation defined in Table 2. So, the framework inherits the *locking property* from CT. This property is the following one. As soon as a majority of processes have positively acknowledged a new estimate new_est sent by the current coordinator, no other value can be decided. This means that henceforth the current value of new_est will be the decided value: this value is locked. Whatever is the round during which a process decides, the locked value is the decided value.

Parameter	Description of the function
GET	Return the initial value v_i proposed by p_i
$\prec$	$\forall v_i, \perp \prec v_i$
$\mathcal{F}$	Select (in the array est_from) a value $\neq \perp$
ACCEPTABLE	Always return *true*
EXCUSED	Always return *true*

Table 2: Solving the Consensus Problem

[4]Since a process can only send more and more significant values (line 10), the test **not**($est \prec est\_from_i[j]$) performed at line 27 is useless when channels are FIFO.

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 agreement protocol (Figure 2) 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.

- The GET function. 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 described in Figure 3 (GET1 function). In this case, the input of p_i to GAF is simply the value of its current triple $< J_i, L_i, suspected_i >$ (where $suspected_i$ is the current value provided to p_i by the underlying failure detector).

```
Function GET1()
begin
    return(J_i, L_i, suspected_i) % query Failure Detector %
end
```

Figure 3: The GET1() Function

- 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. $\forall(< J, L, S >, < J', L', S' >)$:

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

- $\mathcal{F}$: 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 of the general framework 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.

The previous GET1 function 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 the GET1 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, as no acceptable value can be output, processes execute extra rounds and continue to query their GET1

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. The GET2 function described in Figure 4 realizes this property.

So, GAF provides the same result whatever the GET function is used (the one of Figure 3 or the one of Figure 4). However, these two instantiations of GAF have different behaviors (one is quiescent, the other is not).

```
Function GET2()
begin
 (1) wait until ((J_i ∪ L_i ∪ suspected_i) ≠ ∅) % wait for a meaningful value %
 (2) return(J_i, L_i, suspected_i)
end
```

Figure 4: The GET2() Function (it ensures Quiescence)

Another implementation of the GET function consists in limiting the "inaccuracy" of the set provided by the unreliable failure detector. Let us remind that, even if the failure detector works correctly, it only ensures there is one correct member that is not suspected[5]. A non-crashed process, suspected by a majority of processes, can erroneously be removed from the group. A way to reduce this unaccuracy of the underlying failure detector, consists for the processes in exchanging their sets of suspected processes during the execution of the GET function in order to compute an approximation of the transitive closure. This approximation is used to define a "better" set S of suspected processes.

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

6 Conclusion

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. A group is a set of processes that cooperate in carrying out a common task. 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 implement the group is called the current view of the group.

This paper has 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. Several instantiations of the GET function of the GAF framework have been designed and their respective advantages have been identified.

[5]This assumption is required for the GAF protocol to compute a single decision value [5].

References

[1] Aguilera M.K., 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, Ithaca, 1995.

[3] Bartoli A., Reliable Distributed Programming in Asynchronous Distributed Systems with Group Communication. *Tech Report*, Dipartimento di Informatica, Universitá di Trieste (IT), September 1999.

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

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

[6] Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[7] Chandra T.D., Hadzilacos V., Toueg S. and Charron-Bost B., On the Impossibility of Group Membership. *Proc. 15th ACM Symposium on Principles of Distributed Computing (PODC'96)*, pp. 322-330, Philadelphia (PA), May 1996.

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

[9] Cristian F., Synchronous and Asynchronous Group Communication. *Communications of the ACM*, 39(4):88-97, 1996.

[10] Dolev D., Dwork C. and Stockmeyer L. On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[11] 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, April 1985.

[12] Fritzke U., Ingels Ph., Mostefaoui a. and Raynal M., Fault-Tolerant total Order Multicast to Asynchronous Groups. *Proc. 17th IEEE Symposium on Reliable Distributed Systems, (SRDS'98)*, pp. 228-234, Purdue (IN), Oct. 1998.

[13] Guerraoui R., Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus. *Proc. of the 9th Int. Workshop on Distributed Algorithms (WDAG'95)*, Springer-Verlag, LNCS 972, pp. 87–100, September 1995.

[14] Hadzilacos V. and Toueg S., Reliable Broadcast and Related Problems. In *Distributed Systems (Second Edition)*, ACM Press, New-York, pp. 97-145, 1993.

[15] Hurfin M. and Raynal M., A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector. *Distributed Computing*, 12(4):209-223, 1999.

[16] Hurfin M., Macêdo R., Raynal M., and Tronel F., A General Framework to Solve Agreement Problems *Proc. 18th IEEE Int Symposium on Reliable Distributed Systems (SRDS'99)*, pp. 56-65, Lausanne, 1999.

[17] Kal L. and Hadzilacos V., Asynchronous Group Membership with Oracles. *Proc. of the 13th Symposium on Distributed Computing (DISC'99)*, Springer-Verlag, LNCS #1693, pp. 87–100, Bratislava (SL), September 1999.

[18] Larrea M., Arevalo S. and Fernandez A., Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. *Proc. 13th Symposium on Distributed Computing (DISC'99), (Formerly WDAG)*, Bratislava (Slovakia), Springer Verlag, LNCS #1693, pp. 34-48, (P. Jayanti Ed.), September 1999.

[19] Mostefaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Symposium on Distributed Computing (DISC'99), (Formerly WDAG)*, Bratislava (Slovakia), Springer Verlag, LNCS #1693, pp. 49-63, (P. Jayanti Ed.), September 1999.

[20] Neiger G., A New Look at Membership Services. *Proc. 15th ACM Symposium on Principles of Distributed Computing (PODC'96)*, pp. 331-340, 1996.

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

[22] Raynal M., Non-Blocking Atomic Commitment in Distributed Systems: A Tutorial Based on a Generic Protocol. *Journal of Computer Systems Science and Engineering*. Vol. 14(6), November 1999.

[23] Raynal M. and Tronel F., Group Membership Failure Detection: A Simple Protocol and its probabilistic Analysis. *Distributed Systems Engineering Journal*, Vol. 6, December 1999.

[24] Ricciardi A. and Birman K., Using Process Groups to Implement Failure Detection in Asynchrnous Environments. *Proc. 10th ACM Symposium on Principles of Distributed Computing (PODC'91)*, pp. 341-352, 1991.

[25] Schiper A., Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10:149-157, 1997.

[26] Vitenberg R., Keidar I., Chockler G.V. and Dolev D., Group Communication Specifications: A Comprehensive Study. *Tech. Report*, Lab of Computer Science, MIT, September 1999.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399