Mobile Groups

Raimundo José de Araújo Macêdo, Flávio Morais de Assis Silva

Laboratório de Sistemas Distribuídos – LaSiD
Departamento de Ciência da Computação
Universidade Federal da Bahia
Campus de Ondina, CEP: 40170-110, Salvador-BA, Brazil
{macedo, fassis}@ufba.br

Resumo

Comunicação em grupo tem sido largamente aceita como um meio efetivo de se construir aplicativos distribuídos confiáveis [5]. Sistemas de grupos tradicionais são baseados em processos estáticos [4, 5, 6, 7, 8]. Entretanto, processos estáticos não são a única forma de se estruturar tais aplicações. Atualmente, processos migrantes, que podem mudar de localização na rede durante a execução, têm sido propostos como uma forma de se projetar aplicações distribuídas. Analogamente às aplicações com processos estáticos, aplicações baseadas em processos migrantes também necessitam de formas confiáveis de cooperação entre os processos. Com o objetivo de suprir parte dessas necessidades, apresentamos neste artigo o conceito de grupos móveis. Assim como nos sistemas de grupo tradicionais, grupos móveis também oferecem garantias de entrega de mensagens e uma forma de sincronia virtual. No entanto, grupos móveis oferecem essas garantias apesar da mobilidade dos seus membros. Mais ainda, eles tornam a mobilidade de processos não somente visível para o grupo, mas também ordenam esses eventos de uma forma coerente em relação a outras ações do grupo, tais como falhas de processos, entradas e saída de membros. Neste artigo, definimos formalmente grupos móveis, especificamos as propriedades de um protocolo de *membership* para tais grupos, apresentamos o protocolo e provamos sua correção.

Abstract

Group communication has been proved as an effective abstraction for constructing reliable distributed applications [5]. Traditional group communication systems [4, 5, 6, 7, 8] are based on static processes. However, static processes are no longer the unique way of structuring distributed applications. Currently, some form of *migrating process*, i.e., a process that can change its location in the environment during its execution, is being frequently proposed as a basic component for designing distributed applications. Similarly to distributed applications based on static processes, applications based on processes that can migrate also need forms of reliable cooperation between processes. In order to fulfil part of this requirement, we present the concept of *mobile groups*. Analogously to traditional group systems, mobile groups also provide message delivery guarantees and a sort of virtual synchrony. However, mobile groups provide these guarantees *despite the mobility of their members*. Furthermore, they make process mobility not only visible for the group, but also consistently ordered with other group actions (such as process crashes, joins and leaves). In this paper we formally define mobile groups, specify the properties of a membership protocol for such groups, present the protocol itself, and prove its correctness.

Keywords: Distributed Systems, Group Communication, Fault Tolerance, Mobile Agents, Mobile Groups

1. Introduction

Group communication has been proved as an effective abstraction for constructing reliable distributed applications [5]. A *group* is a set of *processes* that communicate with each other by exchanging messages which are multicasted to the whole group. In order to preserve consistency among the state of group members, the group communication protocols must guarantee certain properties such as atomic delivery (either all processes deliver a message or no one deliver it), message ordering guarantees (e.g., causal and total), and a sort of virtual synchrony where modifications on the group membership (caused by events such as process crashes and joins, etc.) are consistently ordered with respect to message delivery. In such an environment, operational processes perceive the same sequence of actions, though, in reality, they may happen in an arbitrary order.

Traditional group communication systems, such as Horus [9], Transis [4], and Newtop [6], are based on *static* processes. After a member joins a group, it remains at the same location in the distributed environment during its whole life time. However, static processes are no longer the unique way of structuring distributed applications. Currently some form of *migrating process,* i.e., a process that can change its location in the environment during its execution, is being frequently proposed as a basic component for designing distributed applications. An example of a type of migrating process that has attracted the attention of researchers in the last years are the *mobile agents.* The mobile agent concept is being proposed to support different types of applications, including electronic commerce [18, 23], workflow management [24], network management [25], implementation of telecommunication services [16], distributed information retrieval [25] and active networks [25].

Similarly to distributed applications based on static processes, applications based on processes that can migrate also need forms of reliable cooperation between processes. In order to fulfil part of this requirement, we present the concept of *mobile groups.* Mobile groups are an extension of the traditional concept of groups that supports *migrating processes* as members of a group. With mobile groups, a migrating process has the ability to change its location in the distributed environment *while belonging to a group*. Analogously to traditional group communication systems, mobile groups also provide message delivery guarantees and a sort of virtual synchrony. However, mobile groups provide these guarantees *despite the mobility of their members*. Furthermore, they make process mobility not only visible for the group, but also consistently ordered with other group actions (such as process crashes, joins, and leaves). An implementation of mobile groups by using conventional group systems is not satisfactory because process mobility would be hidden from the group actions history.

Elsewhere [21], we have introduced the concept of mobile groups in the context of a reliability requirement discussion for mobile agent systems. This paper adds to our previous work, by formally defining mobile groups, specifying the properties of a membership protocol for such groups, presenting the protocol itself, and proving its correctness, i.e., that the protocol satisfies the defined properties.

The remaining of the paper is structured as follows. Section 2 presents an application scenario for mobile groups. Section 3 discusses the most closely related work. In section 4 we present the system model and our assumptions. Section 5 presents our membership protocol for mobile groups. Finally section 6 concludes the paper.

2. A Mobile Group Application Scenario

The following example illustrates the use of mobile groups to support the coordination of mobile agents in an electronic commerce scenario.

A company's employee wants to arrange a business trip. He would like to start an automated process to carry out the arrangement from his computer, a notebook that he uses to do his job. As part of the process, it is necessary to make a flight reservation. According to the policies of the user's company, the flight companies by which the user can make reservations are classified in priority classes. In order to make the reservation, a minimum number of price offers must be compared, say, from three flight companies. While collecting prices, the flight companies' priorities must be taken into consideration.

The software used to automate the trip arrangement is based on mobile agents[1]. In general terms the flight reservation could proceed as follows. First three agents are created. The list of flight companies and their priority classes are given to these agents. The three agents then move each to a flight company with the highest priority (assume, for simplicity, that there are at least three flight companies with the highest priority). Each agent verifies if the company can make the reservation according to the user's requirements and checks the price of the flight ticket. When an agent obtains an offer of a company, it sends this offer's information to one of the three agents, the *coordinator*. The coordinator will collect the prices, will select the cheapest one, and will inform the other agents about the result. The agent that sent the offer selected by the coordinator will commit the reservation.

The agents, however, must be able to react on failures and exceptions during the execution of this process. For example, if the coordinator fails, another agent must be chosen to play the coordination rule. Also, if an agent does not find an appropriate offer by a flight company, it might move to another company of the list.

Mobile groups are thus a suitable abstraction to support the coordination between the agents in this scenario, i.e., to enable the agents to react on failures and to act consistently. The set of agents used in the business trip arrangement will form a mobile group. Each agent represents a process of the group. With the support provided by mobile groups, at any time each agent taking part in the group will have a consistent view of the set of agents that mutually consider each other operational. If the coordinator fails, the other agents will eventually know about this. A new coordinator can then be elected by applying some predefined rule on the set of operational agents (for example, distinct priorities might be assigned to agents and the agent with the highest priority in this set will be the next coordinator). The new coordinator can then, for example, create a new agent to complete the set of three agents looking for offers. Additionally, since mobile groups provide also a consistent view about the location of agents while they move, when an agent decides to look for offers of a company, it will not move to companies it knows were already visited by other agents of the group.

3. Related Work

Mobile groups are an extension of the traditional concept of process groups. In traditional process groups, such as Horus [9], Transis [4] and Newtop [6], processes are static, i.e., they remain at the same place in the distributed environment during its whole life time. Movement of a process in traditional group communication systems could be implemented by making the

[1] We are not going to argue in this paper on the applicability of mobile agents to this example. The rea'der can find reasons for using mobile agents in scenarios like this elsewhere, for example, in [15].

moving process leave the group before the movement and join it again after the movement. The group communication system, however, would recognize these operations (leaving and joining the group) as two distinct operations for two different processes, since in those group systems, when a process joins a group, it obtains a new identification. In mobile groups, a process can migrate from a place in the distributed environment to another, while belonging to the group. The mobile group system recognizes the movement action for the same process, before and after the movement. Considering the movement as a single group operation allows a more efficient implementation of the group service and makes possible the synchronization of messages with relation to migration. Using traditional group communication protocols on top of a layer providing process migration transparency (as, for example, provided by systems such as Voyager [19]) would not provide a satisfactory solution either, since process mobility would be hidden from the group service, making it cumbersome to implement some functionality such as synchronization of messages with relation to movement.

Previous work incorporated process movement in group communication services in the context of mobile computing (environments with mobile devices). Some examples of work in this context are: algorithms for causal ordering [26, 27, 28]; algorithms for atomic multicast [29]; causal and total ordering of messages [30]; and a membership service [31, 32]. In these environments processes start and terminate their executions on the same host. However, since a host may be mobile, a process might change its (physical) location in the environment when the host where it is moves. In mobile groups we are considering that the hosts where processes are running are not mobile, but migrating processes can move from a host to another. Both problems are similar in the aspect that a process of a group is changing its location in the distributed environment. They are different, however, in other aspects, such as scalability (we can expect much more mobile agents, for example, than mobile devices) and in the way messages are routed to the recipients (routing handled at the application level, in the case of mobile agents, and in the network level, in the case of mobile devices). The algorithms proposed for group membership in mobile environments, however, do not tolerate host failures [31, 32] or would restrict the possibility of exploring locality if adapted to be used with mobile agents [32] (the algorithm in [32] is based on a static server with which the group members must communicate even if the whole group has moved occasionally to a local network located far from the server).

In the context of mobile agent systems, different forms of interaction between agents were proposed. Existing mobile agent systems (e.g., Voyager [19] and Aglets [33]) provide currently many forms of communication between agents (Remote Method Invocation, events, unreliable multicasts, etc.). In [34] the authors extend Linda to integrate mobility. In [35] a concept for coordinating groups of agents is described, but which is not fault tolerant. To the best of our knowledge no previous work exists that described a concept for supporting guarantees such as the ones provided by traditional group communication systems to mobile agents.

4. System Model and Assumptions

We assume a distributed system as a collection of mobile and static *processes, locations* and *communications channels*. A location represents a logical place in the distributed environment where processes execute. When a mobile process migrates, it moves from a location to another. Due to its movement capability, a process may be at different locations at different time instants. Processes at different locations communicate by exchanging messages through the communications channels.

Let $L = \{ l_1, l_2, ..., l_n \}$ denote the set of all possible locations. Let P be the set of all possible processes. A mobile group is denoted by the set of processes $g = \{p_1, p_2, ..., p_n\}$, $g \subseteq P$. On a mobile group, four operations are defined:

- *join(g, p):* issued by process *p*, when it wants to join group *g* ;
- *leave(g, p)*: issued by process *p*, when it wants to leave group *g*;
- *move (g, p, l):* issued when a mobile process *p* wants to move from its current location to location *l*;
- *send(g, p, m)*: issued by process *p* when it wants to multicast a message to the members of group *g*.

Each functioning process *p* of a mobile group has an associated *Group Service* which is in charge of assessing the group membership modifications and message delivery. We distinguish between a message being *received* by a *Group Service* through the communication channel and a message being *delivered* to *p* by the Group Service. The delivery of a message to a process can be delayed by the Group Service to satisfy synchronization requirements enforced by the system. Messages are delivered to processes according to the view synchrony semantics, presented in section 5.

Besides receiving messages, a process *p* also *installs views*. Views in mobile groups will reflect not only group membership, but also the locations where the processes in the group are. During its life time, a process *p* may install multiple views. Each view is associated with a number which increases monotonically with group view installations.

A view represents a mapping between processes of group *g* and locations of the distributed environment. Let $C(g) = \{ (p, l) \mid p \in g \text{ and } l \in L \}$ denote the cartesian product of *g* and *L*. We use $v_j^i(g)$ to denote the view number *i* of *g* installed by process p_j. A view $v_j^i(g) \subseteq C(g)$ is a subset of $C(g)$ such that there is an element $(p, l) \in v_j^i(g)$ if *p* is in *g* at location *l* when $v_j^i(g)$ was established.

We consider that a process *p* installs a view when actually the Group Service associated with *p* installs the view on behalf of *p*.

When the distinction between groups is not relevant for the understanding, the parameter that identifies the group will not be used in the notations. That is, the operations on a group and views will be denoted *join(p), leave(p), move (p, l), send(p, m), and* v_j^i.

We consider the communication channels to be reliable, i.e., message delivery in sequential order (FIFO order) is guaranteed. We assume that message transmission and processing times cannot be accurately estimated (i.e., an asynchronous system) and that processes fail only by crashing, i.e., by stopping functioning without producing any further action.

As will be seen later in this paper, our group membership protocol is based on a consensus module in order to reach agreement on the new views to be installed by operational group members. The consensus problem can be informally defined in the following way. Each process proposes a value, and all fault-free processes have to agree on a common value which has to be one of the proposed values. The Consensus problem constitutes a basic building block on top of which solutions to practical agreement problems can be designed. However, solving this problem in asynchronous distributed systems where processes can crash is far from being a trivial task. More precisely, it has been shown by Fischer, Lynch and Paterson [3] that there is no (deterministic) solution to this problem as soon as processes (even only one) may crash. The major advance proposed to circumvent this impossibility result lies in the

Unreliable Failure Detector concept, proposed and investigated by Chandra, Hadzilacos and Toueg [1, 2]. The weakest conditions that have to be satisfied to solve the consensus problem have been identified [2] and, accordingly, several protocols have been proposed to solve the consensus problem [1, 10, 11, 12, 14].

A failure detector is a sort of distributed oracle which gives (unreliable) hints about the state of processes. It is basically defined by two properties: a *completeness* property that is on the actual detection of failures, and an *accuracy* property that limits the mistakes a failure detector can make. Chandra and Toueg have defined several *completeness* and *accuracy* properties that allowed them to define eight classes of failure detectors. Among them, the class ◊S is the most attractive since it imposes the weakest conditions on the run time environment. This class includes all the failure detectors that *satisfy strong completeness* (eventually, every crashed process is suspected by every correct process), and *eventual weak accuracy* (there is a time after which there is a correct process that is never suspected). Ways to implement the failure detector ◊S have been presented elsewhere [13, 36].

As our group membership service makes use of the ◊S consensus protocol as specified by Chandra-Toueg in [1], we assume in our system model the existence of a distributed ◊S failure detector. Furthermore, we also assume that a majority of processes of a group view does not crash as required by the ◊S Consensus protocol. That is, if views $v_k^i(g)$ and $v_k^{i+1}(g)$ are installed by process p_k, then $v_k^{i+1}(g)$ will include the majority of processes of view $v_k^i(g)$.

We will use the primitive FD[p] to inquire the failure detector ◊S. Thus, FD[p] = *true* means that process p was suspected by the local failure detector module.

For interacting with the ◊S Consensus module, we make use of the primitives *Propose(input-value)* and *Decide(outcome-value)*. The first allows us to propose a given value to the consensus, and the latter will return the outcome from the consensus.

5. Mobile Group Membership

The Group Membership Problem consists traditionally of determining the set of group members which are operational in a given instant of the group existence. This set, called the group view, can change dynamically on the occurrence of process crashes or when processes deliberately leaves and joins the group. Every time a change occurs in the group membership view, a new view is installed at every (operational) process member. The main challenge of the group membership protocol is to ensure that each group member installs an identical sequence of views. In other words, the group members will perceive the evolution of the group membership in a mutually consistent way.

In mobile groups an additional event causes the generation of a new view for the group: the movement of a process. Ensuring now that each group member will install an identical sequence of views will provide the group members with a mutually consistent view of the group in relation to process crashes, joins, leaves *and movement operations*. Messages sent to the group will be synchronized in relation to all these events.

5.1. The Mobile Group Membership Properties

We consider that when a mobile group g is created, every group member p_k installs an initial view $v_k^1 = \{ (p_1, l_1), (p_2, l_2), ..., (p_n, l_n)\}$. For each pair $(p_i, l_i) \in v_k^1$, l_i denotes the location in the distributed environment where p_i is. After the initial view is installed, any modification on the structure of the mobile group (migrations, addition or deletion of members) will result in new views being installed, forming the sequence v_k^1, v_k^2,..., v_k^m where m represents a given moment on the view evolvement history.

Process p multicasts messages only to those processes of its current view. A process p that does not belong to a group g communicates with g by sending messages to one of the group members.

In this paper, we will only consider the so-called primary partition membership [5] where a unique sequence of views is installed for a given mobile process group. Primary partition membership is convenient, for example, for implementing fault-tolerant migration of mobile agents with mobile groups. The existence of concurrent views which is dealt with by partitionable membership protocols [4, 6] will be explored in future works.

View Properties

Let $v(g) = \{(p_1, l_1), (p_2, l_2), ..., (p_n, l_n)\}$ be a view of group g. We will denote by $v(g).P$ the set of processes that occur in $v(g)$, i.e., $v(g).P = \{ p \mid (p, l) \in v(g)$, for some $l \in L\}$. In order to simplify the notation, we will say that a process $p \in v(g)$ iff $p \in v(g).P$.

Let $g = \{ p_1, p_2, ..., p_n \}$ be a mobile group. The views installed by processes belonging to g must obey the safety and liveness properties defined below.

Safety Properties

Validity01 : if a process $p_j \in g$ installs a view $v_j^i(g)$, then $p_j \in v_j^i(g)$.

Validity02 : if a process $p_j \in v_k^i(g).P - v_k^{i-1}(g).P$, $i > 1$, then p_j asked to join g.

Validity03 : if a process $p_j \in v_k^{i-1}(g).P - v_k^i(g).P$, $i > 1$, then p_j asked to leave g or it has been suspected of crashing by some group member.

Validity04 : if the pair $(p, l') \in v_k^i(g)$ and $(p, l) \in v_k^{i-1}(g)$ and $l \neq l'$, then p asked to move from l to l'.

Validity01 states that only the members of a group view install the corresponding view. Validity02, Validity03, and Validity04 state that modifications on the group view are justified only by joins, leaves, crashes or crash suspicions, and movements.

Agreement : if a process $p_j \in v_j^i(g)$ installs $v_j^{i+1}(g)$ and a process $p_k \in v_j^i(g)$ also installs $v_k^{i+1}(g)$, then $v_j^{i+1}(g) = v_k^{i+1}(g)$.

Unique sequence of views : Let v_i^k and v_j^k be the view of number k installed by p_i and p_j, respectively (view number $k = k^{th}$ view installed only for the processes which installed the initial group view). Then, v_i^k is necessarily equal to v_j^k. In other words, $\forall\ k, i, j$ if p_i and p_j install views v_i^k and v_j^k, then $v_i^k = v_j^k$.

Unique sequence of views is a necessary condition for the so-called primary component membership where only one component of the group is allowed to make progress (i.e., groups are not allowed to split into, disjoint, subgroups). In order to enforce this behavior it is required that the majority of processes in a view v_j^i assent in the composition of view v_j^{i+1}. Also, observe that agreement, as defined above, does not necessarily implies an unique sequence of views since processes may join the group later, without installing the initial group view v^1.

Liveness Properties

Termination01 : if a process $p_j \in v_k^i(g)$ leaves g or crashes and some group member remains operational, then there will be at least one operational process p_h of g that installs $v_h^l(g)$, $l > i$, such that $p_j \notin v_h^l(g)$.

Termination02 : if a process p_j asks to join g and p_j and some group member of g remain operational then there will be at least one operational process p_k of g that installs $v_k^i(g)$ such that $p_j \in v_k^i(g)$.

Termination03 : if a process p_k of a pair $(p_k, l) \in v_k^i(g)$ asks to move to location l' and it is not excluded from the group, then there will be at least one operational process $p_h \in v_k^i(g)$ that installs $v_h^{i+r}(g)$ such that $(p, l') \in v_h^{i+r}(g)$ and $(p, l) \in v_h^{i+r-1}(g)$, $r > 0$

Message Delivery Properties

A message m *received* by the network transport layer is stored in the group system's local buffers and is delivered only when the delivery properties can be satisfied. If $g = \{p_1, p_2, ..., p_n\}$ is a mobile group, let $deliver(m, p_i, k)$, $p_i \in g$ denote the delivery of a message m to process p_i in view $v_i^k(g)$.

We define the following safety and liveness message delivery properties for our mobile process groups:

Safety Property

Consider processes p_i and p_j, $p_i \in g, p_j \in g$:

MD1 (ATOMICITY): If p_i installs views $v_i^k(g)$ and $v_i^{k+1}(g)$ and p_j installs views $v_j^k(g)$ and $v_j^{k+1}(g)$, then $deliver(m, p_i, k) \Leftrightarrow deliver(m, p_j, k)$. This is the all or nothing message delivery property. That is, any two member processes of g that install two consecutive views, deliver the same set of messages between them.

As a result of MD1, any two member processes of g that never crash nor suspect each other, deliver the same set of messages between two consecutive views.

Liveness Property

MD2 (LIVENESS): If a process p_i sends m in view r, then provided it continues to function as a member of g, it will eventually deliver m in some view $v_i^{r'}$, $r' \geq r$.

Properties MD1 and MD2 together enforce a sort of virtual synchrony similar to the one initially proposed by Birman [5] and correctly formalized in [37]. Birman's virtual synchrony requires that $r = r'$ in the MD2 and can be implemented by blocking sending messages during the view installation procedure. Instead, our definition is more related to the ones proposed in Transis [4] and Newtop [6] systems which differ from ours by the fact that they were primarily intended to partitionable memberships.

5.2. The Mobile Group Membership Protocol

The group communication system consists of the group membership service, a multicast best-effort primitive, called *mcast*, a reliable multicast primitive, called *rmcast*, and a consensus service.

A process sends a message m to a group g by multicasting m to all members of g. The primitive $mcast(m,g)$ causes the delivery of m to g as long as the sender process does not crash. The sending process identifier is recorded in the multicasted message.

The primitive *rmcast(m,g)* will deliver a message *m* atomically to all members of group *g*. That is, if *m* is *rmcast* to *g* and at least a member of *g* delivers *m*, all processes of *g* will also deliver *m*. The primitive *rmcast* is more expensive to implement than *mcast*, since it must guarantee the all or nothing effect. A simple way of implementing *rmcast(m,g)* from a *mcast(m,g)* is as follows. Every member of *g* that receives *m* for the first time, relays *m* to *g* before delivering *m* [1].

As mentioned in section 2, our membership service is based on the execution of the ◊S Consensus protocol. Therefore, we assume the existence of the primitives *propose(input-value)* and *decide(outcome-value)* which are used to propose an input value for the consensus and decide the consensus outcome, respectively.

Handling Process Crashes

Consider a message *m* sent to a mobile group *g*. When *m* is received by a destination process p, $p \in g$, *m* is immediately delivered to *p* and stored in a local buffer called *unstable* until *m* is known to be stable (i.e., received by all processes in g). If a message remains unstable for too long, the membership service will start a new view installation procedure for removing possibly crashed processes from the current membership view and delivering the unstable messages not yet delivered. This procedure, which is carried out through a consensus protocol, guarantees that all correct[2] members deliver the same set of messages and the same sequence of views. Additionally, messages are delivered at operational processes in the same view. In other words, processes deliver the same set of messages between two consecutive views installed. In the example of figure 1, p_1 crashed before finishing the multicast of *m* in a way that p_2 has received (and delivered) *m* but not p_3. After suspecting the crashing of p_1, p_2 and p_3 will start a procedure of view installation to recover the missing message *m* and deliver it to p_3 before installing the new view ($k+1$) which does not include p_1.

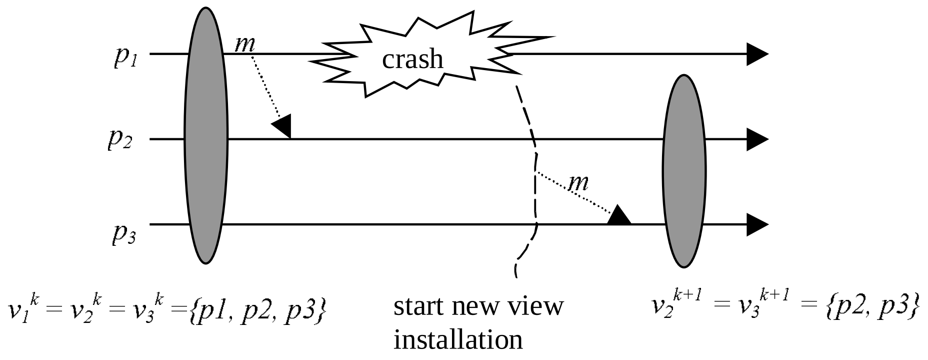
p_1
p_2
p_3
m
crash
m
$v_1^k = v_2^k = v_3^k = \{p1, p2, p3\}$
start new view installation
$v_2^{k+1} = v_3^{k+1} = \{p2, p3\}$

Figure 1. View Installation with Virtual Synchrony

As mentioned before, processes try to install a new view whenever one or more locally delivered messages remain unstable longer than a predetermined timeout period. To recover unstable messages, the group processes will engage themselves in a recovery phase where they exchange their unstable sets. Afterwards, the view installation will only progress if any of the operational group members fail in sending the corresponding unstable set (notice that a real crashed process that failed in sending the acknowledgement to a given message will also fail in sending the unstable set). Such processes – those which did not send the unstable set - are considered as crashed and a new view will be installed which does not include them. We

[2] A *correct* process is a process that does not crash neither is excluded from the group.

should bear in mind, however, that those suspected processes may be just too slow (due to a overloaded site or connection link, for instance). Although a false suspicion[3] may cause the removal of an operational process from a group, the membership service presented in this paper will do this removal in such a way that the membership information will always be kept mutually consistent among all the processes that are not suspected. Let us assume that all the processes maintain the set *unstable* to record the unstable messages. This set is a global variable initialised empty and updated through the message stability assessment task and the procedures of the membership protocol. *currentview* is a global set and represents that latest view installed by a process. That is, given a group *g*, *currentview* is formed by pairs (p_i, l_i), $p_i \in g$ and $l_i \in L$, where l_i is the current location of process p_i. Below the mechanism used by each group member to assess the message stability is presented.

Message Stability Assessment Task
(launched by a process p for every sent or received message m)

```
 set timeout(m)
 unstable ← unstable ∪ {m};  /* puts m into set unstable */
Wait until
    ∀q∈ currentview, received(m,ack) /*the receipt of m was acknowledged by all members */
    or
    expires timeout(m);
 if ∀q ∈ currentview,  received(m,ack)
   then
    cancel timeout(m);
    unstable ← unstable – {m};  /* removes m from the set unstable */}
  else
    begin
       k:= k + 1; /* increments the view counter */
       rmcast (changeViewRequest, k);  /* try to install a new view k */
   end
```

The membership protocol works by all group members executing the ◊S consensus on a new view to be established as well as the set of messages not yet known as being stable. After reaching consensus, each group process first delivers the unstable messages from the consensus execution that have not been delivered yet and then installs the new agreed view. Notice that there may exist several concurrent consensus executions, each one identified by the number *k* of the new view to be stabilized. Though we present here a novel protocol, the use of multiple consensus executions for solving agreement problems (such as static membership and atomic broadcast) have been presented in other works [1, 38].

The membership protocol working on behalf of a group member p_i consists of the task below that is activated whenever a message remains unstable longer than a predetermined timeout period. During the new view establishment process, all the group members are required to send their unstable sets and the new view will be formed by removing those processes which were suspected by the local failure detector FD, provided that a majority of the processes (whether suspected or not) have sent the unstable sets. Given a group *g* with the corresponding process locations, and a process $p_i \in g$, *newview* denotes a set formed by pairs (p_i, l_i), $p_i \in g$ and $l_i \in L$ $(newview.L[i] = l_i)$. The global variable *viewnumber* denotes the number of a view installed by a process p_i. *currentview* and *viewnumber* together indicate the

[3] Timeout values should be carefully chosen in order to make false suspicions rare.

process view in a given moment of a process view evolvement. For example, when $v_i^k = \{p_1, p_2\}$, then *viewnumber* = k and *currentview* = $\{p_1, p_2\}$.

ChangingView Task:

```
(1) Upon receiving (changeViewRequest, k) message
(2) DO block local delivery and mcast(unstable, k) if (unstable,k) has not been multicast yet.
/* collect the unstable messages from all non-suspected group members,
   provided that a majority of them has sent their unstable messages */
(3) Wait until [$\forall q \in$ currentview : received (unstable,k) from q or FD(q) = true]
                    and for $\lceil (n+1)/2 \rceil$processes q : received (unstable,k) from q
(4) allunstable := {message / message $\in$ unstable and received (unstable,k)}
(5) newview := currentview – {(q,l) / q$\in$ currentview.P and (unstable,k)
                                                   was not received from q}
/* run consensus k */
(6) Propose(k, allunstable, newview)
(7) Wait until decide(k, allunstabble, newview);
/* deliver the agreed unstable messages not delivered yet */
(8) Deliver any m $\in$ allunstable /m has not been delivered yet
/* install the new view k only if the current view was modified */
(9) If newview = currentview then exit; }/* it was a false suspicion */
/* install the new view k if $p_i$ still belongs to view k */
(10)If $p_i \in$ newview.P then {viewnumber :=viewnumber + 1; currentview := newview;}
    Else "communicate process $p_i$ that it has been removed from the mobile group"
(11)Unblock delivery
```

Handling Moves

A group process p_i willing to move to a new location l must issue the operation *move*(p_i,l). As a result, the mobile group service will first make sure that there are conditions to the required migration. If so, the location information of p_i is updated and the ChangingView task is started to try to install the new view with the new location for p_i.

As happens with crashes, joins, and leaves, a new process location is established in all group members by installing a new view through a consensus procedure. This consensus procedure is required since all the events (crash, joins, leaves, and message delivery) should be ordered with respect to the new process location installation. Below are the steps performed by the mobile group system in order to move a process p_i to a given location l.

Move(p_i, l); /* moves process p_i to location l ; */

```
(1) Enquire the remote location about the installation of a new process
(2) If movement is authorised than start the membership service on the remote place
    else {exit; return error code}
(3) currentview.L[i] := l;
(4) k : = k + 1;
(5) rmcast(ChangeViewResquest, k); /* try to install a new view */
(6) await ChangeView task to finish;
(7) if $p_i \in$ currentview.P and currentview.L[i] = l
    then
     { migrate $p_i$ to remote location
       install current view at remote location
      forward to the remote location all the received but not delivered messages
      Finish the local $p_i$ }
```

Due to space limitations we will not present in this paper the operations and correctness proofs connected with process joins and leaves. The proofs and operations for joins and leaves are quite similar to the ones presented here and can be found in the complete version of our work [22].

Correctness of the Protocol

Proofs' sketch

In order to present the correctness of the membership protocol, we must show that the safety and liveness properties, namely, validity, agreement, unique sequence, termination, and the messages delivery properties are satisfied in any execution of the protocol. Below we show how all these properties are satisfied by the protocol composed by the message stability assessment and ChangeView tasks, and the move operation actions. The letters C and M will be used to indicate the statement lines of the ChangeView task and move operations, respectively (e.g., C10 stands for statement line number 10 of ChangeView task).

Validity01 : if a process $p_j \in g$ installs a view $v_j^i(g)$, then $p_j \in v_j^i(g)$.

Observe that, by assumption, there is an initial view that is installed by all processes that appear in it. Afterwards, modifications to the group membership (by modifying the global variable *currentview*), are carried out only under the condition that the process which installs the new view belongs to it (C10).

Validity03 : if a process $p_j \in v_k^{i-1}(g).P - v_k^i(g).P$, $i > 1$, then p_j asked to leave g or it has been suspected of crashing by some group member.

If $p_j \in v_k^{i-1}$(g).P – v_k^i(g).P}, i > 0, is because process p_j was removed from the group membership before view $v_k^i(g)$ was installed. On the other hand, a new view is formed only by modifying the variable *newview* (C5) which is latter subjected to the consensus procedure in the body of the ChangingView task (C6). The consensus outcome for *newview* (which by definition is proposed by one of the processes of v^{i-1}(g)) defines then the new set of processes of the group (C7 and C11). Finally, notice that the *newview* maintains all the processes of *currentview* except those which are suspected of crashing by some group member (i.e. have not sent the corresponding unstable set). Therefore, if p_j is not in $v_k^i(g)$, it is because it has been suspected by some group member. The proof for leave is analogous.

Validity04 : if $(p, l') \in v_k^i(g)$ and $(p, l) \in v_k^{i-1}(g)$ and $l \neq l'$, then p asked to move from l to l'.

Verification of validity04 is trivial from the execution of *move*(p, l') operation which changes the location of the process before launching a new view installation (M3).

Agreement : if a process $p_j \in v_j^i(g)$ installs $v_j^{i+1}(g)$ and a process $p_k \in v_j^i(g)$ also installs $v_k^{i+1}(g)$, then $v_j^{i+1}(g) = v_k^{i+1}(g)$.

Agreement can be proved by induction on the numbers of view installed during a group g lifetime. First notice that by assumption all processes install the same initial view 1. Now let us suppose by induction hypothesis that all processes installed view i, $i > 1$, respecting the agreement property. That is, if a process $p_k \in v_k^{i-1}(g)$ installs the view $v_k^i(g)$ and another process $p_r \in$ view $v_k^{i-1}(g)$ installs view $v_r^i(g)$, then $v_r^i(g) = v_k^i(g)$ for any p_k and $p_r \in v_k^{i-1}$(g). Now, we must show that $v_j^{i+1}(g) = v_k^{i+1}(g)$.

The updating of the variables *viewnumber* and *currentview* reveals the establishment of a new view (C10). As *viewnumber* is initialized with value 1 (when the initial view is set) and it is increased by 1 for every new view installed, when $v_j^i(g)$ and $v_k^i(g)$ are established, the values of *viewnumber* for processes p_j and p_k will be updated to i. The new view v^{i+1} is then formed when the value of *viewnumber* is updated to $i+1$ and *currentview* to the set *newview* in line C10.

In order to change the view from $v^i(g)$ *to* $v^{i+1}(g)$, processes p_j and p_k will construct their own *newview* sets (by collecting the identifiers of the processes which the unstable sets have been received from[4] (C5)) which they propose as their input values for the ◊S consensus protocol. The view v^{i+1} will be formed and installed only after the processes in view v^i have decided the outcome of the corresponding consensus (C6 and C7). As the agreement property of the ◊S Consensus protocol certifies that every process decides for the same set *newview*, all processes that install view v^{i+1} will install the same view.

Unique sequence of views : Let v_i^k and v_j^k be the view of number k installed by p_i and p_j, respectively (view number $k = k^{th}$ view installed only for the processes which installed the initial group view). Then, v_i^k is necessarily equal to v_j^k. In other words, $\forall\ k, i, j$ if p_i and p_j install views v_i^k and v_j^k, then $v_i^k = v_j^k$.

Suppose by contradiction there are 2 processes p_1 and p_2 that install different sequences of view and take the first view i of the sequence such that these views are different, for any $i > 0$. The agreement property proved above guarantees that process view i of p_1 and p_2 are equal.[5]

Livenees Properties

Termination01 : if a process $p_j \in v_k^i(g)$ leaves g or crashes and some group member remains operational, then there will be at least one operational process p_h of g that installs $v_h^l(g)$, $l > i$, such that $p_j \notin v_h^l(g)$.

If p_j crashes, it will not acknowledge messages sent to g and therefore the ViewChange task of a operational process p_h will eventually receive the *rmcast(changeViewRequest, k)* issued by the message stability assessment task of p_h. By definition, *rmcast* will guarantee that all operational group members receive the change_view request and, thus, start the membership service. Since by assumption the majority of processes of $v^i(g)$ does not crash and will send the corresponding ($i+1$, unstable) message, the protocol eventually finishes C3 and define the new view in C5. Finally, notice that the variable *newview* (which is used to set a new view in C10) will not include p_j since it will not send the required ($i+1$, unstable) message.

Termination03 : if a process p_k of a pair $(p_k, l) \in v_k^i(g)$ asks to move to location l' and it is not excluded from the group, then there will be at least one operational process $p_h \in v_k^i(g)$ that installs $v_h^{i+r}(g)$ such that $(p, l') \in v_h^{i+r}(g)$ and $(p, l) \in v_h^{i+r-1}(g)$, $r > 0$

After executing the *move*(p_i, l) operation, the ChangeView Request will be received by all operational group members (as it is transmitted by a reliable multicast) which will in turn start a new view installation with the new location l of p_k (M5) Therefore, the new pair (p_k, l) will be part of the *newview* set (C5) in all operational group members (as p_k will send its unstable set). Therefore, if p_k is not excluded from the group during the view installation procedure (cause by some other parallel consensus execution), all operational group members will install the new view $v^{i+1}(g)$ with the new location of p_k.

[4] Notice that the *newview* set will contain the majority of such processes despite false suspicion from the failure detector FD (C3) since by assumption of the ◊S detector, the majority of processes of v^i does not crash.

[5] This proof is only valid if joins are not considered, as with joins processes may have different initial views.

Message Delivery Properties

Safety Property

Consider processes p and q belonging to g.

MD1 (ATOMICITY): If p_i installs views $v_i^k(g)$ and $v_i^{k+1}(g)$ and p_j installs views $v_j^k(g)$ and $v_j^{k+1}(g)$, then $deliver(m, p_i, k) \Leftrightarrow deliver(m, p_j, k)$. This is the all or nothing message delivery property. That is, any two member processes of *g* that install two consecutive views, deliver the same set of messages between them.

Consider a message *m* sent to a group *g*. Message *m* is sent with the *mcast* primitive. Thus, unless the sender process crashes during the multicasting of *m*, *m* will reach all destinations. By assumption, unless delivery is explicitly blocked by the membership service, a received message is immediately delivered after its arrival. Now consider the case when the *m* sender crashes in such a way that atomicity is violated. That is, some (but not all) processes will get the message and others will not. Without loss of generality, let process *p* be such a process which received *m*. As *m* sender have crashed, *m* will remain unstable long enough that *p* will try to install a new view to remove the *m*'s sender by executing the ChangingView Task. In the ChangingView task, every operational process will multicast its unstable set which may or not include *m* (notice that *m* will be present in the unstable set multicast by *p*). In the ChangingView task of an operational process, all the unstable messages will be collected and a new membership defined with the processes identifiers which the unstable sets were received from. These two sets, that is, the collected unstable set and the new membership, will be the input value of a given process for the consensus module. After that, the consensus module will decide for one of the proposed pairs, and all messages of the collected unstable set from the consensus outcome will be delivered in all operational processes of the old membership - before a new view is installed. Therefore, if a process delivered *m* in the old view and remains in the new view (established from the consensus), then *m* will be delivered to all processes of the new view before the referred new view is installed.

Liveness Property

MD2 (LIVENESS): If a process p_i sends *m* in view *r*, then provided it continues to function as a member of *g*, it will eventually deliver *m* in some view $v_i^{r'}$, $r' \geq r$.

First notice that *mcast* guarantees that, as long as the sender of *m* remains operational, *m* will reach all destinations. If no view changes occur, then m will be delivered in v_i^{r}(g). Suppose now that a view $v_i^{r'}$(g) was installed before the delivery of *m* and that process p_i belongs to the new view $v_i^{r'}$(g). So, if *m* was not delivered before, then either *m* will be part of the collected unstable set, and therefore will be delivered before the new view is installed, or it will be delivered after the new view is installed.

6. Conclusion

We have presented mobile groups as an extension of the traditional concept of process groups that support their members to migrate between locations of a distributed environment. Mobile groups provide a reliable form of coordination between migrating processes and thus fulfill part of the requirements that arise in current distributed applications. For example, for supporting reliability of mobile agent based applications in electronic commerce scenarios.

In this paper we formalized the properties of a membership service for mobile groups, presented a membership protocol and proved that it satisfies the defined properties. Mobile groups support a form of virtual synchrony in which messages are synchronized with not only crashes (suspicious), leaves and join operations of group members, but also with movement operations. The algorithm presented is based on a ◊S failure detector based consensus and it implements a primary partition membership protocol. In order to validate our model in broader environments, we are currently implementing the mobile membership protocol as defined in this paper on top of a ◊S based general framework intended to solve agreement problems in a efficient way [10]

To the best of our knowledge this is the first work that provides and formally defines a concept of group which supports moving processes by integrating the movement events inside the group communication protocols. By doing that, we were able to enforce semantics for synchronizing the delivery of messages with relation to movements, and, as a consequence, we can extend the functionality of the system. For instance, by defining alternative semantics for message delivery that recognizes a moving process (for example, total order delivery). That would not be achieved satisfactorily by using traditional group communication systems.

The presented concept is a first step towards developing a form of enabling mobile process coordination with broader applicability. It complements other efforts that provide reliability of mobile process based applications, such as the concepts for mobile agent fault tolerance and transactional support [15, 17, 20]. The concept presented in this paper is being implemented at LaSiD/DCC/UFBA as part of a reliable distributed platform that will be used to support, among other applications, the development of a reliable workflow management system.

7. References

[1] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, 43(2):225-267, March 1996.

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

[3] 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.

[4] Amir, Y., Dolev, D., Kramer, S., Malki, D. Transis: A Communication Subsystem for High Availability. In Proc. of the 22nd Int. Symp. on Fault-Tolerant Comp. pp. 76-84, Boston, July, 1992.

[5] Birman, K. The Process Group Approach to Reliable Distributed Computing. Communications of the ACM, Vol. 9, No. 12. pp. 36-53, December 1993.

[6] Ezhilchelvan, P., Macêdo, R., Shrivastava, S. Newtop: A Fault-Tolerant Group Communication Protocol. In Proc. of the IEEE 15th Int. Conf. on Dist. Comp. Syst. Vancouver, pp. 296-306, 1995.

[7] Kashoek, M, Tanenbaum, A.. Group Communication in the Amoeba Dist. Op. System. In Proc. of the Int. Workshop on Parallel and Distributed Systems, Vol.5, No.5, pp. 459-473, May, 1994.

[8] Melliar-Smith, M.P., Moser, L.E., Agarwala, V. Processor Membership in Asynchronous Distributed Systems. IEEE Trans. on Parallel and Distributed Systems, 5(5):459-473, May1994.

[9] Renesse, R., Birman, K., Cooper, R., Glade, B., Stephenson, P. The Horus System. In K. Birman e R. Renesse, editores, Reliable Distributed Computing with the Isis Toolkit, pp. 133-147. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[10] Hurfin, M., Macêdo, R., Raynal, M., Tronel, F. A General Framework to Solve Agreement Problems. Proc. of the IEEE Int. Symp. on Reliable Distributed Systems, SRDS'99, Lausanne. 1999.

[11] Badache, N., Hurfin, M., Macêdo, R. Solving The Consensus Problem In A Mobile Environment. Proc. of the IEEE International Performance, Computing, and Communications Conference – IPCCC'99, Phoenix/Scottsdale, USA: IEEE Press, 1999. p.29-35.

[12] Greve, F., Hurfin, M., Macêdo, R., Raynal, M. Consensus Based on Strong Failure Detectors : A Time and Message Efficient Protocol. Lecture Notes in Computer Science, v.1800, p.1258-1267, May/2000.

[13] Aguilera, M., Chen, W., Toueg, Using the heartbeat failure detectors for quiescent reliable communication and consensus in partitinable networks. Theoretical Comp Science, 220:3-30, 1999.

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

[15] Assis Silva, F.M.. A Transaction Model based on Mobile Agents. PhD Thesis. Technical University Berlin. 1999

[16] Magedanz, T., Popescu-Zeletin, R. Towards "Intelligence on Demand" - On the Impacts of Intelligent Agents on IN. in Proceedings of the 4th International Conference on Intelligence in Networks. Bordeaux, France. November, 1996

[17] Rothermel, K., Straßer, M. A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents. Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS'98). West Lafayette, USA. October, 1998. pp. 100-108

[18] Straßer, M., Rothermel, K., Maihöfer, C. Providing Reliable Agents for Electronic Commerce. in Trends in Distributed Systems for Electronic Commerce - International IFIP/GI Working Conference TREC'98. Springer. Berlin. 1998. pp.241-253

[19] ObjectSpace. Voyager – ORB 3.1 Developer Guide. Object Space, Inc. 1999

[20] Assis Silva, F.M., Popescu-Zeletin, R. Mobile Agent-based Transactions in Open Environments. IEICE Transactions on Communications. IEICE/IEEE Joint Special Issue on Autonomous Decentralized Systems. Vol.E83-B, No.5. Maio 2000

[21] Assis Silva, F.M., Macêdo, R.J.A. Reliability Requirements in Mobile Agent Systems. Anais do II Workshop de Testes e Tolerância a Falhas (II WTF). SBC. Curitiba. Julho 2000

[22] Macêdo, R.J.A., Assis Silva, F.M. Mobile Groups. Technical Report RI001/01. LaSiD/UFBA (Distributed Systems Laboratory / Federal University of Bahia). February, 2001.

[23] White, J.E. Telescript Technology: Scenes from the Electronic Marketplace. General Magic. 1994

[24] Cai, T., Gloor, P.A., Nog, S. Dartflow: A Workflow Management System on the Web using Transportable Agents. Technical Report PCS-TR96-283. Department of Computer Science. Dartmouth College. 1996

[25] Fuggetta, A., Picco, G.P., Vigna, G. Understanding Code Mobility. IEEE Transactions on Software Engineering. Vol.24, No.5. May, 1998

[26] Alagar, S., Venkatesan, S. Causal Ordering in Distributed Mobile Systems. IEEE Transactions on Computers. Vol. 46, No. 3. March, 1997

[27] Yen, L.-H., Huang, T.-L., Hwang, S.-Y. A Protocol for Causally Ordered Message Delivery in Mobile Computing Systems. ACM/Baltzer Mobile Networks and Applications. Vol. 2. 1997

[28] Prakash, R., Raynal, M., Singhal, M. An Efficient Causal Ordering Algorithm for Mobile Computing Environments. Proceedings of the 16th International Conference on Distributed Computing Systems. May 27-30, 1996, Hong Kong, IEEE Computer Society, 1996

[29] Endler, M. A Protocol for Atomic Multicast among Mobile Hosts. Proceedings of the 1st Brazilian Workshop on Wireless Communication (1o. Workshop de Comunicação sem Fio - WCSF) , UFMG/PRONEX. Belo Horizonte, Brazil. July 1999

[30] Anastasi, G., Bartoli, A., Spadoni, F. Group Multicast in Distributed Mobile Systems with Unreliable Wireless Network. Proceedings of the Eighteenth Symposium on Reliable Distributed Systems. Lausanne, Switzerland. October, 1999

[31] Prakash, R., Baldoni, R. Architecture for Group Communication in Mobile Systems. Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems. Indiana, USA. October, 1998

[32] Bartoli, A. Group-based Multicast and Dynamic Membership in Wireless Networks with Incomplete Spatial Coverage. Mobile Networks and Applications. Vol. 3. Baltzer Science Publishers. 1998

[33] Lange, D.B., Chang, D.T. IBM Aglets Workbench – Programming Mobile Agents in Java – A White Paper. IBM Corporation. September, 1996

[34] Picco, G.P., Murphy, A.L., Roman, G.-C. Linda Meets Mobility. Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles (USA), D. Garlan and J. Kramer (eds.). ACM Press. May, 1999

[35] Baumann, J., Radouniklis, N. Agent Groups in Mobile Agent Systems. Distributed Applications and Interoperable Systems (DAIS'97). H.König, K.Geihs, T.Preuá (eds.). Chapman & Hall. 1997

[36] Macêdo, R. Failure Detection in Asynchronous Distributed Systems.Proc. of II Workshop on Tests and Fault-Tolerance (II WTF 2000), pp. 76-81, July 2000, Curitiba, Brazil, Brazilian Computer Society.

[37] 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, USA, 1995.

[38] Guerraoui, R., Schiper, A. Consensus Services : A Modular approach for building agreement protocols in distributed systems. FTCS'99. IEEE.