ARTICLE IN PRESS

Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

J. Parallel Distrib. Comput. ▮▮▮ (▮▮▮▮) ▮▮▮–▮▮▮

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

The mobile groups approach for the coordination of mobile agents

Raimundo J. Araújo Macêdo*, Flávio M. Assis Silva

Distributed Systems Laboratory—LaSiD, Computing Science Department, Federal University of Bahia—UFBA, Campus de Ondina, Av. Adhemar de Barros, S/N, 40170-110, Salvador-BA, Brazil

Received 18 September 2003; received in revised form 14 July 2004

Abstract

We present the concept of mobile groups as a basic mechanism for the reliable coordination of mobile agents. Analogously to traditional group systems, mobile groups also provide message delivery guarantees and virtual synchrony. Furthermore, they make agent mobility not only visible for the group, but also consistently ordered with other group actions (such as crashes, joins, leaves, and other migrations). The mobile groups approach represents a novel mobility support mechanism, which can be used to handle reliability of mobile agents required at both, the application and system level (e.g., for coordinating distributed agents and for reliable agent migration, respectively). In this paper, we discuss the motivations for the mobile groups approach, formally define their properties, and present a membership protocol for such groups. We also discuss some implementation issues and related performance data, and present the advantages of mobile groups against mechanisms commonly employed for the coordination of mobile agents.
© 2004 Elsevier Inc. All rights reserved.

Keywords: Mobile agents; Reliable agent coordination; Fault tolerance; Group communication; Distributed algorithms

1. Introduction

The mobile agent concept has attracted the attention of many researchers in the last years and it is being proposed as a basic component for designing distributed applications, in areas including electronic commerce, workflow management, network management, and distributed information retrieval [22]. Essentially, a mobile agent is an agent that can change the node in which it is running, during its execution. In these systems, location awareness (i.e., where and when a given agent executes a given computation) plays an important role for the related applications [30].

Similarly to distributed applications based on static processes, applications based on agents that can migrate also need forms of reliable co-operation between agents. This need may appear at the application or at the system level. At the application level, for example, consider the case of a user who wants to find some resource in the Internet. A group of agents might be used to look for the resource in parallel. The actions of the agents, however, should be coordinated. For example, if any of them fails, another one might be created to complete a desired minimum set of agents looking for the resource. In order to coordinate the group, an *agent coordinator* should be elected. Additionally, each agent should move to locations that have not been previously visited by other agents in the group. In this scenario, at all stages of the computation, the group members need a mutually consistent view of both, the group configuration (i.e., the current group agents and their corresponding locations) and the relative order of certain events (such as agent crashes and message delivery). At the system level, coordination of replicas of an agent is, for example, a requirement for implementing mobile agent fault tolerance [6,33,40].

In order to fulfill the coordination requirements of such scenarios, we present the concept of *mobile groups*. Mobile groups are an extension of the traditional concept of process groups that supports *moving agents* as members of a group. Process groups have been widely used as a

* Corresponding author. Fax: +55 71 263 6145.
E-mail addresses: macedo@ufba.br (R.J.A. Macêdo), fassis@ufba.br (F.M. Assis Silva).

0743-7315/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.10.001

ARTICLE IN PRESS

mechanism for supporting consistent execution of sets of co-operating processes in distributed systems [1,10]. In a process group, processes communicate with each other by exchanging messages that are multicast to the whole group. In order to preserve consistency, the group communication protocols guarantee certain properties such as atomic delivery (either all processes deliver a message or no one delivers it), message ordering guarantees (e.g., causal and total), and 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 a mutually consistent ordered sequence of events, though, in reality, they may happen in an arbitrary order. In traditional group communication systems [3,7,11,19,38], while a process is a member of a group, it generally remains at the same location in the distributed environment. In mobile groups, an agent 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 virtual synchrony. However, mobile groups provide these guarantees *despite the mobility of their members*. Furthermore, they make agent mobility not only visible for the group, but also consistently ordered with other group actions (such as crashes, joins, leaves, and migrations). In other words, a mobile group configuration (or view) will reflect not only the set of members of a group but also their corresponding locations.

By combining conventional group communication operations (such as joins and leaves) with a move operation (used by an agent that wants to migrate) and a location-aware view installation procedure, mobile groups represent both a means for coordinating mobile agents and a mobile system infrastructure. Using traditional group communication protocols on top of a layer providing agent migration would not yield a satisfactory implementation for the mobile groups, since agent mobility would be hidden from the group service, making it cumbersome to implement some functionality such as synchronization of messages with relation to agent movement.

Mobile groups can be used, for instance, for implementing mobile agent fault tolerance, which consists basically of co-ordinating a set of agent replicas in such a way that if the replica currently executing the application fails, another replica resumes its execution. An usual requirement of such a mobile agent fault tolerance is to guarantee the so-called exactly once semantics (EOS) [6,25,40,33]. The EOS property has also been used in other contexts such as in message delivery [27] and e-transactions [21]. In the context of mobile agents, EOS implies that at each stage in the execution of an application (executed by a set of agent replicas), the effects of *exactly one of the replicas* is committed. This property does not, however, tackle other fault tolerant requirements: for instance, in applications where the results of more than one migrating agent can be combined to give rise to a new stage. The properties of virtual synchrony renders the mobile groups approach the suitable functionality for implementing EOS and other reliability requirements not addressed by the existing mobile agent fault tolerant systems.

A previous version of this paper introduced the main concepts of mobile groups and related membership protocol [29]. Here, we extend our work by further discussing and refining the concepts regarding the proposed membership protocol and presenting performance data from a number of experiments—where we measured the time to reach agreement in a new view caused by move operations and process crashes. The remainder of this paper is organized as follows. In Section 2 we discuss related work. In Section 3 we present the main part of the paper: the mobile groups properties, a related membership protocol and its correctness proofs, performance figures from an implementation, and some discussion on the complexity of the protocol. Finally, in Section 4 we draw our conclusions.

2. Related work

Previous work incorporated movement into group communication services for environments with mobile devices, for example, protocols for total or causal ordering and a membership service (e.g. [9,16,35,36]). In these environments processes start and terminate their executions on the same host. However, since the host may be mobile, a process may change its location in the physical environment when the related host moves. In mobile groups we are considering that hosts are not mobile, but agents can move from a host to another. These problems are similar in the fact that a group member can change its location in the distributed environment, but they differ in other aspects, such as the way messages are routed (done at the application level, in the case of mobile agents, and mainly at the network level, in the case of mobile devices) and functionality provided. Furthermore, unlike the mobile groups, none of these works take into account the movement action as an event to be ordered against other events in the system (such as crashes and message delivery). Thus, the directed adaptation of such algorithms to groups of mobile agents is restricted. The algorithm in [9], for instance, is based on a static server with which the group members must communicate no matter where they are. This constrains the use of such protocols for the mobile agent scenario in some conditions, for example, if the agents forming the same mobile group move altogether to a local network that latter becomes disconnected from the static server. With mobile groups, no supplementary entity is required, not even for keeping or updating the addresses of moving agents. In [16], the authors considered the problem of having mobile clients of a pool of servers. The servers are distributed in nodes in the fixed network (i.e., they are stationary). The main outcome of their solution is to show how a group could be used to give unbreakable point-to-point TCP connections from a base system to a single mo-

bile device. Their solution is similar to ours in the sense that both use virtual synchrony within groups, but they differ in the functionality provided and target environment (mobile devices and migrating agents, respectively).

In the context of mobile agent systems, different forms of interaction between agents were proposed. Existing mobile agent systems (e.g., Voyager [32]) provides many forms of communication between agents (remote method invocation, events, unreliable multicasts, etc.). In [34] the authors extend Linda to integrate mobility. Communication through tuple spaces and with guarantees of group systems are two different approaches for supporting mobile agent coordination that fulfill different sets of requirements. In [8] a concept for coordinating groups of agents is described, but which is not fault tolerant. Approaches for reliable message delivery to mobile agents were also proposed [31,37]. In [31] the approach supports uni- and multicast, but faults are not considered in the model. In [37] failures are considered, but the approach supports only unicast (point-to-point communication). In [30], the authors contribute to the understanding of synchronization requirements of distinct mobile agent Internet applications, by analyzing eight models for synchronization and communication between distributed agents. However, fault tolerance has not been addressed in this analysis. A central issue of the abovementioned models is the concept of *location synchronization*, which they propose to allow one or more agents to coordinate the location of their executions. For instance, in the *location-dependent synchronous communication model*, they propose the use of a specific host, deterministically chosen, where agents requests are first matched for afterwards being informed of each other's location addresses. Thanks to the location-aware membership modus operandi, the mobile groups can serve as a basic mechanism to implement such a model. In such a scenario, no third entity is required, as migrating agents will have their current location addresses mutually updated by the group membership protocol. Moreover, mobile groups do it in a fault tolerant manner. Therefore, in some sense, these works can be regarded as complementary efforts towards the accomplishment of reliable coordination and synchronization required by mobile agent systems.

To the best of our knowledge no previous work exists that describes a concept for supporting guarantees as those provided by traditional group communication systems in the context of mobile agent-based systems. In particular, as far as we know, there is no formalism for group membership that considers the movement action as an event to be synchronized (or ordered) against other events such as crashes and message delivery. Note that it is not only the fact that a view reveals the current locations of mobile agents, but also that movements are automatically synchronized with other actions (including other movements). Synchronizing the movements with the computations performed by distributed agents is, however, an important requirement of agent systems [30].

3. Mobile groups

3.1. System model and assumptions

We assume a distributed system as a collection of agents, locations and communication channels. A location represents a logical place in the distributed environment where agents execute. When a mobile agent migrates, it moves from a location to another. Agents communicate by exchanging messages through reliable communications channels, i.e., transmitted messages are received uncorrupted and in the sequential sent order, as long as the message sender does not crash until the message is received (reliable channels can be implemented over unreliable channels by tagging transmitted messages with sequential numbers, delivering such messages according to the sequential order and asking for retransmission in case of missing messages). As implied by our reliable channel assumption, we assume that network partitions do not occur or, when they occur, they are repaired within a finite amount of time and communication reestablished.

No bounds on message transmission or relative agent execution times are assumed (asynchronous environment). Agents and locations are assumed to fail only by crashing (without producing any further action), and the agents of a faulty location are assumed to have crashed. The failure of a given location is not directly handled. Instead, it is only detected when the associated agents are detected faulty. An agent that never crashes is named correct.

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

- $join(g)$: issued by an agent, when it wants to join group g;
- $leave(g)$: issued by an agent, when it wants to leave group g;
- $move(g, l)$: issued when an agent wants to move from its current location to location l;
- $send(g, m)$: issued by an agent when it wants to multicast a message m to the members of group g;
- $receive(g, m)$: issued by an agent to receive a message m multicast from the group g.

An agent p_i *of a group* g also *installs views*, *named* $v_i(g)$. In mobile groups a view $v_i(g)$, $v_i(g) \subset \{(p, l) \mid p \in g \text{ and } l \in L\}$, is a mapping between agents of group g and locations. A view represents the set of group members that are mutually considered operational in a given instant of the group existence and indicates the locations where these members are (a pair (p, l) in a view indicates that agent p is currently at location l). This set can change dynamically on the occurrence of agent crashes (suspicions) or when agents deliberately leave, join, or move to another location. Every time a change occurs in the group view, a new view is installed by a group membership protocol. Each

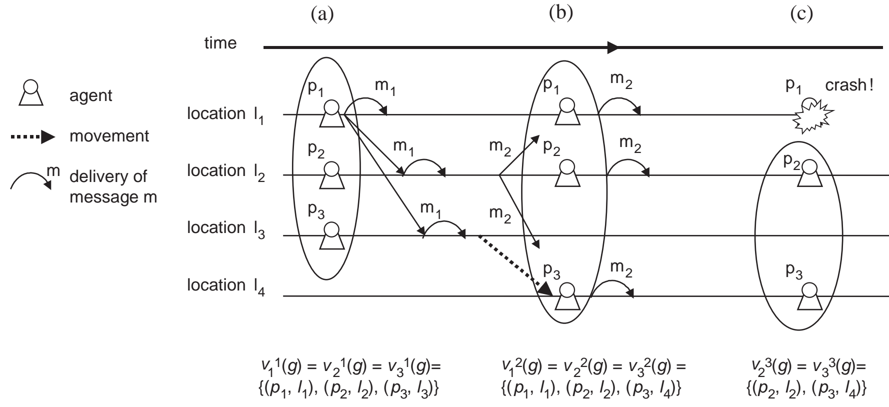

Fig. 1. An example of the mobile groups.

view installed by an agent is associated with a number that increases monotonically with group view installations. In a group $g = \{p_1, p_2, \ldots, p_n\}$, $v_i^k(g)$ denotes the view number k installed by agent p_i. Where suitable (i.e., there is no ambiguity), we will omit the agent identity of a view (e.g., $v^k(g)$), the group identity (*e.g.*, v_i^k), or even simply refer to view k of g meaning $v^k(g)$.

Fig. 1 illustrates a group initially with three agent members, p_1, p_2 and p_3. These agents install, respectively, views $v_1^1(g), v_2^1(g)$ and $v_3^1(g)$ (Fig. 1a). These views are identical and indicate that agents p_1, p_2, and p_3 consider each other operational and know where they are in the distributed environment ($v_1^1(g) = v_2^1(g) = v_3^1(g) = \{(p_1, l_1), (p_2, l_2), (p_3, l_3)\}$). Later, agent p_3 moves to location l_4 (the movement is represented by the dashed line in Fig. 1b). A new view is installed by each process, reflecting that agent p_3 is now at a new location, l_4 (views $v_1^2(g) = v_2^2(g) = v_3^2(g) = \{(p_1, l_1), (p_2, l_2), (p_3, l_4)\}$). Afterwards, location l_1 crashes (Fig. 1c). A new view will be, then, installed by agents p_2 and p_3 reflecting that agent p_1 was removed from the group (views $v_2^3(g) = v_3^3(g) = \{(p_2, l_2), (p_3, l_4)\}$).

3.2. Mobile group properties

We consider that when a mobile group g is created, every group member p_i installs the same initial view $v_i^1 = \{(p_1, l_1), (p_2, l_2), \ldots, (p_n, ln)\}$. After the initial view is installed, any modification on the configuration of the mobile group (due to migrations, addition or elimination of members) will result in new views being installed, forming the sequence $v_i^1, v_i^2, \ldots, v_i^k$ where k represents a given moment on the view evolvement history. Agent p_i multicast messages only to the agents of its current view.

In this paper, we will only consider the so-called primary partition membership [10,17] where a unique sequence of views is installed for a given mobile agent 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 [3,17,19], will be explored in future works.

Let $v(g)$ be a view of group g. We will denote by $v(g).P$ the set of agents that occur in $v(g)$, i.e., $v(g).P = \{p \mid (p, l) \in v(g)\}$. In order to simplify the notation, we will say that an agent $p \in v(g)$ if and only if $p \in v(g).P$. Similarly, we will denote by $v(g).L$ the set of locations that occur in $v(g)$, i.e., $v(g).L = \{l \mid (p, l) \in v(g)\}$.

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

3.2.1. View safety properties

*Validity*01: if an agent $p_i \in g$ installs a view $v_i^k(g)$, then $p_i \in v_i^k(g)$.

*Validity*02: if an agent $p_j \in v_i^k(g).P - v_i^{k-1}(g).P$, $k > 1$, then p_j asked to join g.

*Validity*03: if an agent $p_j \in v_i^{k-1}(g).P - v_i^k(g).P$, $k > 1$, then p_j asked to leave g or it has been suspected of crashing by some group member.

*Validity*04: if the pair $(p_j, l') \in v_i^k(g)$ and $(p_j, l) \in v_i^{k-1}(g)$ and $l \neq l' (k > 1)$, then p_j asked to move from l to l'.

Validity01—also known as *self-inclusion* property—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.

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

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.

Observe in Fig. 1 that views with the same number are identical.

3.2.2. View liveness properties

*Termination*01: If an agent $p \in v_i^k(g)$ asks to leave g or crashes and there exist at least two correct agents in $v_i^k(g)$, then there will be a view $v_j^r(g)$ installed by an agent p_j, $r > k$, such that $p \notin v_j^r(g)$ and $p \in v_j^s(g)$ for all s such that $k \leqslant s < r$.

*Termination*02: If an agent p tries to join g, then there will be a view $v_j^r(g)$ installed by an agent p_j, such that $p \in v_j^r(g)$ and $p \notin v_j^s(g)$ for all s such $s < r$.

*Termination*03: If an agent p_i of a pair $(p_i, l) \in v_i^k(g)$ asks to move to location l' (i.e., executes the $move(g, l')$ operation), and it is not excluded from g, then there will be a view $v_j^r(g)$, $r > k$, installed by an agent p_j such that $(p_i, l') \in v_j^r(g)$ and $(p_i, l) \in v_j^s(g)$ for all s such that $k \leqslant s < r$.

The properties Termination01, Termination02, and Termination03, guarantee that a protocol implementing the mobile groups should eventually install a new view whenever an agent crashes (or leaves a group), joins a group, or moves to a new location, respectively.

3.2.3. Message delivery properties

If g is a mobile group, let $deliver(m, p_i, k)$, $p_i \in g$ denote the delivery of a message m to agent p_i in view $v_i^k(g)$. We define the following safety and liveness message delivery properties, $MD1$ and $MD2$, respectively:

$MD1$ (*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 $deliver(m, p_i, k) \Leftrightarrow deliver(m, p_j, k)$. That is, any two group members of g that install two consecutive views deliver the same set of messages between them. See in Fig. 1 that messages m_1 and m_2 are delivered by all agents at the same view (m_1 in view $v_i^1(g)$ and m_2 in view $v_i^2(g)$).

$MD2$ (*Liveness*): if an agent p_i sends a message m in view $v_i^r(g)$, then provided it continues to function as a member of g, it will eventually deliver m in some view $v_i^{r'}(g)$, $r' \geqslant r$. A message sent in a view might be delivered in a future view. This is illustrated in Fig. 1, where message m_2 was sent in view $v_i^1(g)$, but delivered in view $v_i^2(g)$.

MD1 and MD2 together enforce virtual synchrony between agents. The actual semantics of the virtual synchrony enforced by MD1 and MD2 is similar to the one initially proposed by Birman [11] and formalized in [5]. Birman's virtual synchrony requires that $r = r'$ in $MD2$ and it can be implemented by blocking sending messages during the view installation procedure. Our definition is more related to the ones of Transis [3] and Newtop [19] systems, which differ from ours by the fact they were primarily intended to partitionable memberships.

In order to save space, we have omitted the formal definition of some message delivery and view properties such as local monotonicity, initial view event, delivery integrity, and no duplication [17], which can be trivially deduced from our system model assumptions.

3.3. A membership protocol for the mobile groups

Solving the membership problem in an asynchronous system is not an easy task. In fact, it has been proved that the primary membership problem has no deterministic solution in such systems and that partitionable membership approaches can yield to group degeneration (multiple singleton member groups) [5,14]. This comes from the very fact that there is no way to accurately guarantee that correct members will not be removed from the membership, which may lead the group to virtual partitioning. On the other hand, even if one admits that a correct agent may be erroneously excluded from the membership, virtual synchrony still requires that group members see the same sequence of view changes (i.e., they must agree or come to a consensus on each view installed). However, it is well known that one cannot solve consensus in such systems when failures may occur [20]. We have tackled these problems in two directions. First, we have augmented our asynchronous system model with unreliable failure detectors in order to solve consensus [13,15]. Thus, we assume the existence of a distributed $\diamond$S (eventually strong) failure detector, as defined in [15]. The assumption $\diamond$S is necessary because this is the weakest condition, as proved in [13], to achieve the consensus in such an asynchronous environment. Furthermore, we also assume that a majority of agents of a group view does not crash as required by the $\diamond$S-based consensus protocol.

Second, we do not rely on unreliable information from the failure detector in order to remove an agent from the membership. Instead, view changes will only occur when either an explicit move, leave, or join is required or when sent messages are unstable during a long period (i.e., not received by at least one of the group members). If either condition occurs, all the agents (suspected of failure or not) are required to run consensus to agree on the new view. Therefore, the failure detectors are only used at this particular moment—during consensus—and, consequently, any mistakes made by a failure detector module has no impact on the composition of the group when the system is stable (no move and no late stabilization of messages). More importantly, even if agents are erroneously removed from a given view (they do not belong to the consensus agreed view), the $\diamond$S-based consensus ensures that at least a majority of the view members will remain members of the next view.

Our membership protocol uses repeated (possibly concurrent, but completely independent) executions of consensus where the kth execution of consensus is used to decide on the kth tentative view change to be atomically installed by operational group members. All the messages related to the kth tentative view change (and therefore, to the kth consensus execution) are tagged with the number k. Therefore, as in

[15], we denote the *propose* and *decide* primitives for the kth execution of consensus as $propose(k, -)$ and $decide(k, -)$. We also assume the primitive FD[p] to inquiry the failure detector ◇S. Thus, FD[p] = true means that agent p was suspected by the local failure detector module.

An agent sends a message m to a group g by using a best-effort multicast primitive, denoted $mcast(m, g)$, which causes the delivery of m to the current membership of g (denoted $m.g$) as long as the m sender does not crash before finishing the multicast.

3.3.1. Handling agent crashes

When m is received by a destination agent p, $p \in m.g$, m is immediately delivered to p and stored in a local buffer until m is known to be stable (i.e., received by all agents in $m.g$). If a message remains unstable for too long, the membership service will start a new view installation procedure for removing possibly crashed agents from the current view and delivering the unstable messages not delivered yet. In order to guarantee that all group members engage in the same view installation procedure, a reliable multicast primitive, denoted $rmcast(ChangeViewRequest, k)$, is employed to launch the change view procedure. The agreement property of the reliable multicast primitive guarantees that if any operational group member delivers the message ($ChangeViewRequest$, k), all the other operational members will do so. This request is then processed by a changing view task (described subsequently) which makes use of a ◇S-based consensus protocol to install identical views at all group members (thanks to the agreement property of the consensus).

During view installation, agents are required to send their sets of unstable messages and the new view will be formed removing those agents that were suspected by the local failure detectors, provided that a majority of the agents (whether suspected or not) has sent the unstable sets. Thus, the new view installation will only progress if any group member fails in sending the corresponding unstable set. Such agents—those that did not send the unstable set—are considered as crashed and a new view will be installed which does not include them. We should bear in mind, however, that those suspected agents may be just too slow (due to an overloaded site, for example). Although a false suspicion may cause the removal of an operational agent from a group, the membership service assures that the membership information will always be kept mutually consistent among the not suspected agents (timeout values should be carefully chosen in order to make false suspicions rare).

The membership protocol working on behalf of a group member p_i consists of the *ChangingView* and *Message Stability Assessment* tasks, and the operations move, leave, and join. The tasks and operations can all execute concurrently. However, each of them runs atomically (i.e., there is only one execution of a given task in a given agent). The referred tasks and the Move operation are described below. Due to space limitations we omit here the description of the Leave and Join operations, which can be found in [28].

3.3.2. The message stability assessment task

Let us assume that all agents maintain the set *unstable* to record the unstable messages. This set, shared by all the tasks, is initialized empty and updated in the Message Stability Assessment task. The set variable *currentview*, also shared by all tasks, represents the latest view installed by an agent. For a given 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 agent p_i (*currentview* is initialized with the initial view v^1—see Section 3.2 for the notation definitions).

The message stability assessment task (shown on Fig. 2, upper part) is run for each sent or received message m. In this task, initially a timeout is set (statement 1) and the message m is put in the *unstable* set (statement 2). After that, it is waited until acknowledgements for m are received from all agents belonging to the current view or until the timeout expires (statement 3). In the first case, the timeout is cancelled for m, which is then removed from the *unstable* set. If the timeout expires, the global variable *tvc* (tentative view counter) is incremented and reliably multicast to the group together with a view change request message (statement 4).

3.3.3. The ChangingView task

The ChangingView task, shown on Fig. 3, is initiated on the receiving of a ($ChangeViewRequest$, k) message and it is performed to completion only if it has not already been executed for the received k (see the exit command in statement 2). In order to prevent the move operation and Message Stability Assessment task from initiating ChangingView tasks for values of k already used (thus, resulting in the exit command), the value of *tvc* is updated to keep the maximum value between its current value and the received k of the last execution of the ChangingView.

After updating the value of *tvc*, all group members are required to send their unstable sets (statement 3). This procedure is used to define a possibly new view in the following way: an agent is removed from the next proposed view if its unstable set has not been received (statements 4–7). Notice that a majority of unstable sets is certain to be received since, by assumption, the majority of agents is correct—see Section 3.1. The proposed new view is recorded in the variable *newview*, where $newview.L[i]$ identifies the location associated with an agent p_i in the corresponding view (statements 7 and 8).

The parameter *Moves*, passed with the *unstable* set in statement 3 and received as *Mset* in statement 4, is related to the *move* operation and carries the identities and corresponding new locations of moving agents (see *handling moves* later in this section). In order to update (possibly) new locations of moving processes in the proposed view, all new locations received in the parameter *Mset* are collected into the

```
Message Stability Assessment Task
(launched by an agent p for every sent or received message m)
(1) set timeout(m);
(2) unstable ← unstable ∪{m}; /* puts m into set unstable*/
(3) Wait until ∀ q ∈ currentview, received(m,ack)  or  expires timeout(m);
(4) if ∀ q ∈  currentview,  received(m,ack) then
        /* the receipt of m was acknowledged by all members */
        { cancel timeout(m);
          unstable ←  unstable – {m}; /* removes m from the set unstable */}
    else  /* if timeout expired */
        {  tvc ← tvc + 1; /* increments the tentative view counter */
           rmcast (changeViewRequest, tvc); /* try to install a new view */ }


Move(g , l);  /* moves process pi to location l */
(1) "Ask the remote location about the installation of a new agent"
(2) if movement is authorized then
        "start the membership service on the remote place
         with blocked local delivery"
    else {exit; return error code}
(3) Moves := Moves + (pi,l);
(4) tvc ←  tvc + 1;
(5)  rmcast(changeViewRequest,tvc);
     /* try to install a new view with the new location*/
(6)  await ChangingView task to finish;
(7) if  pi  ∈ currentview.P
    then { migrate pi to remote location l;
             install current view at remote location;
             forward to l all the received  but not delivered messages;
             unblock delivery at remote process
             terminate the local pi}
    else exit; /* pi was suspected and removed from g*/
```

Fig. 2. Message stability assessment task and move task.

Moves variable (statement 6). The proposed view, *newview*, is then updated to include the new locations (statement 8).

All the unstable messages present in the unstable sets are also collected into the variable *allunstable* (statement 5).

After defining a new proposed view (*newview*) and the set of all unstable messages (*allunstable*), the group members execute the ◇S consensus to agree on the new view, represented by the pair (allunstable, *newview*), returned by the *decide* primitive (statements 9 and 10). After reaching consensus, each group agent first delivers the not yet delivered unstable messages from the consensus outcome (statement 11), and then installs the new agreed view (statements 13 and 14). Notice that, as the system is asynchronous, there may exist several concurrent consensus executions, identified by distinct numbers (for instance, $k-2, k-1, k$). 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 elsewhere [15,24].

If the new agreed view brings no modifications on the current group view or if the local agent does not appear in the decided view (because it was removed by a remote agent), no new view is installed by it (statements 13 and 14). Otherwise, a new view is installed by updating the variables *currentview* and *viewnumber* (*viewnumber* denotes the number of a view installed by an agent p_i—it is set to 1 when the initial view $v_i^1(g)$ is formed). Note that *currentview* and *viewnumber* together indicate the agent view in a given moment.

Blocking message delivery is necessary to guarantee properties MD1 and MD2 (see section 3.2.3).

3.3.4. Handling moves

A group agent p_i willing to move to a new location l must issue the operation $move(pg, l)$, shown on Fig. 2 (lower

ChangingView Task * executed by agent P_i^*

```
(1) Upon receiving (changeViewRequest,k)  message do
(2) tvc := maximum(tvc,k);
    if (mcast(unstable,k,Moves) has already been multicast) then exit
(3) block normal local delivery; mcast(unstable,k,Moves);
/* collect the unstable messages from all unsuspected  members, provided that a  majority
of them has sent their unstable messages */
(4) Wait until
   [( ∀q ∈ currentview : received(unstable,k,Mset) from q or FD(q) = true)
    and  for ⌈(n+1)/2⌉ processes q: received (unstable,k,Mset) from q]
(5) for each received (unstable,k,Mset) do allunstable := allunstable ∪ unstable;
(6) for each received (unstable,k,Mset) do Moves := Moves ∪ Mset;
(7) newview := currentview  – {(q,l) | (q,l) ∈ currentview and (unstable,k,Mset) was not
     received from q }  /*remove suspected processes*/
/*update newview with the new locations*/
(8) ∀ (p,l) ∈ Moves, newview.L[p] := l;
/* run consensus k */
(9) Propose(k, allunstable, newview);
(10) Wait until decide(k, allunstable, newview);
/* deliver the agreed unstable messages not  delivered yet */
(11) Deliver all m ∈ allunstable | m has not been delivered yet
/* update the Moves set*/
(12) ∀(p,l), ((p,l) ∈ Moves and (p,l) ∈ newview),  Moves := Moves – {(p,l)};
/* install the new view for consensus k only if the current view was modified */
(13) if newview = currentview then exit;   /* it was a false  suspicion and no
                                               moves have been decided*/
/* install the new view if pi still belongs to the view decided */
(14) if pi ∈ newview.P
       then { viewnumber := viewnumber + 1; currentview := newview; }
       else "inform process pi that it has been removed  from the mobile group";
(15) Unblock normal delivery
```

Fig. 3. ChangingView task.

part). As a result, the mobile group service will first make sure that there are conditions to the required migration (statements 1 and 2). If so, the new location information of p_i is updated into the shared set variable, called Moves, that maintains the pairs (p, l) of all agents willing to move (statement 3) and the *ChangingView* task is started—by multicasting a *ChangeViewRequest* message—to try to install the new view with the new location for p_i (statements 4 and 5). In the *ChangingView* task, the local *Moves* set is transmitted to remote agents together with the *unstable* set in statement 3 and updated from the remote *Moves* sets in statement 6 (from the receiving parameter *Mset*). That is, group members will exchange their *Moves* sets and the union of all exchanged Moves will be taken as the new *Moves* set (statement 6). Afterwards, the updated *Moves* is used to inform the new locations of the tentative view (*newview* variable) to be subjected to the consensus module (statements 9 and 10). This consensus procedure is mandatory since the mobile groups specification requires all the events (crashes, joins, leaves, moves, and message delivery) to be ordered with respect to the new agent location installation. The moving agent terminates after installing the current view at the remote location and forwarding to it the received but not delivered messages (statement 7 of move operation).

What makes a move operation to eventually conclude is the fact that the moving process identifier is placed into *Moves* before incrementing *tvc* and initiating the *ChangeView* task, and it is only removed after appearing in the decided *newview* (statement 12 of *ChangingView* task). Incrementing *tvc* ensures that there will be at least one execution of *ChangingView* task, as the incremented *tvc* is always larger than the number k of the last *ChangingView* task execution (notice that *tcv* is updated to the maximum between its current value and the received k—see statement 2 of the *ChangingView* task). If two or more parallel movements lead to multiple incrementations of *tvc*, there will be multiple executions of the *changing view* task. However, all parallel moves can be resolved in just one execution, making the other executions without effect (as far as movements are concerned). We could also optimize the algorithm

to avoid unnecessary executions of *ChangingView* task, but we chose not to incorporate such optimizations to facilitate the understanding of the protocols.

Finally, notice that it is possible for an agent to execute the *ChangingView* task and not succeed in migrating (statement 7 of move task), if such an agent was considered as crashed by some remote agent ($p_i \notin currentview.P$).

3.4. Implementation issues

The protocol just described has been implemented in a JAVA/Unix environment. The implementation has a layered architecture. The lowest layer is the Transport layer, responsible for sending/receiving messages to/from the network. Above the transport layer is a layer with basic building blocks: *mcast* and *rmcast* primitives, a failure detector module, and a timer. Above this layer is the Consensus implementation. Above the Consensus layer is the ChangingView Task. In the uppermost layer are the components that implement the mobile group operations (join, leave and move operations) and the message stability assessment task. The local communication between components of the architecture is performed by using an event channel and direct method invocations. The transport layer is based on UDP. We did not use a specific agent system (such as Voyager). We developed an own implementation of the agent migration support, based on Java RMI. This support consists basically of a set of Java classes for enabling agent transportation, execution, and dynamic class loading.

As our main interest is to use mobile groups for implementing mobile agent fault-tolerance, where an application is typically executed by a few agent replicas, we have run experiments for groups from 3 to 6 agents spread over a local area network, where each agent was sited in a distinct location (only one location per computer). The experimental environment consisted of six Linux Pentium III computers (800 MHz, 191 MB RAM) connected though a 100 megabits network. Three series of experiments were performed to measure the time necessary to reach agreement on a new view caused by move operations or by an agent crash.

In the first series, a group of agents reaches agreement on a new view caused by a move operation. In this particular experiment, we did not measure the migration time (the time to send a serialized agent from one location to another). In each experiment, one of the agents proposes a new view to reflect its movement to a new agency. Table 1 presents, for sets of three to six agents, the mean, median, and standard deviation of the time measured from the instant when the agent proposes a new view the instant when it installs this view. This experiment was repeated 1000 times for each set of agents (total of 4000 runs).

In the second series, the migration time is included. In each experiment, a specific agent moved from one location (original location) to another (destination location). Table 2 presents, for sets of three to six agents, the mean, median, and standard deviation of the time measured from the instant when the agent proposes a new view until the original location receives the confirmation that the agent has been successfully installed at the destination location. This experiment was repeated 200 times for each set of agents (total of 800 runs).

In the third series, it was measured the time necessary to reach agreement on a new view when an agent fails. Table 3 presents the mean, median and standard deviation of the time measured from the instant when one of the agents suspects that another agent of the group has failed until it installs the next view (which excludes the faulty agent). The values for three agents in Table 3 correspond to the scenario when one of these agents detects the failure of a fourth agent and the three agents agree on the new view. The values for four and five agents are defined similarly (i.e., they involve the detection of the failure of an additional agent). This experiment was repeated 1000 times for each set of agents (total of 3000 runs).

In the third series, the high figures are due to the time needed for detecting the faulty agent. In our prototype, failure detection is based on *heartbeat* messages that are transmitted every 500 ms (in fact, only when no application message is transmitted in this period). The *heartbeat* messages are handled by the *MessageStabilityAssessement* Task (Fig. 2—upper part) like application messages. When a message arrives (be it a heartbeat or not), the *MessageStabilityAssessement* task will start a timeout of 500 ms for this message to be stable. Therefore, if an agent fails just after sending a *hearbeat*, it will take, in the worst-case scenario, approximately 1000 ms to launch a *ChangingView* task to remove such a failed agent—500 ms for sending the next *heartbeat*, plus 500 ms for expiring the related timeout. In the best case scenario, if the agent fails just before sending the *heartbeat*, the detection time will be close to 500 ms (the timeout for the message to become stable). This explains why these figures varied between 500 and 1000 ms and the high standard deviation measured.

Observe also that the mean and median times for 4 agents in Table 3 are slightly higher than the values for 5 agents. This happened because the average detection time observed for the experiment with 5 agents was slightly smaller. Notice that for such a small number of agents, the detection time dominates the total time to establish a new view (while the detection time ranges from 500 to 1000 ms, the mean time value for reaching agreement—without movement—ranged, for example, from 24 to 42 ms for 3–5 agents in Table 1).

Finally, notice that an agreement on a new view takes place inside the *ChangingView* task (see Fig. 3) where a new view is proposed taking into account all pending moves (those initiated but not realized yet) (see lines 6–8 of the *ChangingView* task). Therefore, the time spent in a single agreement procedure due to a single move operation is the same to handle multiple, parallel moves. Moreover, any agent crash that occurs during the handling of a movement will also be resolved in the same agreement procedure, yielding a new view that reflects all movements and crashes

ARTICLE IN PRESS

Table 1
Time to reach agreement on a new view due to agent migration (without including agent migration time)

First series:					
Mean (ms)	Median (ms)	Standard deviation (ms)	Mean (ms)	Median (ms)	Standard deviation (ms)
3 agents			4 agents		
24.5	24.0	2.9	36.6	37.0	3.3
5 agents			6 agents		
42.1	42.0	3.6	49.8	50.0	3.9

Table 2
Time to reach agreement on a new view due to agent migration (including agent migration time)

Second series:					
Mean (ms)	Median (ms)	Standard deviation (ms)	Mean (ms)	Median (ms)	Standard deviation (ms)
3 agents			4 agents		
44.9	43.0	7.8	57.6	55.0	10.2
5 agents			6 agents		
63.3	60.0	11.5	67.7	64.0	11.4

Table 3
Time to reach agreement on a new view, after suspecting an agent failure

Third series:					
Mean (ms)	Median (ms)	Standard deviation (ms)	Mean (ms)	Median (ms)	Standard deviation (ms)
3 agents			4 agents		
771.6	768.8	142.1	794.6	799.8	144.3
5 agents					
785.4	778.8	150.3			

perceived. Consequently, the time taken to solve a move is a good indication of the protocol performance in the general case when crashes and moves may occur at the same time.

3.5. Discussion

The semantics of virtual synchrony, which is required by some applications, puts a limitation on the scalability of the membership protocol. Virtual synchrony requires total order of views and total order is equivalent to consensus under the failure detectors assumption [15]. Hence, the performance of consensus can be considered as a bottleneck on the improvements one can achieve. The Chandra–Toueg algorithm used to solve consensus has multiple rounds with a distinct agent acting as the coordinator in every round (the so-called rotating coordinator paradigm) [15]. Each round involves four communication steps: the first for the participants to propose consensus values, the second for the coordinator to propose a decision value, the third for the participants to acknowledge the decision, and finally, the fourth for the coordinator to send the decided value (it has been shown that this algorithm can be optimized to 3 or even 2 communications steps when used with transport multicast facilities [41]). If failures occur (or false suspicions), new rounds can be executed (with different coordinators). A good point of this approach is that the algorithm can tolerate an unbounded number of incorrect failure suspicions, which is indeed required by the asynchronous environment. Analysing the protocol complexity in a deterministic way (number of rounds or communication steps) cannot be realized for the general case where failure patterns cannot be predicted. Nevertheless, if we consider runs without failures, the consensus module will solve consensus in the very first round. Another interesting outcome of the algorithm is that if too many failures occur so that the $\diamond S$ assumptions cannot be achieved, the system is fail-safe. That is, the decision may be indefinitely postponed but no incorrect decision will be ever made (agreement on the views is never violated).

By adopting a modular approach for the membership protocol, we allow for the exploitation of alternative solutions for the underlying consensus. An advantage of this approach is that none of the concepts, definitions, and basic algorithms need to be changed to validate a new optimized membership protocol.

Improvements on the performance and complexity of the protocol can be made if we strengthen some of the assumptions. For instance, if we assume that the system has a perfect (or eventual perfect) failure detector [15], we can achieve consensus more efficiently—as a decision on a new view can immediately be taken on a failure notification. In that case, extra care must be taken to guarantee that the violation of such a stronger assumption will not lead to the violation of the agreement property. Other alternatives to improve the scalability of the mobile group membership pro-

tocol would lead to the weakening of our membership specification, such as in light-weight membership [4,18,23,39], the exploitation of specific communication topology or restricting the running of the membership protocol to a set of dedicated servers that work on behalf of the whole group [2,4,18,23,26]. A complete discussion of these approaches is, however, beyond the scope of this paper. Nonetheless, we could also implement the mobile groups with some of the above-mentioned assumptions and techniques without modifying the main properties and concepts presented in this paper.

One could argue that the movement action and its synchronization with other events—in fact, the mobile groups themselves—could be implemented by using conventional group systems combined with a mobility support. For instance, by making the moving agent leave the group before the movement and join it again after the movement, or by moving the agent—through a mobility layer—and then using a total order multicast to synchronize the movement action with the other group events. However, these solutions have two major drawbacks. First, the application would have to pay an extra care to handle and synchronize the events observed during the movement (such as message multicast and agent crashes)—since agent mobility would be hidden from the group action history. Second, because the mobile groups approach provides both reliable coordination and mobility support, it is less costly than the alternatives discussed above. To begin with, the cost incurred by the location aware view installation procedure of mobile groups (which is dominated by the consensus procedure) is equivalent to the two view installations (caused by the leave and join) or the total order required to synchronize the movement actions—as previously observed, Chandra and Toueg have proved that consensus and total order multicast are equivalent problems under the failure detection assumption [15]. Furthermore, for the later approach (using total order), the view installation procedure provided by the conventional group system is still required to synchronize other events (such as agent crashes). Therefore, these approaches would have to pay an extra cost to maintain the reliability level for the agent migration present in conventional mobile systems (for instance, by carefully updating the new location addresses of the moving agents). On the other hand, reliable agent migration and corresponding updating of moving agent locations is a direct benefit achieved with mobile groups as new location addresses are reliably installed by functioning agents.

3.5.1. Applying mobile groups to other application scenarios

Because mobile groups provide all the facilities available in conventional group systems (such as virtual synchrony), they can be applied for any standard group communication application that requires code migration (such as process migration for load balancing), an issue not properly addressed so far in conventional group systems. Furthermore, the mobile groups concept could also be applied to other distributed systems domains such as mobile computing and ad hoc networks, where the synchronization of device movements against other events may be an important requirement. As an example, consider mobile agents running on mobile devices that do not need continuous network connectivity, as connections should last just the necessary time to inject agents from terminals into the fixed network. With this approach, end users may access services and obtain the related results upon reconnection. In this scenario, due to the mutually consistent view of group events, including agent movements, mobile groups can be employed to maintain the services required by users while migrating (i.e., their working environment and subscribed services). Furthermore, mobile groups could be used to co-ordinate the activities of a set of mobile users (which collaborate within a given computation) by consistently disseminating device (or terminals) connection and disconnection information.

3.6. Correctness of the protocol

Due to space limitations we do not present in this paper the operations and correctness proofs connected with agent joins and leaves and omit the proofs for most properties (they can be found in [28]). The letters C, M, and S will be used to indicate statement lines of the *ChangingView* task, move operation, and task, respectively (e.g., C10 stands for statement line number 10 of *ChangingView* task).

Unique sequence of views: Consider a group g and let v_i^k and v_j^k be the view of number k of g installed by p_i and p_j, respectively (view number $k = k$th view installed only for the agents 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 $v_i^k = v_j^k$.

Message Stability Assessment: Let us first show that all correct agents execute *ChangingView* task for the same set of k values (Lemma 1) and they execute the same sequence of consensus (Lemma 2). We subsequently use these lemmas to prove the *unique sequence of views* property.

Lemma 1. *All correct agents of g activate executions of the ChangingView task* (*i.e.*, *from lines* C3–C15) *for the same set of k values*.

Proof. The *ChangingView* task is activated in agent p_i every time it receives a (*changeViewRequest*, k) message, which is transmitted either from the *Message Stability Assessment* task (S4, *else* branch) or from the move operation (M5), when *rmcast* (*changeViewRequest*, *tvc*) is executed in any $p_j \in g$. The agreement property (atomicity property) of the reliable multicast used to send the (*changeViewRequest*, *tvc*) messages assures that any correct p_i ($p_i \in g$) will receive the same set of (*changeViewRequest*, k) messages, and, therefore, will trigger executions of the *ChangingView* task for the same set of values k. □

Lemma 2. *All correct agents execute consensus k (i.e., propose$(k, \ldots)$ and decide$(k, \ldots)$, in lines* C9 *and* C10, *respectively) for the same set of values k.*

Proof. By Lemma 1, all correct agents will receive the same set of (*changeViewRequest*, k) messages, which in turn will trigger the *ChangingView* tasks for the same set of *k values*. Since k is not *modified inside the ChangingView Task*, a consensus k (C9 and C10) will not be executed only if an agent executing *ChangingView* task blocks in the wait command of line C4. The condition of the wait command is expressed as a conjunction of two predicates. Given that messages are assumed not to be lost, the first predicate eventually becomes true. This is due to the fact that a (*unstable*, k, *Moves*) message will not arrive only if the sending agent fails. If it happens, it will eventually be suspected as faulty, due to the ◇S completeness property. The second predicate of C4 demands that messages are received from a majority of agents, which will eventually become true, as messages are not lost and we have also assumed a majority of correct agents. It follows that Lemma 2 is true. □

Theorem. *Agents respect the Unique Sequence of View property.*

Proof. A new view is installed by an agent when it updates its variables *viewnumber* and *currentview* inside the ChangingView task (C14). Unique sequence of view can be proved by induction on the numbers of view installed during a group g lifetime. Notice that by assumption all agents install the same initial view v^1 (Section 3.2). Now let us now suppose by induction hypothesis that all agents installed view $v^k,\ k > 1$, respecting the agreement property. That is, if an agent $p_i \in v_i^{k-1}(g)$ installs view $v_i^k(g)$ and another agent $p_j \in$ view $v_i^{k-1}(g)$ installs view $v_j^k(g)$, then $v_i^k(g) = v_j^k(g)$ for any p_i and $p_j \in v_i^{k-1}(g)$. Now, we must show that $v_i^{k+1}(g) = v_j^{k+1}(g)$.

As *viewnumber* is initialized with value 1 (when the initial view is set) and it is increased by 1 for every new view installed (C14), when $v_i^k(g)$ and $v_j^k(g)$ are established, the values of *viewnumber* for agents p_i and p_j will be updated to k. The new view $v^{k+1}(g)$ is then formed when the value of *viewnumber* is updated to $k + 1$ and *currentview* to the set *newview* in C14.

Let $x, x \geqslant k + 1$, be the number of the consensus and change view request (used as a tag for the change view messages) which resulted in the establishment of $v_j^{k+1}(g)$. The view $v^{k+1}(g)$ will be formed and installed only after the agents in view v^k have decided the outcome of the corresponding consensus x (C10). From Lemma 2 and as we have assumed no partitioning (Section 3.1), then p_i and p_j will be engaged in the same execution of consensus x. Because the agreement property of consensus guarantees for both, p_i and p_j, the same output for the *newview* set (C10), we only need to show that p_j also incremented its *viewnumber* after consensus x. Now, suppose by contradiction, that this was not the case. That is, p_i and p_j executed *decide*(x, *allunstable*, *newview*), and p_i incremented its *viewnumber* to $k + 1$, but p_j did not increment it. This could only happen in two cases: (i) p_j detected no change for the *newview* set decided in relation to *currentview* (view v^k) (command *exit* in C13) or (ii) p_j did not belong to the *newview* decided (branch *else* of C14). It follows that both cases could not actually have happened. (i) If p_i detected the change (as it incremented *viewnumber*), but p_j did not detect it, then p_i and p_j would have to disagree on the decided *newview* as *currentview* for v^k is the same for both in C13 (this contradicts the agreement of consensus) and in (ii) p_j would not be able to install a view with number larger than k as it is considered as crashed (*else* branch of C14). So, we found in both cases a contradiction.

4. Conclusions

In this paper, we presented the properties of mobile groups and described a membership protocol that satisfies them. Mobile groups support virtual synchrony in which messages are synchronized with not only crashes (suspicious), leaves and join operations, but also with movement operations. The mobile group approach represents a novel mobility support mechanism that fulfills part of the reliability requirements of mobile agent systems.

To the best of our knowledge this is the first work that provides and formally defines a concept of group that supports moving agents by integrating the movement events inside 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 agent (for example, total order delivery with respect to moving actions). That would not be achieved satisfactorily by using traditional group communication systems.

Mobile agent fault tolerance requires that (a few) agent replicas be placed on distinct hosts of the network. If this is not respected, this protocol (or any other) will loose resilience. For instance, if a given host maintains several agents of the same group, the protocol will tolerate a smaller number of host crashes. A way to tackle this problem is to put a unique instance of the consensus module in a given host and make this module to run consensus on behalf of all agents hosted. This can be a useful approach not only to maintain a desirable level of reliability, but also for scalability purposes (for some classes of applications where groups maintain a great number of agents) as consensus will produce less transmitted messages over the network.

Apart from the conventional agent based applications, mobile groups could also be a useful mechanism to cope with another dimension of mobility where mobile devices can change their access point in a network. This kind of mobil-

ity, usually referred to as mobile computing, has sometimes been addressed through a mobile agent mechanism [12]. The rationale behind these classes of solutions lies in the fact that the mobile agents do not need continuous network connectivity, as connections should last just the necessary time to inject agents from terminals into the fixed network. With this approach, end users may access services and obtain the related results upon reconnection. In this scenario, the agents executing on behalf of disconnected users (or terminals) and assigned services could form a mobile group. The mutually consistent dissemination of device (or terminals) connection and disconnection events can be used to dynamically adapt applications (working on the users side) to distinct environments.

Mobile groups complement other efforts in providing reliability of mobile agent based applications, such as concepts for mobile agent fault tolerance and transactional support. Mobile groups are, thus a further step towards providing an effective support for coordinating groups of agents. Some effort must still be expended in developing this concept, for example, for providing stronger message delivery guarantees and support for partitioning.

Acknowledgments

The authors would like to thank Ken Birman and Michel Hurfin for their constructive comments on earlier versions of this paper.

References

[1] Association of Computing Machinary, Comm. ACM (April 1996) 39(4), special issue on group communication systems, April 1996.
[2] D.A. Agarwal, L.E. Moser, P.M. Melliar-Smith, R.K. Budhia, The Totem multiple-ring ordering and topology maintenance protocol, ACM Trans. Comput. Systems 16 (2) (May 1998) 93–132.
[3] Y. Amir, D. Dolev, S. Kramer, D. Malki, Transis: a computing subsystem for high availability, in: Proceedings of the 22nd International Symposium on Fault-Tolerant Computing, Boston, July 1992, pp. 76–84.
[4] Y. Amir, J. Stanton, The Spread wide area group communication system, TR CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University, 1998.
[5] E. Anceaume, B. Charron-Bost, P. Minet, S. Toueg, On the formal specification of group membership services, Technical Report 95-1534, Cornell University, Ithaca, USA, 1995.
[6] F.M. Assis Silva, R. Popescu-Zeletin, An approach for providing mobile agent fault tolerance, Lecture Notes on Computer Science, vol. 1477, Springer, Berlin, September 1998, pp. 14–25.
[7] O. Babaoglu, M. Baker, R. Davali, L. Gianchini, Relacs: a communication infrastructure for constructing reliable applications in large-scale distributed systems, BROADCAST Project Deliverable Report, October 1994.
[8] J. Baumann, N. Radouniklis, in: H. König, K. Geihs, T. Preuá (Eds.), Agent Groups in Mobile Agent Systems, Distributed Applications and Interoperable Systems (DAIS'97), Chapman & Hall, New York, 1997.
[9] A. Bartoli, Group-based Multicast and Dynamic Membership in Wireless Networks with Incomplete Spatial Coverage, Mobile Network and Applications, vol. 3, Baltzer Science, 1998.
[10] K. Birman, The process group approach to reliable distributed computing, Commun. Assoc. Comput. Mach. 9 (12) (December 1993) 36–53.
[11] K. Birman, A. Schiper, P. Stephenson, Lightweight causal and atomic group multicast, ACM Trans. Comput. Systems 9 (3) (August 1991) 272–314.
[12] P. Bellavista, A. Corradi, C. Stefanelli, Mobile agent middleware for mobile computing, IEEE Comput. 34 (3) (March 2001) 73–81.
[13] T. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, J. Assoc. Comput. Mach. 43 (4) (July 1996) 685–722.
[14] T. Chandra, V. Hadzilacos, S. Toueg, B. Charron-Bost, On the impossibility of group membership, in: 15th ACM Symposium on Principles of Distributed Computing (PODC), 1996, pp. 322–330.
[15] T. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. Assoc. Comput. Mach. 43 (2) (March 1996) 225–267.
[16] K. Cho, K. Birman, A group communication approach for mobile computing, Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, California, December 8–9, 1994; Also Technical Report CS TR94-1424, Cornell University.
[17] G.V. Chockler, I. Keidar, R. Vitenberg, Group communication specifications: a comprehensive study, ACM Comput. Surveys 33 (4) (December 2001) 1–43.
[18] D. Dolev, D. Malkhi, The Transis approach to high availability cluster communication, Commun. Assoc. Comput. Mach. 39 (4) (April 1996) 64–70.
[19] P. Ezhilchelvan, R. Macêdo, S. Shrivastava, Newtop: a fault-tolerant group communication protocol, in: Proceedings of the IEEE 15th International Conference on Distributed Computing Systems Vancouver, 1995, pp. 296–306.
[20] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, J. Assoc. Comput. Mach. 32 (2) (April 1985) 374–382.
[21] S. Frolund, R. Guerraoui, Implementing e-transactions with asynchronous replication, Internatinal Conference on Computational Systems and Networks (DSN 2000), New York, June 2000, pp. 449–458.
[22] A. Fuggetta, G.P. Picco, G. Vigna, Understanding code mobility, IEEE Trans. Software Eng. 24 (5) (May 1998).
[23] B. Glade, K. Birman, R. Cooper, R. van Renesse, Lightweight process groups in the Isis system, Distributed Systems Eng. 1 (1993) 29–36.
[24] R. Guerraoui, A. Schiper, Consensus service: a modular approach for building fault-tolerant agreement protocols in distributed systems, in: Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26), Japan, June 1996, pp. 168–177.
[25] D. Johansen, K. Marzullo, F.B. Schneider, K. Jacobsen, D. Zagorodnov, NAP: practical fault-tolerance for itinerant computations, in: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99), Austin, TX, USA, May–June 1999.
[26] I. Keidar, J. Sussman, K. Marzullo, D. Dolev, A client-server oriented algorithm for virtually synchronous group membership in WANs, in: 20th International Conference on Distributed Computing Systems (ICDCS), April/2000, pp. 356–365.
[27] B.W. Lampson, Reliable messages and connection establishment, in: S. Mullender (Ed.), Distributed Systems, Addison-Wesley, Reading, MA, 1993.
[28] R.J.A. Macêdo, F.M. Assis Silva, Mobile groups, Technical Report RI001/01, LaSiD/UFBA (Distributed Systems Laboratory/Federal University of Bahia), February 2001.
[29] R.J.A. Macêdo, F.M. Assis Silva, Coordination of mobile processes with mobile groups, in: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, DSN2002, Washington, DC, June 23–26, 2002, pp. 177–186.
[30] S. Mishra, P. Xie, Interagent communication and synchronization support in the D'Agent mobile agent-based computing system, IEEE Trans. Parallel Distributed Systems (TPDS) 14 (3) (March 2003).

ARTICLE IN PRESS

[31] A. Murphy, G.P. Picco, Reliable communication for highly mobile agents, Proceedings of the Joint Symposium ASA/MA'99, October 1999.

[32] ObjectSpace, Voyager—ORB 3.1 Developer Guide, Object Space, Inc. 1999.

[33] S. Pleisch, A. Schiper, Modeling fault-tolerant mobile agent execution as a sequence of agreement problems, in: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS), Nuremberg, Germany, October 2000, pp. 11–20.

[34] G.P. Picco, A.L. Murphy, G.-C. Roman, Linda meets mobility, in: D. Garlan, J. Kramer (Eds.), Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles (USA), ACM Press, New York, 1999.

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

[36] R. Prakash, M. Raynal, M. Singhal, An efficient causal ordering algorithm for mobile computing environments, in: Proceedings of the 16th International Conference on Distributed Computing Systems, May 27–30, 1996, Hong Kong, IEEE Computer Society, New York, 1996.

[37] M. Ranganathan, M. Bednarek, D. Montgomery, A reliable message delivery protocol for mobile agents, Proceedings of the Joint Symposium ASA/MA2000, September 2000.

[38] R. Renesse, K. Birman, R. Cooper, B. Glade, P. Stephenson, The horus system, in: K. Birman, R. Renesse (Eds.), Reliable Distributed Computing with the Isis Toolkit, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 133–147.

[39] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Verissimo, K. Birman, A dynamic light-weight group service, in: 15th IEEE International Symposium on Reliable Distributed Systems (SRDS) (October 1996); pp. 23–25.

[40] K. Rothermel, M. Strasser, A fault-tolerant protocol for providing the exactly-once property of mobile agents, in: Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS), West-Lafayette, Indiana, October 1998, pp. 100–108.

[41] A. Schiper, Early consensus in an asynchronous system with a weak failure detector, Distributed Comput. 10 (3) (April 1997) 149–157.

Raimundo José de Araújo Macêdo is a Professor of Computer Science at Federal University of Bahia (UFBA) in Brazil, where he founded the Distributed Systems Laboratory (LaSiD) in 1995. Currently, he is the head of LaSiD and coordinates the Masters Program on Mechatronics. He received a B.Sc, M.Sc., and Ph.D. in Computer Science from UFBA, University of Campinas (Unicamp/Brazil), and University of Newcastle upon Tyne (England), respectively. His research interests include the many aspects of dependable distributed systems. He has served as a PC member on a number of conferences, including IEEE/IFIP International Dependable Systems and Networks Conference (DSN) and the Brazilian Symposium on Computer Networks (SBRC).

Flávio M. Assis Silva received his BSc, MSc and PhD in Computer Science, respectively, from the Federal University of Minas Gerais, Brazil (1999), University of Campinas (Unicamp), Brazil (1993), and Technical University Berlin, Germany (1999). Since 1999 he is a researcher at LaSiD, the Distributed Systems Laboratory of the Federal University of Bahia (UFBA), Brazil, and since 2001 he holds a position as lecturer at the Department of Computer Science of the same University. His main research interest areas are mobile agent systems and applications, reliable distributed system, and wireless sensor networks.