Coordinating Mobile Agents through the Broadcast Channel

Vera Nagamuta[1] *and Markus Endler*[2]

[1]Instituto de Matemática e Estatística
Universidade de São Paulo
Rua do Matão, 1010
05509-900 São Paulo, Brazil
nagamuta@ime.usp.br

[2]Departamento de Informática
PUC Rio
Rua Marquês de São Vicente, 225 – Gávea
22453-900 Rio de Janeiro, Brazil
endler@inf.puc-rio.br

Abstract

In distributed applications based on mobile agents, coordination and synchronization of the actions executed by a team of mobile agents are difficult tasks. The main difficulty comes from the fact that coordination requires the agents to interact with each other in spite of their dynamically changing locations.

In this paper we present a mechanism for coordinating mobile agents which handles the problem of locating and addressing members in a group of mobile agents. This mechanism, which we called *Broadcast Channel*, implements reliable broadcasts of messages to a group of mobile agents, independently of their current locations.

Keywords: mobile agent, coordination, broadcast

1 Introduction

Much work has been done in the design and implementation of execution environments for mobile agents [24, 15, 23, 1], but until now only few environments provide higher-level services that give support for the coordination among mobile agents. In our research we focus on exactly this problem: How to support coordination for groups (teams) of mobile agents. This problem is complex mainly due to the lack of a fixed address (location) of the agents.

Coordination is mainly required for distributed programs consisting of a team of cooperating agents, where each agent is responsible for performing part of a common, global task. Teams of mobile agents are likely to become the means to implement several distributed and networked applications in the future [11]. For example, one possible application is the search for some information in the network, to be performed in parallel by a group of agents and which are supposed not to visit a same host more than once. Another application could be a network management task, where a set of agents is in charge of executing a system-wide, consistent reconfiguration of software modules of a distributed program.

Some coordination models for mobile agents have been proposed [15, 3, 7, 6], and have been classified by Cabri *et al*[5], but all of them require either spatial or temporal coupling.

The main contribution of our work is a new approach for achieving coordination among mobile agents, which is based on a mechanism which we called *Broadcast Channel*. The main characteristic of our approach is that location management is entirely separated from the application-specific inter-agent communications. With our approach, it is possible to program communications among agents without worrying about their current locations. Instead, it is the mechanism's responsibility to keep track of the current location of agents and to guarantee that the broadcast messages are delivered to all agents in the team. Thus, our mechanism facilitates the programming of coordinated activities (e.g. migrations) within teams of mobile agents.

The remainder of the paper is organized as follows: in section 2 we discuss some group communication systems and compare them with our mechanism. In section 3, we present some coordination models proposed for mobile agents. In section 4, we present our coordination mechanism, its basic elements and functioning, and section 5 presents an informal description of the protocol used to implement our mechanism. In section 6, we give an example showing how to use our mechanism to achieve mutual exclusion among a set of agents. Section 7 discusses results of some performance tests performed with our proto-

type implementation. Finally, in section 8 we make some concluding remarks.

2 Related Work

Since the early 80's much effort has been invested in the development of many sorts of group communication mechanisms and services for distributed systems, because it was recognized that they are a fundamental building block for many distributed protocols. Among many developed group communication protocols and tools, the best-known are probably the early Isis [4], Horus [21] and Transis [2], and the more recent ORB-based systems, such as Orbix+Isis [12], Electra [18] and OGS [10]. Most of these works were focused on reliability and fault-tolerance, but reliable group communication is also a fundamental service for coordination and consensus protocols.

As for Mobile Agents, only few works have incorporated group communication mechanisms into the agent programming environments. To our knowledge, only Tacoma [14, 13] and Concordia [25] provide some support for group communication in a mobile agent environment.

Project Tacoma focuses on operating system support for mobile agents. Its execution environment consists of a Tcl interpreter that uses the Horus library. The advantage of the approach is that Horus's robustness and rich functionality is made available to mobile agent systems. The drawback is a lack of an integrated language model which exposes the agent programmer to the details of the Horus environment.

Concordia does not actually support group communication in the traditional sense, but instead provides a base class called *AgentGroupImpl* which can be extended by the application programmer with application-specific *group operations*. Invocations of these operations by mobile agents are then forwarded to a *group daemon* executing at a specific host. Subclasses of *AgentGroupImpl* are thus implemented as a centralized agent that is used for storing data sent by the group of migrating agents, but which is unable to deliver messages to the agents asynchronously.

Compared with these works, the *Broadcast Channel* is based on message diffusion, which has the advantage of allowing asynchronous delivery of messages to the group members (the agents). Moreover, the mechanism guarantees delivery of messages to *all* group members in the *same order*, even if some agents are temporarily non-available due to a migration. Our mechanism is also seamlessly integrated into the mobile agent execution model of ASDK (Aglets Software Development Kit [17]). Moreover, it handles all issues related to agent location transparently to the the programmer, so that he (or she) can focus on the application-specific interaction protocols in the agent groups.

Actually, in our work, as well as in Tacoma and Concordia, the main goal has been to support agent cooperation rather than enhancing fault-tolerance. In the next section we present some coordination models for mobile agents, showing the advantages and drawbacks of each of them before presenting our mechanism.

3 Coordination Models for Mobile Agents

Cabri, Leonardi and Zambonelli [5, 6] presented a taxonomy of coordination models for mobile agents based on *spatial* and *temporal coupling*, and defined four categories of coordination models:

- **Direct Coordination.**
  In this model, the mobile agents send messages directly to each other. This model requires spatial coupling, because the sender must know the receiver's identity and temporal coupling, because the receiver must be active during the communication.

  This interaction mechanism has been incorporated in most programming systems for mobile agents, such as Sumatra [1] and AgentTcl [15]. In spite of its usefulness, it is not well suited for coordinating mobile agents, mainly because it is based on peer-to-peer communication, requires the applications to track the current location of agents, and usually does not give support for deferred message delivery, e.g. when the message destination is not currently reachable.

- **Meeting-oriented Coordination.**
  In the meeting-oriented models, mobile agents interact through meetings points, i.e. the place where a meeting can occur. In this model, the agents must enter a given meeting point in order to communicate and synchronize with other agents. Meetings points impose a locality constraint, since only local agents can participate in the meeting.

  This model solves the problem of locating agents, found in direct coordination, but requires agents to know the meeting point. Moreover, it requires synchronization among the agents, e.g. they must be colocated at the meeting point during at a certain period of time in order to be able to interact with each other. Examples of systems based on this coordination model are Ara [20] and MOLE [3].

- **Blackboard-based coordination.**
  In this model, the agents interact through shared message repositories at each place, called *blackboards*, i.e. the sender puts a message on the blackboard and the receiver can either read or retrieve the message from the blackboard.

  The main advantage of this model is the temporal uncoupling: messages are left on the blackboard no matter where the corresponding receivers are or when they will read the message. The drawback of backboard systems is the spatial coupling: the agents have to visit the correct place and agree on common messages types and formats. Ambit [7] is an example of a system using the blackboard-based coordination.

- **Linda-like Coordination.**
  Linda-like coordination is also based on a shared namespace, but unlike blackboards, it uses an associative *tuple space* where information is organized as tuples and can be accessed and retrieved through pattern-matching.

  The main advantage of the Linda-like coordination is its temporal uncoupling and partial spatial uncoupling. Although it does not require agent synchronization (e.g. meeting at a certain place), the patterns used to access the tuple space embody some implicit knowledge of the peer agent's interaction requirements. *MARS* [6] implements a variant of the Linda-based coordination, called reactive blackboard, where changes to the tuple space can be automatically mirrored on several places.

Each of the above coordination models requires some form of temporal or spatial coupling among agents, and most of them do not handle the problem of locating the peer agent or the shared data space required for interaction. In our work we follow an approach to mobile agent coordination based on message broadcasts within groups of agents. In this approach, spatial coupling is present only as the requirement to be member of the group, and there is no temporal coupling, meaning that all broadcast messages will eventually be delivered in the same order to all group members.

4 Coordination via the Broadcast Channel

The choice of message broadcasting as the primary means for coordinating mobile agents was motivated by our understanding that many synchronization and consensus algorithms are based on message diffusion. The main idea was to design a mechanism supporting message diffusion within groups of migrating elements, e.g. the mobile agents. We called this mechanism *Broadcast Channel* [19].

One of the components of the Broadcast Channel is an element called *Broadcast Proxy* (or simply *Proxy*), which executes at a fixed address and acts as main representative of the group. It is responsible for processing broadcast request issued by agents and making sure that all members of the group do in fact receive the broadcasted message. Thus, for each team of cooperating agents a *Proxy* must have been instantiated at some *place*.

The *Proxy* is similar to the *AgentGroup* of Concordia [9], but it differs in that it has the additional function of broadcasting messages to the group members and holding a local copy of the messages until all agents have sent acknowledged its receipt.

Location management, i.e. tracking the agent's locations, is also responsibility of the *Broadcast Channel*, and is done as follows. Each of the *places* in this system is associated with a *domain*, and within each domain there is a specific place, called *reference place*. While this *reference place* maintains the information about the current location (e.g. a *place*) of every mobile agents in that domain, the *Proxy* only keeps track of the current domain where each mobile agent is currently located. Thus, the *reference places* act as intermediates between the *Proxy* and the mobile agents for locating and delivering broadcast messages. The main reason for using such an architecture with this number of levels was our understanding that it is a good trade-off between the overhead of message forwarding (by the intermediates) and the costs of handling location updates due to agent migrations.

In order to perform its intended task, the *Broadcast Channel* requires an agent deployment infra-structure, and makes the following assumptions about the execution environment:

1 there is no loss or corruption of agent-to-agent messages;

2 agents within a group start migrating only after the *Broadcast Channel* (representing the group) has been properly instantiated and configured;

3 only static groups are supported, i.e. mobile agents cannot enter or leave a group dynamically;

4 all components that implement the *Broadcast Channel* are trustful and always available, i.e. there are no failures of hosts or problems with the agent infrastructure;

5 agents migrate only to *place* that are within a registered domain and migration takes a finite time;

6 agents can migrate arbitrarily often, but eventually each agent must stay a sufficient long period of time at a *place*;

7 message transmissions can have an arbitrary, but finite duration.

If all of the above assumptions about the execution environment are met, the *Broadcast Channel* guarantees that all broadcast messages are eventually delivered to all group members, and that delivery is in total order.

4.1 Basic Concepts and Architecture

The *Broadcast Channel* implementation is based on the following elements:

- *place*: corresponds to the execution environment for mobile agents. While executing at a place, an agent may: (a) interact with other agents at the same *place*; (b) send messages to agents at other places or (c) request to be dispatched to another *place*. Each *place* must be assigned to a single host, but a host can have several places.

- *domain*: is a set of *places* with a singular representative, called *reference place*.

- *reference place*: is the representative of the domain. It is a *place* with the following additional functions: (a) hold references and forward messages to mobile agents within the domain; (b) update the reference to an agent when it changes place, and (c) interact with other reference places when agent migrates to another domain.

- *Proxy*: is a singular element with a fixed address, which is known by all *reference places* and mobile agents participating in a group. The *Proxy* maintains information about the *reference places*, the mobile agents that are members of the group and the domains where they are currently located. Its main responsibilities include: (a) broadcast of messages received from an agent that is member of the group, (b) receive and keep track of acknowledgements for each broadcasted message, (c) maintain an updated record of agent locations (in terms of domains) and (d) re-transmit messages when inter-domain migration is notified.

- *message*: is the communication object of this mechanism. Each message has a unique identifier given by the *Proxy*, and is composed of a CONTROL component, used by the communication protocols within the *Broadcast Channel*, and the APPL component, which carries application-specific data.

Figure 1 depicts the architecture of the *Broadcast Channel*. Each rectangle represents a domain (D_1 , D_2‘, ..., D_n) with the corresponding *reference places* (RP_1 , RP_2 , …RP_n), represented by hachured rectangles, a set of *places* (P_1 , P_2 , … , P_j), represented by white rectangles and a set of mobile agents represented by circles. We assume that all the mobile agents in this figure are already registered with the *Proxy*. Arrow ① shows a mobile agent MA_1 sending a message to the *Proxy*. In ②, this message is broadcasted by the *Proxy* to the *reference places*. In ③, each *reference place* is sending the message to *places* (within its domain) holding some agent, and in ④, the *places* are forwarding the message to the mobile agents. In ⑤, the mobile agents are sending na acknowledgment to the *Proxy*.

The choice of this architecture has two main advantages. First, by managing location information at two levels (at the *Proxy* and at each of the *reference places*) most of the updates related to agent migrations can be handled by the local *reference places* and need not be send to the *Proxy*. Only in the case of an inter-domain migration, the *reference place* of the involved domains exchange some information and the *Proxy* is notified about this migration, making it possible for it to update its information about the agent location, and to retransmit non-acknowledged messages to the new *reference place*.

Second, this architecture decentralizes the task of message broadcasting, by delegating part of the work to the *reference place* in each domain.

5 Informal Description of the Protocol and Messages

The protocol used to implement the *Broadcast Channel* is based on a set of control messages exchanged among its elements (see Table 1).

The protocol also defines messages for the registration and de-registration of agents. In order to be able to use the *Broadcast Channel*, the agent must first register itself with some *Proxy* at creation time. This is done by sending a registration message (REGISTRY_AGENT) directly to the *Proxy*. The *Proxy* announces the new agent registered to all *reference places*.

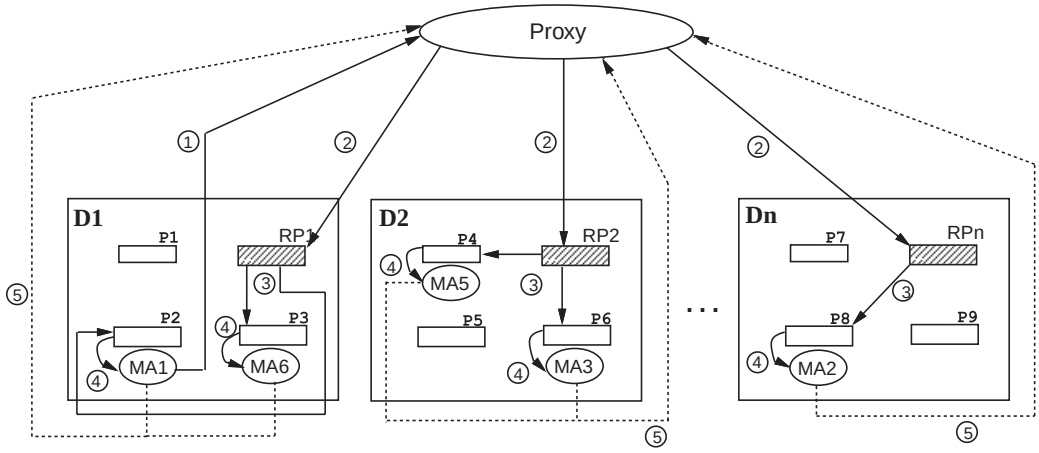

Figure 1: System Architecture

In order to ensure the reliable delivery of a message, the *Proxy* assigns it a unique identifier *MessageID* (e.g. a sequence number), which is also used to implement total ordering. Mobile agents handle the messages according to the MessageID and are able to detect missing or duplicated messages.

When an agent moves to another *place* it may not receive some messages which were sent during its migration, and which then have to be re-sent to the agent. For this, both the *Proxy* and *reference place* store all sent messages until all agents (members of the group) acknowledge their receipt, by sending message CONFIRM(AgentID) to the *Proxy*. When all acknowledgments for a message have arrived, the *Proxy* removes the corresponding entry from its records and sends a request to *reference places* to remove the message also from their messages queues.

5.1 Intra- and Inter-domain Migrations

There are two cases of migration: intra-domain and inter-domain migrations. In both cases it is necessary to update the information about the current location of the migrated agent, and to re-send messages.

Before a migration, the agent sends a GOING_TO(AgentID) to the *place* where it is currently executing and dispatches itself to another *place*. When it reaches the new *place*, it sends an ARRIVAL message indicating the *reference place* of the domain where it came from, the identifier of the last handled message, and identifiers of messages not handled yet (called *message repository*). The first argument is used to identify if the migration was intra- or inter-domain, while the second and third are used to identify which messages must be re-sent to the agent.

After receiving message ARRIVAL the *place* sends a REGISTRY message (with the same arguments of ARRIVAL plus its own identifier) to the *reference place* of the domain.

Through the message REGISTRY, the *reference place* is able to determine if the migration was intra- or inter-domain. In the first case, the *reference place* updates the agent's address locally, checks which of the messages have not been received by the agent (comparing its local message queue and the received *message repository*) and eventually re-sends messages to the agent. Thus, in the case of intra-domain migration, the *Proxy* does not receive any notification.

Inter-domain migration requires more interactions among the *reference places* and the *Proxy*, and in this case the *Proxy* is also responsible for the message retransmissions. When an agent moves from the domain with *reference place* RP_1 to domain with *reference place* RP_2, the RP_1 must delete its information about the agent. For this purpose, RP_2 sends a DEREGISTRY message to inform RP_1 that the agent is leaving its previous domain. RP_2 also locally registers the agent as pertaining to its domain and notifies the *Proxy* about the migration through a UPDATE message carrying its own identification (i.e. the new *reference place* address). With this message, the *Proxy* will update also its registry about the agent's location (i.e. its current domain), check for non acknowledged messages and re-send them to the new *reference place*.

Table 1 summarizes all control messages mentioned above, indicating the transmitted arguments and its purpose, where MA stands for Mobile Agent, RP for reference place and P for *place*.

Msg Type	Arguments	Origin - Destiny	Meaning
REGISTRY_AGENT	AgentID, RefPlaceAddr, PlaceAddr	MA → Proxy	Registers a new MA at the Proxy
GOING_TO	AgentID	MA → RP, MA → P	Notifies the P or RP where is located that it is migrating to oder P or RP
ARRIVAL	AgentID oldRefPlaceAddr msgRep	MA → RP MA → P	Notifies that MA is entering to this P or RP
REGISTRY	AgentID oldRefPlaceAddr newPlaceAddr, msgRep	P → RP	Registers the MA in the corresponding migrations
DEREGISTRY	AgentID	RP → RP	De-registers the MA from the old RP

Table 1: *Broadcast Channel* Control Messages

5.2 A Migration Scenario

In order to illustrate how the *Broadcast Channel* handles message retransmissions caused by inter- and intra-domain migrations, Figure 2 shows a scenario from the moment a broadcast message is sent to a mobile agent, until the acknowledgment from the agent is received by the *Proxy*. For the sake of simplicity, this scenario assumes a group with a single mobile agent.

For this scenario, we assume the existence of two domains D_1 and D_2, where the first has *reference place* RP_1 e and *places* P_1 and P_2, and the second has *reference place* RP_2 and places respectively P_3 and P_4.

Suppose that the mobile agent (MA) is initially located at *place* P_1, has not received any messages before the *Proxy* "broadcasts" message m_1, and that before it receives this message from P_1 it migrates to *place* P_2 (intra-domain migration – ①). When RP_1 receives the REGISTRY (ID,RP_1,P_2,*empty*) message, it identifies that the migration is intra-domain (comparing the first argument) and that the agent did not receive any message yet (since the last argument, is empty) and then, re-sends m_1.

Now let's assume that meanwhile the agent migrated to *place* P_3 (in ②), of domain D_2. In this case, RP_2 identifies this migration as inter-domain migration and notifies the *Proxy* and RP_1. When the *Proxy* receives message UPDATE (in ③), it checks for non-acknowledged messages (in this case m_1), and re-sends it, now to *reference place* RP_2.

The agent finally receives the message and sends a CONFIRM (m_1,ID) message to the *Proxy*. Finally, the agent moves to *place* P_4, but since the message has been acknowledged, it is not sent again.

6 Example

In this section we give an example showing how the *Broadcast Channel* can be used to solve a specific coordination problem, i.e. mutual exclusion, and describe the development steps we followed for the implementation of this example.

An example for agent coordination requiring mutual exclusion could be a team of agents in charge of visiting a set of places, and where one wants to avoid that a place is visited by more than one agent. In order to achieve this coordination, before migrating to a new place, agent would have to notify all other agents about the place it plans to visit next. Such notifications could be naturally implemented with the *Broadcast Channel*.

Yet another example would be if a team of agents (say T_1) is supposed to find any member of another team of agents (T_2), and where at most one pair of different agents should start an interaction. This sort of restriction may be important for applications where inter-team transactions must only be performed once, and where individual agents have the autonomy to perform actions without any further query to a central database.

In order to test our mechanism, we used Lamport's well-known mutual exclusion algorithm [16] (based on logic clocks) and developed a simple agent-based application program that implements this algorithm using the *Broadcast Channel*. Although Lamport's algorithm is quite simple, and despite the fact that several optimizations of it have been suggested [22, 8], it was well suited for our purpose, since it is based on broadcasts. In Figure 3 is shown part of the pseudo-code for the agents, structured as statements of the form *event =>action.*

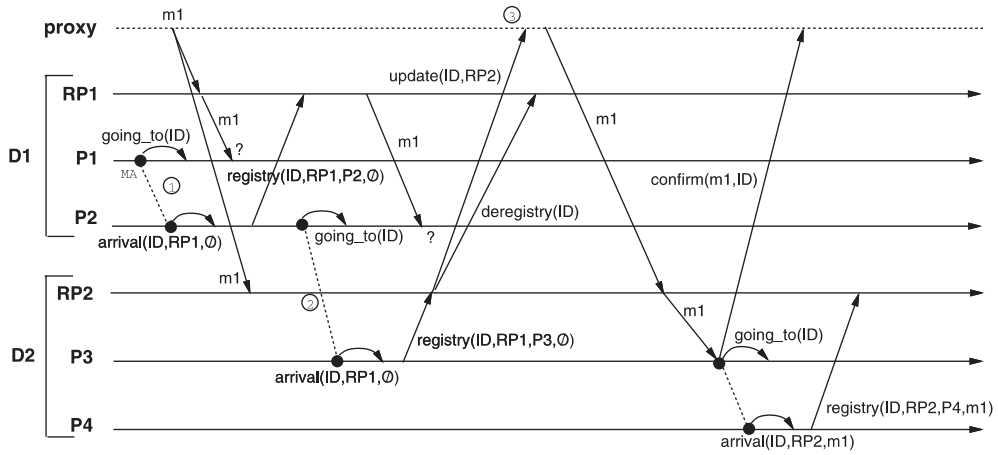

Figure 2: Example of Inter- and Intra-domain Migrations

The main idea of the original algorithm is as follows. Each of the *N* process maintains a Lamport timestamp (a counter) and an array status of size N, where element status[i] contains the latest message (either of type REQ, ACK or REL) received from process i, with the corresponding timestamp of the sender. Whenever a process requires the mutual exclusiveness, it broadcasts a REQ message to all processes, including itself. The receipt of this message is acknowledged by each other process with an ACK message and its local timestamp. Thus, REQs and ACKs sent from any process are stored in the corresponding entries of status, at every process.

Whenever a process gives up its right for exclusiveness, it broadcasts a REL message with its identifier and current timestamp, and array status at all processes are updated accordingly. Then, every process checks the elements in status to find out which is the pending request with the oldest (i.e. lowest) timestamp [1], and the corresponding process takes its exclusive access right. Since all processes maintain their status arrays perfectly synchronized, and local clocks are updated according to Lamport's method, the algorithm in fact implements mutual exclusion.

In order to use Lamport's algorithm with the *Broadcast Channel* we made the following minor adaptations: One of the *Proxy*'s arguments is the size *N* of the group, i.e. the number of mobile agents participating in the mutual exclusion algorithm. As soon as the *Proxy* receives *N* registration messages from the participants, it broadcasts the list of all agentIDs to all agents, which perform an identical mapping from agentIDs to status indices. After the initial set-up, all mobile agents use the *Broadcast Channel* as the only means of communication with the other agents. Each broadcast request for the *Proxy* is a string of the form (TYPE, ID, TS), where TYPE is either REQ, REL or ACK , ID is the agent's Id and TS is its current timestamp. Notice these are the application specific messages, as opposed to the CONTROL messages used to control the mechanism per se. In order to test the mutual exclusion, we also implemented a stationary resource agent , which logs all accesses to it.

For the implementation of this and other examples [19], we followed the sequence of steps below:

1 Define the application specific messages (in this case, messages REQ, REL and ACK);

2 Extend the class **`Agent`** (of our mechanism API) with application specific attributes and functions (in this example we created two kinds of agents: a request agent and a resource agent);

3 Extend the class **`Proxy`**;

4 Define and create the infrastructure of the *Broadcast Channel* (section7) and instantiate the agents developed in step (2).

Through our implementation of this mutual exclusion application, we confirmed that at each moment only one request agent accessed the resource agent, and hence showed that all broadcast messages issued via the *Broadcast Channel* were received and handled correctly.

1 In the case of requests with same timestamp, the algorithm gives higher priority for the requests originated from the process with lower process-Id.

```
Agent {
        Array status [N];                   //N: size of mobile agents group
        int ts = 1;                         //local clock
        BOOLEAN granted = FALSE;            //indicates the access permission to the resource
        int id;                             //the agent´s identifier (values between 1,N)

        Initialize_status(int N) {
                //initializes the array status inserting REL with ts = o for all elements
        }
        boolean min(int i) {                                                              10
                //returns TRUE if the element at the index i of array status has the lowest ts of the array
        }
        OnRequest => {
                //sends (REQ,id,ts+1) to Proxy if status[id] <> ´REQ´
        }
        OnRelease => {
                //send (TEL,id,ts+1) to Proxy if status[id] = ´REQ´
        granted := FALSE;     //the agent has no more access permission
        }
        OnReceive(m) => {                                                                 20
        Ts := max (ts,m.TS) + 1;
        if (m.TYPE= ´REQ´) {
                Status[m.ID] := m;      //insert the REQ message at the position corresponding to the sender
                // sends (ACK,id,ts+1) to Proxy
        }
        else if (m.TYPE = ´REL´) {
                status[m.ID] := m;
        }
        else if (m.TYPE = ´ACK´) {      //if the message is na ACK and the previous one from the same
        if(status[m.ID] <> ´REQ´)       //sender is not a REQ, ACK is stored                30
                status[m.ID] := m;
        }
        //verifies if the agent did a REQ and if the ts is the lowest
        if(status[id] = ´REQ´ AND min(id) AND NOT granted)
                granted := TRUE;        //obtain the access permission
        }
}
```

Figure 3: Pseudo-code for the agent

7 Prototype Implementation and Tests

The *Broadcast Channel* was implemented using the Aglets Software Development Kit (ASDK) [17], an agent programming environment developed at the IBM Tokyo Research Laboratory. The main elements of the *Broadcast Channel* infra-structure: the *Proxy*, *reference places* and *places* are all implemented as stationary *aglets*[2] (i.e. classes **`Proxy`**, **`Place`** and **`Reference Place`** are extensions of ASDK's **`Aglet`** class [3]**).**

These elements are instantiated, configured and dispatched to hosts by an aglet called **`Infrastructure-Launcher`**, which provides a graphical interface to the user. Through this interface, the user creates the *places*, chooses at which host they will execute, defines the *domains* and the locations of the *Proxy* and the *reference places*. After the set up of the *Broadcast Channel* configuration is finished, it is expected to remain fixed, i.e. no *places* or *reference places* can be added or removed. Finally, mobile agents can be instantiated at the various *places* and their first action will be their registration with a *Proxy*.

In order to use the *Broadcast Channel* for a specific application, classes **`Proxy`** and **`Agent`** have to be extended by the programmer, in order to add application-specific attributes and functions.

7.1 Tests

In this section we present the results of some tests that aimed at measuring the overhead incurred by the *Broadcast Channel* both for stationary agents, and for two pat-

2 In ASDK, agents are programmed in Java and are called *aglets* (agent+applet).

3 The source code of the *Broadcast Channel* is available at http://www.ime.usp.br/~nagamuta/bchannel.html/.

terns of agent migrations. We called the first ones static tests and the latter dynamic tests. In all tests, we measured the time from the moment the *Proxy* broadcasts a message until it receives the last acknowledgment for the message. The tests were run on SparcStations (60-167 MHz) executing Solaris 5.7 as clients in our network. It is worthwhile to note that the tests were not aimed at providing a complete and detailed picture of the mechanism's performance, but only to identify which factors do most contribute to the mechanism's overhead.

In the static tests we compared the performance of the *Broadcast Channel* with that of a simple ASDK program which broadcasts messages to a set of stationary agents, by using ASDK's message passing facility (method sendAsychMessage). We called it *direct sending*. The goal with these tests was to assess how much overhead the message forwarding within *Broadcast Channel* produces for different numbers of agents and domains.

Figure 4 represents the result of a static test done with 4 hosts, 9 *places* and 3 domains (for the *Broadcast Channel*).

We considered groups of mobile agents with 3, 6, 12, 24, 36 and 48 agents, and we took the mean value from measures for 60 broadcasts for each group size.

As expected, and shown in Figure 4 , the time required for *direct sending* is lower than the one using *Broadcast Channel*, but the second is always less than the double of the first one.

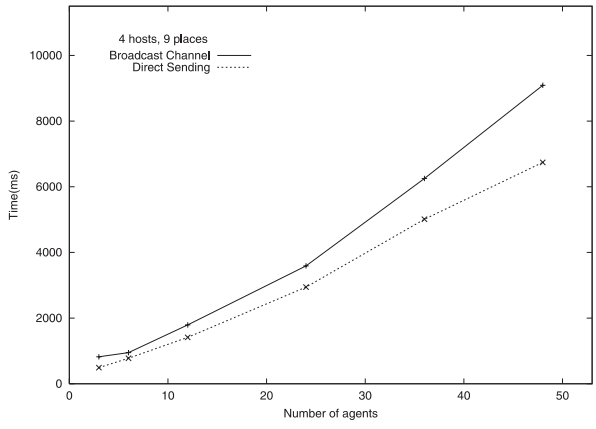

Figure 4: Static test with 9 places

The same test was done with 2 hosts, 6 places and 4 hosts, 6 *places*, considering the same number of agents. The result of the direct sending for these three cases was almost the same but, as shown in Figure 5 in which we have just the results for the *Broadcast Channel*, we can see that the best performance of the *Broadcast Channel* is achieved when we have a greater number of hosts and *places*. This is due to the descentralized processing within the *Broadcast Channel*, and the advantages of having less number of agents per domain.

In the dynamic test we defined a fixed, but arbitrary, itinerary for each mobile agent, and compared two different stay periods (i.e. the period of time each agent stays at a place). Thus, a short stay period means high migration frequency, and vice-versa. We compared the times required by the *Broadcast Channel* to broadcast a number of messages to different numbers of agents when their stay times are high and when they are low. In these tests, we used a single stationary agent whose only task was to periodically request a broadcast to the group of mobile agents.

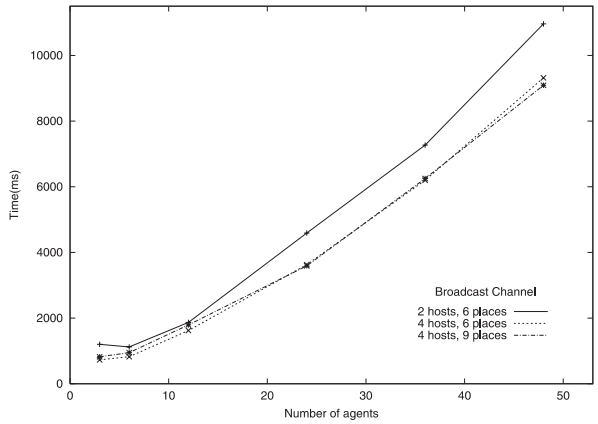

Figure 5: Comparison

The goal of these dynamic tests was to assess how much the mechanisms performance degrades when the migration frequency increases. Although we did tests only for two stay periods (8 and 25 seconds), it was sufficient to notice that it takes more than the double of time to deliver (and get acknowledgments for) all messages when a significant number of agents (more than 15) are frequently migrating among six places within three domains (as shown in Figure 6). The main reason for such increase in message delivery times is the fact that due to the frequent migrations, much more messages have to be re-transmitted, either by the *Proxy* or by the *reference places*.

We also measured the delivery times for the same two stay periods (8 and 25 seconds), but with a configuration of nine places (instead of six places) and three domains. Figure 7 shows that with this configuration, the increase of delivery times is much slower. This is probably caused by the fact that since in this case we have more places per domain, there is a lower probability of occurrence of the more costly inter-domain migrations.

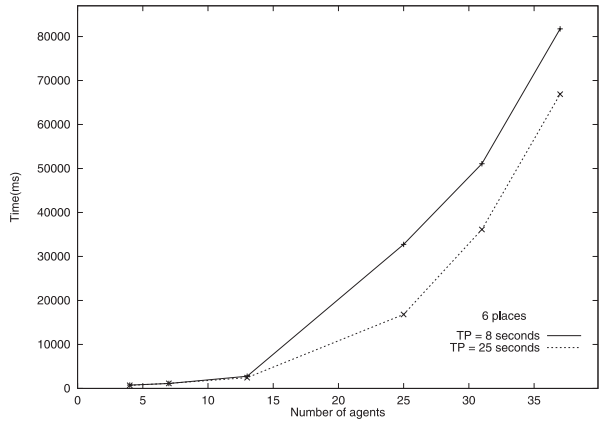

Figure 6: Dynamic test with 6 places

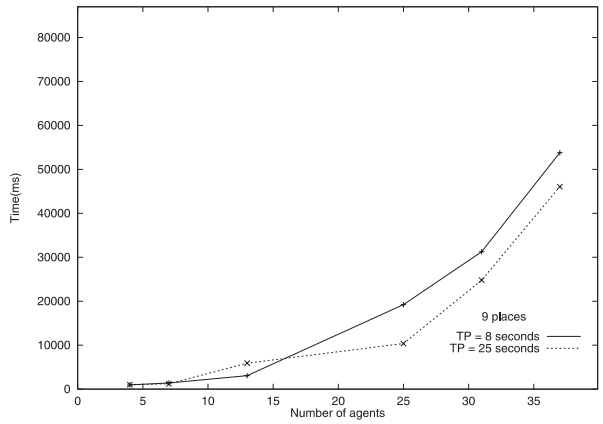

Figure 7: Dynamic test Figure 7: Dymanic test with 9 places

Based on the results of both sorts of tests, we noticed that the *Broadcast Channel* achieves best performance when the frequency of all agent migrations within a group is relatively low, and when the probability of inter-domain migrations is much lower than that of intra-domain migrations.

We are aware that in order to get more accurate information about the *Broadcast Channel* performance, much more tests would be required. In the future we plan to perform also some tests where *places* and *reference places* are shared by more than one *Proxy*, hoping to identify places where the mechanism can be optimized.

8 Conclusion

In this paper we presented a coordination mechanism called *Broadcast Channel*, which offers an alternative way to coordinate groups of mobile agents. The main purpose of the mechanism is to facilitate the task of programming teams of cooperating mobile agents. The main benefit is that location management is completely separated from inter-agent communication. We implemented a prototype of the *Broadcast Channel* IBM's Aglets Software Development Kit (ASDK) [17].

When comparing our mechanism with other coordination models for mobile agents (see section 3), we think it has the following advantages:

- There is no need for the agents participating in a group to know each other's identity or current location. Instead, all messages are broadcasted to all agents registered with a *Broadcast Channel* regardless of their current location, and are eventually delivered even if an agent is migrating between *places*;
- The mobile agents do not need to be synchronized or be located at the same *place* to interact with each other, as in the meeting-oriented coordination model;
- Because the *Broadcast Channel* requires only a message passing mechanism, it can be implemented on the top of most mobile agent execution environments, given that the assumptions listed in section 4 are all satisfied.

The main drawbacks of our mechanism are the following:

- In order to use the *Broadcast Channel* a mobile agent must know the address of the *Proxy* corresponding to the group of which it should be a member. This information must be provided by the user at the agent instantiation;
- Each broadcast request to the Broadcast Channel causes an additional overhead due to message retransmission, as shown by the static tests. Agent migrations are the major cause of the overhead produced by the *Broadcast Channel*, specially inter-domain migrations, since they require both more control messages and the retransmission by a central element.

Despite these problems, we believe the Broadcast Channel to be an useful tool for developing cooperating teams of mobile agents, mainly because the programmer has not to deal with the issue of tracking the agents locations. However, the Broadcast Channel might be useful mainly for applications with low performance requirements, where domains have many places, and where migrations are not so frequent.

As future steps, we plan to use our mechanism for a real-world agent-based applications, such as network-wide software configuration. Moreover, we plan to re-evaluate our design and protocols, looking for possible optimizations.

Acknowledgement

This project is partially supported by CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) and FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) - Grants No. 98/06138-2 (Project SIDAM) and No. 00/08742-6.

References

[1] A. Acharya, M. Ranganathan and J. Saltz. Sumatra: a Language for Resource Aware Mobile Programs. Mobile Object Systems, Lecture Notes in Computer Science, Spring Velarg(D)(1222):111-130, February 1997.

[2] Y. Amir, D. Dolev, S. Kramer and D. Malki. Transis: A Communication Subsystem for High Availability. Tech. Report TR CS91-13, Computer Science Dept., Hebrew University, Jerusalem, 1991.

[3] J. Baumann, F. Hoh, K. Rothermel and M. Strasser. Mole - Concepts of a mobile agent system. *World Wide Web*, 1(3):123-137, 1998.

[4] K. P. Birman and R. V. Renesse. Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press, Los Alamitos, 1994.

[5] G. Cabri, L. Leonardi and F. Zambonelli. How to Coordinate Internet Applications based on Mobile Agents. *IEEE Seventh International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 104-109, June 1998.

[6] G. Cabri, L. Leonardi and F. Zambonelli. Reactive Tuple Spaces for Mobile Agents Coordination. In *Proc. of the 2. International Workshop on Mobile Agents*, LNCS Vol. 1477, pages 237-248, Stuttgart, Germany, September 1998.

[7] L. Cardelli and D. Gordon. Mobile Ambients. In *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, LNCS Vol. 1378, pages 140-155, Berlin, Germany, March, 1998.

[8] O. Carvalho and G. Roucariol. On Mutual Exclusion in Computer Networks. *Communications of the ACM*, 26(2):146-147, February, 1983.

[9] A. Castillo, M. Kawaguchi, N. Paciorek and D. Wong. Concordia as Enabling Technology for Cooperative Information Gathering. *Proc. of Japanese Society for Artificial Intelligence Conference*, June 1998.

[10] P. Felber. The CORBA Object Group Service: A Service Approach to Object Groups in CORBA. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998.

[11] Michael N. Huhns. Agent Teams: Building and Implementing Software. *IEEE Internet Computing*, 4(1):93-95, February 2000.

[12] Isis Distributed Systems Inc. and IONA Technologies Ltd. Orbix+Isis Programmer's Guide, Document D070-00, 1995.

[13] Dag Johansen, Keith Marzullo, Fred B. Schneider, Kjetil Jacobsen and Dmitrii Zagorodnov. NAP: Practical Fault-Tolerance for Itinerant Computations. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, May 1999.

[14] Dag Johansen, Fred B. Schneider and Robbert van Renesse. What TACOMA Taught Us. Mobile Agents and Process Migration - An edited Collection, Addison Wesley, 1998.

[15] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla and G. Cybenko. Agent TCL: Targeting the Needs of Mobile Computers. *IEEE Internet Computing*, 1(4):58-66, July 1997.

[16] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558-565, July 1978.

[17] D. B. Lange and M. Oshima. Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley, 1998.

[18] Silvano Maffeis. Run-Time Support for Object-Oriented Distributed Programming. PhD thesis, University of Zurich, 1995.

[19] Vera Nagamuta. Coordenação de Agentes Móveis através do Canal de Broadcast. Master's thesis, IME, University of São Paulo, R. do Matão 1010, 05508-900 São Paulo, Brazil, November 1999.

[20] H. Peine and T. Stolpmann. The Architecture of ARA Plataform for Mobile Agents. *Proceedings of the First International Workshop on Mobile Agents*, LNCS 1219, Berlin(D), pages 50-61, April 1997.

[21] R. V. Renesse, K. Birman and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76-83, April 1996.

[22] G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 24(1):9-17, January 1981.

[23] K. Rothermel and R. Popescu-Zeletin. Mobile Agents. *First International Workshop on Mobile Agents (MA '97*LNCS), Vol. 1219, Springer-Verlag, Berlin Germany, 1997.

[24] G. Susilo. Infrastructure for Advanced Network Management based on Mobile Code. Technical Report SCE-97-10, Systems and Computer Engineering, Carleton University, 1997.

[25] David Wong, Noemi Paciorek, Tom Walsh, Joe DiCelie, Mike Young and Bill Peet. Concordia: An Infrastructure for Collaborating Mobile Agents. In *Proc. First International Workshop on Mobile Agents 97 (MA'97)*, Berlin, LNCS 1219, pages 86-97, Springer-Verlag: Heidelberg, Germany, April 1997.