Reliable Communication in the Presence of Agent Mobility

Hervé Paulino
Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
herve@di.fct.unl.pt

Abstract

Although mainly resorting to local communication, all mobile agent systems also provide some kind of remote communication facility. This introduces issues not addressed in the protocols provided by the transport layer, such as TCP, nor in its extensions to cope with physical mobility. The problem consists of tracking the agent to which the channel must be established, and providing support for the migration of any of the channel’s endpoints. In this paper, we propose a model that solves both problems, transparently to the application, by resorting to sophisticated interaction with a naming service and by featuring a protocol to support the migration of both the connection’s endpoints.

1 Introduction and Motivation

In recent years, the mobile agent paradigm has proved to be a useful tool in the implementation of widely distributed applications. Code or process mobility has become a common feature in many of the Internet development languages and platforms. Nonetheless, mobility is a relatively young approach to distributed computing, and many of the issues it arises are still under study. In this paper, we address one of these issues: the problem of establishing a reliable and transparent communication channel between two mobile agents. The problem can be decomposed in finding the agent, and supporting the migration of the endpoints.

We know that mobile agents are autonomous software programs that have the ability to migrate from one host to another in a network. By definition, they rely on local communication, migrating towards the location of the resources, services or peers to which they want to interact with. The goal is to avoid the maintenance of costly remote communication sessions and provide disconnected execution, decoupling the application from the physical device from where it was launched. However, and somehow a paradox, remote communication is also regularly used in mobile agent based applications. Paradigmatic examples of such use are mostly related to cooperative computing, where the agents involved may want to synchronize, to perform life checking operations or even to communicate results from the ongoing work. The amount of information involved is usually far smaller than the agent’s state. Thus, current mobile agent platforms prefer to compromise between efficiency and design, avoiding migrating the whole agent when it is not efficient.

The classic remote communication protocols, such as TCP, provided by the transport layer do not support mobility. They build on the fact that the process that wants to establish the connection link knows the whereabouts of its counterpart, and that the later is statically bound to that location. The introduction of mobility makes these assumptions impracticable, since processes may migrate from host to host. A communication channel must now be defined between two processes (agents), rather than between two network locations. Furthermore, the usual connection-oriented approach provided by the transport layer only ensures the integrity and the ordering of the data exchanged. From the moment that one of the endpoints is missing the connection is broken. Connections between mobile agents must be able to recover from the migration of one of the endpoints, by re-establishing the communication channel with the migrating agent at its new location. The challenge is to guarantee that every message is delivered at least once, without restricting the agents mobility nor assuming permanent connectivity. The ideal would be to have the message delivered exactly once.

Work has been done in the domain of physical mobility. Protocols such as Mobile IP [3], TCP-R [2] and M-TCP [9] allow for network links between mobile resources. However these are not targeted to handle logic mobility, where is the application that migrates, rather than the device. In the field of logic mobility, the increase of platforms that feature some kind of mobility also have originated some proposals. Some simply address the issue of migrating a connection [11], other also address the problem of tracking the agent’s location, by resorting to distributed snapshots [5], mailboxes [1] or groups [8]. Our approach was developed in the scope of the MOB mobile agent platform [7] and addresses both problems. It provides means to track an
agent in the network, and to migrate a communication channel between two agents. It contributes in both areas by:

- providing a new approach to connection establishing, by incorporating a contact notification mechanism in the naming service;
- allowing overlapped migration over a communication channel suspend/resume protocol.

The paper is structured as follows: the next section presents a more in-depth discussion of the field’s state of the art, section 3 presents our approach to the problem, section 4 provides an implementation of our model in Java, and finally, we present some conclusions in section 5.

2 State of the Art

Murphy and Picco present a study in [5] that assesses the limitations of the classic approaches, such as the spanning tree broadcasting protocol, and the forwarding technique with a home agent, used to enable physical mobility [3]. The former, that broadcasts a copy of the message to each neighbour of the current node, fails when the agent is traveling in the message’s opposite direction. The later cannot be efficiently applied to this context. It does not ensure the delivery to highly mobile agents, and it introduces a centralization point that bottlenecks all incoming communication.

In consequence they proposed an algorithm, based on the Chandy-Lamport distributed global snapshot, that sends a copy of the message to each neighbour of the current network node, attempting to find its destination. A node that receives a message, checks if the target agent is running within its boundaries. If not, it sends the message to every outgoing channel it holds that has not received the message yet. This model does not require any global naming service, guarantees the delivery of the message and enables multicast. However, the overhead is of one message per channel traversed, which generates a large amount of traffic, due to the flooding of the network. Although several copies of the message can reach the site hosting the agent, it is ensured that only one copy of the message is delivered.

Zhang and Dao [10] proposed an algorithm for persistent connections between logical endpoints. These endpoints are mapped into their physical locations by a centralized naming service with an updating policy. The naming service propagates any change in the network’s topology to all the agents registered. The great amount of overhead and network traffic provoked by the need of propagating every change in the topology to all the agents is a performance killer on systems with highly mobile agents.

Mishra and Xie [4] proposed models for both synchronous and asynchronous communication. The first relies on a centralized clearinghouse protocol. Each agent publishes to the network a host as its clearinghouse. This host knows the whereabouts of its agent, and acts as a proxy that forwards the messages, placed by the sender (using a put operation) to the agent (when it invokes a get operation). Asynchronous communication uses a centralized mailbox, where the messages targeted at an agent are placed. Sending a message to an agent is to place it in its mailbox.

Cao et al [1] proposed an algorithm that also associates a mailbox to each agent. The mailbox is decoupled from the agent, allowing the later to migrate without its mailbox. This mechanism only works if the mailbox is always reachable from the agent. This may oblige the mailbox to migrate along the agent’s path, which requires it to have a road-map of the agent’s migration path. The whole algorithm relies of the definition of three aspects: mailbox-to-agent message delivery, mailbox migration, and the synchronization between the message-forwarding and the agent-migration. This mechanism is targeted at asynchronous communication, and not well-suited for synchronization, which is welcomed in some cooperative applications. The migration of the mailbox, although less frequent that the agent, requires a mailbox tracking mechanism not that different from the ones used to track agents.

Silva and Macêdo [8] proposed an algorithm based on mobile groups. Each agent in the group has a view of where its peers are located. Every agent migration or loss of connectivity is propagated to all the remainder members of the group, so that they have an updated view of the group. The algorithm guarantees that every message is delivered, but once again the need of propagating the change in the group’s topology to all its members is not efficiently implementable.

Zhong and Xu [11] do not focus on tracking the agents. Their work provides a middleware layer (NapletSocket) placed between the application and the Java sockets API, that provides reliable communication on a established TCP connection. They bring concepts used to keep TCP connections alive during physical mobility, used, for instance in TCP-R [2] and M-TCP [9], to the field of logical mobility. They add two new operations over the socket: suspend and resume. Before migrating, an agent suspends all the connections it holds resuming them once it reaches its new destination.

3 Our Approach

Our model addresses both the problems of tracking the agents, and of their migration during communication. It provides transparent connection to a mobile agent, and transparent migration of one or both of the channel’s endpoints. We track the agents by resorting to an agent naming service (ANS) that holds the updated locations of the agents
running in the network. Unlikely Zhang and Dao, with do not propagate the changes made in the network’s topology. We choose to have a lazy reconfiguration, where a connection between an agent and the ANS is only established when the resolution of a location is necessary for communication to happen.

The transparent handling of the migration of the endpoints is supported by wrapping the application’s communication channel, and using it to send control messages along with the application’s communication. The protocol is based on the same concepts used by the NapletSocket, but it differs in the sense that it does not impose any restrictions on the communication protocol. The NapletSocket is a middleware between the application’s and the TCP protocol, while our approach works on top of the protocol use by the application, be it synchronous or asynchronous, connection-oriented or not, transport or application layer, as long as it respects a given interface. Furthermore, our approach allows for overlapped migration, which is not supported by the NapletSocket, where only one of the agents involved in the communication can migrate at a time. This introduces the overhead of resuming a connection only to be suspended again. In this section, we discuss the protocol in general. For a more in-depth description we refer the reader to [6].

The target of the model is the large number of systems and languages that offer code or process mobility. Thus, we do not make any assumptions that are not current features of these systems. The only restriction we impose is the existence of a unique identifier for each agent which is used to register the agent’s location in some sort agent naming service. We do not make assumptions on the architecture of the naming service. It might be centralized, distributed, with a flat or hierarchically name structure (although we suggest and aim for a distributed implementation). We do, however, incorporate some specific features regarding the operations it supplies and the information it stores.

### 3.1 The Agent Naming Service

The main roll of the ANS is to hold and resolve the current location of each agent in the network. For that purpose it supplies two operations: **REG**, registers the agent’s identifier and location; and **RESV**, resolves an agent’s location given its identifier. To ensure that the information kept by the ANS is always reliable and updated, a bond is made between the agent and the node hosting the ANS instance (or simply the ANS node) that holds its registry. From this point, every migration is reported to this node. Before migrating, the agent sends a **MIG** message to the node, and as soon as it reaches its new location, it sends a **REG** that overwrites the previous location.

While an agent is on the move, its location is not known to the ANS, and thus the later cannot successfully reply to resolution requests regarding that agent. In this case, the request is answered with the **ERROR<NotKnown>** message, which the querying agent handles as it wishes. A common approach would be to query the ANS again and again, until the new location of the agent is known. This floods the ANS system with queries that will probably lead to the same **ERROR<NotKnown>** result. Distributed implementations of the system will suffer a large overhead, due to the need to exchange messages. Another problem is that, once the location is known to the ANS and the queries are successfully answered, there is no guarantee that the agent to track has not migrated again, thus causing the whole tracking process to start all over again.

To cope with these two problems we introduced a different resolution operation (**RESVW**), and added a **to-contact** list to the registry of an agent in the ANS. When the agent to track is on the move, **RESVW** places on its ANS registry the identifier of the agent trying to track its location. Once the moving agent sends its **REG** message to update its location, it receives this list, and becomes aware of the agents trying to contact it. This requires that the information stored in the ANS includes, besides the agent’s location, a flag indicating if the agent is moving, and the list of the agents that are waiting for the current agent to contact them (note that by location we mean the representation given by the system, no restriction is imposed).

As we mention before, distributed implementations of the ANS introduce the overhead of internal message exchanging. To reduce this overhead, each agent features a cache to store the ANS nodes associated to the agents it has contacted before. This improves the overall performance of the system, since it avoids the need for the ANS to discover which is the node holding the registry to query. We provide this facility by piggybacking on demand the location of the ANS node when it replies the location of a given agent. This feature is available on both the **RESV** and **RESVW** operations, by suffixing them with **ANS** (e.g. **RESVANS**).

To terminate the discussion on the ANS, and demonstrate one of the advantages of caching the ANS nodes, we introduce the **FWD** operation. It allows an agent to contact any ANS node to perform an operation and indicate which is the node that holds the registry where the operation must be performed, i.e., to forward an operation to a given ANS node. This is particularly useful, when the agent lands on a network that does not have a route to the ANS node on which it wants to perform the operation, being it the registering of its new location, or the tracking of some other agent’s location.

Table 1 presents the ANS interaction protocol. Each type of request its followed by the possible replies, whether the operation completed successfully or generated an error. Successful replies are labelled with **OK**. Error replies,
such as the inability to perform the registry for some reason, are labelled with ERROR, and contain the reason of the error. The contents of a message are delimited by <>. The DEL message causes the registry for the agent to be deleted. agentId stands for the identifier of the agent to query, localAgentId for the identifier of the agent performing the query, agentLoc for the location of agent, and ansLoc for the location of the ANS node associated to the resolved agent.

### 3.2 The Communication Channel

We now present the protocols to establish a communication channel between two agents, and to provide transparent migration of one or both of the channel’s endpoints.

### Implementation of the ANS

Establishing a connection from b to a previously acquainted agent a (figure 3) begins by (1) checking if agent a is still running at its last known location. When it is not, the ANS node associated to a is fetched from the cache and an attempt to contact it is done (2). If there is no route to the given ANS node, the local node is contacted and the query forwarded (3 and 4). After this step the process reduces to the ones regarding a first contact.

### Figure 2. Notify the target agent of the communication request.

Once the connection is up, communication may occur until one of the agents migrates to a new location. When...
this happens, we need a protocol to ensure the reliability of the channel after the migration.

Migration of an Endpoint: the protocol we propose is an extension of the one proposed by Zhong and Xu in [11]. It extends the interface of the communication channel with the suspend and resume operations. suspend must be called if one of the endpoints of the channel wants to migrate (b). It sends a SUSP control message to agent on the other end of the channel (a), informing it that the channel is temporarily unavailable, and that it will be resumed as soon as the agent lands on its new location. The resume operation establishes a new communication channel with the agent waiting for the channel to be resumed, and sends it a RESU message (figure 4).

![Figure 4. Migration of an endpoint.](image)

Migration of an Endpoint: when agent (b) tries to re-establish a communication channel with agent (a), it may happen that the later is in transit or actually moved to a distinct host. In this case, agent (b) must contact the ANS to first locate agent (a) (figure 2) before re-establishing the channel (figure 5). For the sake of simplicity we removed the actual protocol messages from the figure.

As we described above, if the query to the ANS to obtain the new location of (a) fails (results in ERROR<NotFound>) then, the identifier for agent (b) is placed in a table holding agents waiting for agent (a) to register again with the ANS. Agent (a) will in turn know the new location of (b) when it registers its new location in the ANS and processes the table of pending agents.

4 Implementation

This model was developed in the scope of the MOB platform, and is presented as the org.rvs.mob.recom Java package. MOB is service-oriented scripting language for programming mobile agents that uses TCP-based communication. The system resorts to an ANS to store the locations of the agents.

The recom package features two interfaces (listing 1): Connector is the interface that must be implemented by the application’s communication protocol, and; Identifier represents an network identifier. It is the key for the registries kept by the ANS.

Listing 1. Connector and Identifier interfaces

```java
import java.io.Serializable;

public interface Connector extends Serializable {
    public void send(Message msg) throws IOException;
    public Message recv() throws IOException;
    public Location getLocalEndPoint();
    public Location getRemoteEndPoint();
    public void enableInput() throws IOException;
    public void enableOutput() throws IOException;
    public void close() throws IOException;
}

public interface Identifier extends Serializable {
}
```

The Location abstract class is the representation of a communication endpoint that accepts incoming connections. It requires the implementation of the protocol specific accept and connect operations.

Listing 2. The Location abstract class

```java
import java.io.Serializable;

public abstract class Location implements Serializable {
    private Identifier identifier;

    public Location(Identifier identifier) {
        this.identifier = identifier;
    }

    public Identifier getId() { return this.identifier; }
    public abstract Connector accept() throws IOException;
    public abstract Connector connect(Location loc)
        throws ConnectionFailedException;
}
```

The package also features four classes that implement the actual model. They are abstracted in the connector provided for the communication. Class Channel wraps a connector to provide synchronous communication between agents. AsynChannel performs the same action but to provide asynchronous communication. Its implementation features a thread that continuously listens the channel for incoming messages, that, on arrival, are placed on a local queue. Both these classes feature methods to connect to an agent, send and receive messages, and to close the channel. The messages passed along a connector are instances
of the **Message** class labelled by the **MessageLabel** enumeration.

The **AnsBinder** class provides the means for an agent to contact an ANS node. It features methods that use the connector to send the queries to the given node. The **AnsNode** class abstracts a node of the ANS, requiring the concrete definition of the connector used to communicate with the remainder of the network, and of the technology used to store the data, e.g., Java data structures or an external database.

## 5 Conclusions

Our proposal has the ability to provide complete transparent communication between (possibly) mobile processes identified logically by some unique identifier. It bases its tracking capabilities on an ANS, that by design is aimed to be distributed, although a centralized alternative works as well. The distribution avoids the computational and network bottleneck, but introduces the overhead to solve the queries within the system. For that purpose, we minimize the impact of the ANS on the overall performance by creating a bond between an agent and the node of the ANS that holds its registry. This node is cached by every agent that has previously contacted the agent, allowing the bypassing of the ANS resolution mechanism, and perform the query directly on the source. We also reduce the amount of queries performed by an agent when it is trying to solve the location of an agent in-between locations. We add a `to-contact` list that stores the agents trying to contact the agent, thus replacing the query/reply mechanism by an notification driven one. We can also see that the model is targeted at a distributed ANS in its ability to forward requests without imposing any protocol for the discovery of the local ANS node.

This tracking mechanism although not perfect, solves the bottleneck and scalability problems of a centralized approach, it does not require any kind of forwarding, neither it floods the network to deliver the messages.

The communication channel migration protocol allows for transparent migration of both endpoints, with the possible help of the ANS. Furthermore, it does not require any extra channels, only an initial handshaking protocol, and a byte to tag each message. Its implementation is easy to use, since it only requires the definition of the communication protocol, the connector.

## References


