The best agents, would not only need to exercise a particular form of expertise, but also take into account the peculiarities of the user and situation. In this sense an agent fills the role of what Negroponte calls a “digital sister-in-law.”

“When I want to go out to the movies, rather than read reviews, I ask my sister-in-law.

We all have an equivalent who is both an expert on movies and an expert on us. What we need to build is a digital sister-in-law.

In fact, the concept of “agent” embodied in humans helping humans is often one where expertise is indeed mixed with knowledge of you. A good travel agent blends knowledge about hotels and restaurants with knowledge about you... A real estate agent builds a model of you from a succession of houses that fit your taste with varying degrees of success. Now imagine a telephone-answering agent, a news agent, or an electronic-mail-managing agent. What they all have in common is the ability to model you.” [78].

A preliminary proposal based on the FIPA mobility specifications has been implemented. In this chapter, the structure of this mobility service, its integration with the JADE is described. The implementation of DDS is described in two recent papers[45][48].
5.1 MOBILITY SERVICE STRUCTURE

DDS implementation is built on top of the JADE agent platform[48]. A container is a non standard abstraction defined in JADE to support distributed computing and consisting in an agent execution environment. Every container usually runs on an independent Java virtual machine, and it communicates via RMI with the rest of containers belonging to the same platform.

Since JADE is chosen for implementation, DDS is made in accordance with its services architecture[38]. This architecture, thanks to its flexible structure based on vertical and horizontal commands, makes it possible to add new services to the platform without modifying it. Moreover, it has the benefit of allowing interaction and collaboration between other services. This architecture is very flexible, but it is also quite complex, because it has to manage all platform services and to coordinate their instances over all platform main container.

The DDS is a complex piece of software. In order to simplify the service organization and its tasks it has been structured into these parts: main service, code analyser, class analyser and packer, code warehouse, and migration protocol. As we can see in Fig.5.1 each one of these service parts has an instance in each container, except for the migration protocol. This protocol only resides in the platform main container, which is the only one visible according to FIPA specifications.

Fig. 5.1: DDS service parts

The main component offers a JADE service image of the migration
software. It deals with the coordination of all service parts over one or more containers.

The class analyser, implemented as a standalone library called Class Analysis Library (CAL), is a key component to ease the agent dependant classes collection. As JADE does not have class spaces differentiated for agents, a search of the dependant classes has to be done in order to decide which classes must be packed to be sent to the destination platform. The CAL recursively searches dependencies for a given class by including them on the agent’s code package.

To analyze each class the code analyser component is used. It loads the constant pool\texttt{[96]} table of a Java class in memory, being able to easily access to it. This table shows the relationships between the analysed class and the dependant ones.

The CAL, by using the code analyser, first gets the direct dependencies of the agent class, and then it searches for the dependencies of these direct dependencies, and so on. It is easy to see the enormous amount of classes that can finally be found. This is a problem, because most of the classes will be in the mobile agent target platform, and it is not needed to transfer so much data. In order to fit the classes transferred to the real needed, a filter is made based on the class package names to exclude some of them from the analysis. Finally, with the dependency list, all needed classes are packed into a JAR file (known as a JAR agent) ready to be got and sent by the migration protocol component to the destination platform.

Another important component of the service is the code warehouse. It is implemented by a class called CodeLocator that is placed inside the JADE Agent Management Service. This component is used to maintain a binding between all mobile agents of the platform and their code (the JAR files). Has to be noted that the first time a mobile agent is executed, previously to migrate, it is not generally packed in a JAR file and, because of that, it is not registered in the code warehouse. It is from the first migration that, by using the code analyser component, the JAR file is made and registered.

Furthermore, a method preventing from duplicated JARs, in case of the existence of many equal code agents, has been implemented. A unique identifier is included in each one according to the code inside them. Together
with the code warehouse, a specific ClassLoader to dynamically load classes from the JAR files has been developed. It allows to have a separated class space for every JAR agent avoiding class conflicts.

And the last component, the migration protocol, is in charge of running the dialogue between platforms involved in an agent migration. It defines the steps needed for a successful migration and the data structures used in them.

To exchange messages the interaction protocols proposed by FIPA have been used, which contribute to standardize communications. More precisely, the chosen one has been the FIPA Request Interaction Protocol[35], that allows to make a request to an agent and to be answered accordingly to the result (with an inform or a failure message).

Basically two steps have been defined, each one implemented with one instance of the FIPA Request Interaction Protocol[35] with an action associated. The first step sends a "move" action with the agent code and instance and waits for the success. Then, the second step only sends a "power-up" action to start the agent previously sent (the messages exchanged can be seen in Fig.4.3).

1. A "move" request message with the agent code and data is sent.
2. An acknowledge of the agent creation is received from the target platform.
3. The receiver is requested with a "power-up" message to start the agent sent.
4. An acknowledge of the agent startup is received from the target platform.

A migration ontology similar to the one proposed by FIPA[34] has been used for data structure(for more details, refer chapter-4). The ontology used includes code and data of the agent and, moreover, information about the agent platform, the language of the agent, and the migration protocol used(as discussed in section 4.4.3). This prevents the platform to execute an agent incompatible with some of the mentioned characteristics.

5.2 INTEGRATION WITH JADE MOBILITY SERVICE

The mobility service currently presented has been integrated with the JADE, allowing an agent to migrate between containers and platforms by
using the same methods.

The destination of an agent migration is indicated by using an object implementing the Location interface. With the intra-platform migration service the ContainerID class, implementing the Location interface, is used as agent container destination representation. Then, with the inter-platform migration service a PlatformID class has been implemented to represent the destination platform. That's the fact that allows the user to use the same methods to start a migration. When an agent needs to migrate, it just has to call the doMove method with a ContainerID or a PlatformID depending on the desired kind of migration (inside or outside of the agent platform).

Moreover in current service special attention has to be paid to the agent's code location among the platform's containers. When an agent decides to migrate to another platform its code could be in another container because an older intra-platform migration of the agent. On this kind of migration, the agent's code is not pushed to the destination container. Instead of that, the code is requested on demand, remaining the original code all the time in the same container. For this reason a service horizontal command has been implemented to ask the source agent container to generate a JAR with the agent code.

5.3 LAUNCHING DDS

The DDS is not built into the platform and must be installed as an add-on. To install the add-on, the package distribution file must be unzipped inside the JADE folder from where the ant lib command is used to create a JAR file containing all compiled class files. To use the service, it must be explicitly specified on the command line as follows

```
java  jade.Boot -services
  jade.core.mobility.AgentMobilityService;jade.core.migration.DDS
```

5.4 PROGRAMMING A MOBILE AGENT

A primary goal of the mobility services is to be as simple to use as possible. In this vein, it is necessary to overload some methods and call the doMove() or doClone() methods of the Agent class to move or clone the agent (note that doClone() is only available with the Intra-Platform Mobility Service).
To indicate the destination of a mobile or cloned agent, JADE defines the jade.core.Location interface. Two implementations of this interface are provided: jade.core.ContainerID for intra-platform migration and jade.core.PlatformID for inter-platform migration. The ContainerID must be initialized with the name of the destination container and its transport address. The PlatformID must be initialized with the AID of the remote platform’s AMS agent including its transport address (Objective-3).

To move an agent to another container or platform, an agent behaviour must include a call to the method doMove(). This method changes the agent to the TRANSIT state indicating that it is about to migrate. The following code illustrates an intra-platform migration:

```java
// Create some variables
String containerName = "Container-1";
ContainerID destination = new ContainerID();

//Initialize the destination object
destination.setName(containerName);

//Change of the agent state to move
myAgent.doMove(destination);
```

In the case of inter-platform migration, a similar example is as follows:

```java
// Build the AID of the corresponding remote platform's AMS
AID remoteAMS = new AID("ams@remotePlatform:1099/JADE", AID.ISGUID);

// Specify the MTP by setting the transport address of the remote
// AMS
remoteAMS.addAddresses("http://remotePlatformaddr:7778/acc");
// Create the Location object
PlatformID destination = new PlatformID(remoteAMS);
```
To clone an agent the doClone() method is used to change the agent state to COPY. To use it, two arguments must be passed, the container destination and the new agent name. This is an example:

```java
// Create some variables
String containerName = "Container-1";
String newAgentName = "myClone";
ContainerID destination = new ContainerID();

//Initialize the destination object
destination.setName(containerName);

//Change of the agent state to clone
myAgent.doClone(destination, newAgentName);
```

As these mobility services implement weak migration, a code structure based on a finite state machine is required. This is because the program counter of the agent execution is not transmitted, making it is impossible to continue agent execution from the next line of the doMove() or doClone() instruction. Instead, agent execution can continue only from the beginning of the agent's behavioural code. Using a finite state machine representation, a structure can be created which segments the agent code into sections with assigned variables indicating the agent code execution state.

A switch statement is used, for example, an agent with both seller and buyer roles can have code with two states, one for selling and another for buying. The agent must set a state variable before leaving a container or a platform to indicate which role will be initially adopted in the next location. To create such a finite state machine representation. For example, in a two-container trip where the destination containers are in an array of locations, the behaviour code to be executed in each container is separated as in the following example:

```java
addBehaviour(new CyclicBehaviour(this) {
    public void action() {
```
switch(_state) {
  case 0 :
    // Agent starts to migrate
    _state++;
    myAgent.doMove(_dests[0]);
    break;
  case 1:
    // Agent migrates to the second container
    _state++;
    myAgent.doMove(_dests[1]);
    break;
  case 2 :
    // Agent dies
    myAgent.doDelete();
    break;
  default:
    myAgent.doDelete();
  }
}

private ContainerID[] _dests = ...;
private int _state = 0;)

This example shows how agent code must be structured in JADE to preserve its state by using a variable.

During the serialization[55] and deserialization process of migrating agent code, some resources used by the agent will also be transferred, while others will be disconnected before migration of the agent and reconnected at the destination (this is the same distinction between transient and non-transient fields used in the Java Serialization API). JADE provides two matching methods in the Agent class for resource management that need only be overloaded by the programmer:

- beforeMove() - called at the source location when the move operation has successfully completed such that the moved agent instance is about to be activated on the destination container and the original agent instance is
about to be terminated. This method is the correct place to release any local resource used by the original agent instance (e.g. closing open files and GUIs). If these resources were released before knowing if a migration was successful, they would just have to be reopened once again. However, a consequence of this is that any information that must be transported by the agent to the new location must be set before the doMove() method is called. Setting an agent attribute in the beforeMove() method will have no impact on the moved instance.

- afterMove() - called at the destination location as soon as the agent has arrived and its identity is in place, but before the behaviour scheduler has restarted.

For agent cloning, JADE provides a corresponding method pair, the beforeClone() and afterClone() methods. These are called in the same fashion as the beforeMove() and afterMove() methods. All four of these methods are protected members of the Agent class, defined as empty placeholders. User-defined mobile agents can override the four methods as needed.

5.5 SUMMARY

This chapter gives an overview of the DDS implementation issues. The DDS structure, its integration with JADE, launching and programming of DDS are discussed in detail.