# Agent Migration between Incompatible Agent Platforms

Pauli Misikangas and Kimmo Raatikainen
Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 UNIVERSITY OF HELSINKI, Finland
email: {pauli.misikangas,kimmo.raatikainen}@cs.helsinki.fi

## Abstract

*Several agent platforms of general purpose exist — for example, Voyager, Jade, and Grasshopper — each of which provide an environment for building and executing software agents. Unfortunately, the platforms are usually incompatible with each other. Thus, agents built for one platform cannot be used in another platform, nor can they interact with agents in other platforms. Some effort is put into standardizing agent communication and migration in FIPA and in OMG, but these standards are not yet supported by most of the existing platforms. Therefore, we should find some other ways to allow interaction between agents in different platforms.*

*In this paper we will show that it is possible to make platform independent agents that are able to migrate between incompatible platforms. We will also describe how messages can be delivered to agents in other platforms, and show how to build platform independent service agents that are used via method calls. The ideas have been tested in practice with Voyager, Jade, and Grasshopper platforms.*

## 1. Introduction

The software agent society has re-invented the old problem of several incompatible systems, which has annoyed programmers for decades in the forms of operating systems, programming languages, and so on. There are dozens of different agent platforms including Voyager [9], Jade [1], and Grasshopper [6], each of which has the same basic functions extended with some fancy features. Unfortunately, the general rule is that these platforms are incompatible with each other. Therefore, agents built for one platform cannot be used in another platform. And even worse, agents cannot communicate with other agents that reside in another platform. This means, for example, that a shopping agent built for platform $X$ cannot buy products sold by agents in other platforms.

Agent system developers are hopefully looking to FIPA [4] and OMG [8] which are trying to standardize agent communication and migration. If everything goes well, some day we will have nice standards supported by most of the agent platforms. However, it would be unrealistic to hope that all agent platform vendors will support those standards in the near future. In the worst case, some vendors might create their own extensions or competitive standards because "the existing standard was not adequate." If they have enough customers and applications to back up their opinions, the agent world will split and we are back with the original problem.

Thus, there are good reasons for trying to develop techniques that would narrow the gap between agent platforms. At least, we should be able to re-use the same agent source code in different platforms. This is analogous to writing portable programs that can be compiled under different operating systems. A commonly used technique for this is to separate all system specific code into modules that are used through a system independent *application programming interface* (API). The advantage of this approach is that only those API modules must be re-implemented for each supported operating system. This simple idea can easily be applied to agents as well. We can separate the platform specific part of an agent and create an interface for it. The rest of the agent is platform independent and can be used in other platforms without modifications.

This idea can be elaborated even further. In this paper we will show that it is possible to make platform independent agents that are able to migrate between incompatible platforms — for example, between Voyager and Jade. In addition, we will describe how to deliver messages between agents running in different platforms so that the message passing is transparent to the sender and the receiver. We will also outline the principles of building platform independent service agents so that they can be exploited by using normal method calls. Finally, we will present a simple example agent that has been used to test agent migration between Voyager, Jade, and Grasshopper platforms.

## 2. Background

Every project that is planning to take advantage of agents must choose whether to use an existing agent platform or to build one of its own. The choice is not easy because there are several platform candidates and none of them outdoes others in all respects. Especially, if one is developing a commercial product, such as shopping agents for electronic commerce, one should try to select the platform which will have the most users in the future. Whichever is selected, it might turn out to be the wrong decision in the long run.

On the other hand, building a new platform is even more risky since users who are already using some agent system will probably not be very anxious to move to a new system. However, many research projects that are not aiming to build a commercial product might find the development of a new agent platform a fascinating solution.

The dilemma of using an existing agent platform versus building a new one was also faced in the Monads project[1] [3]. Our goal is to advance working in wireless environments, i.e. a laptop computer or an advanced PDA connected to a fixed network via wireless networks, such as GSM Data [7], GPRS [2], wireless LAN [10], by using intelligent and adaptive agents. Since none of the existing agent platforms is designed to be used with wireless connections, we were tempted to build an agent system of our own. Another possibility was to modify an existing platform so that it suits our purposes better. In both cases, the result would have been a system that is incompatible with existing ones.

Our decision was to build the Monads system on top of an existing agent platform that uses Java as the basic agent implementation language. However, we were not ready to commit ourselves to a single platform. Instead, we designed a system that can easily be implemented for several platforms so that user-level Monads applications will work on each of them. Furthermore, Monads agents are able to interact with native agents, such as pure Voyager agents, to operate in agent servers without Monads support, as well as to migrate and to send messages to another platform via the *Monads Agent Gateway*, which is described in Section 4.

Figure 1 outlines the basic conceptual architecture of the Monads System. All Monads agents are built upon the Monads API, which defines the operations and services needed by the agents. The operations and services are implemented using the corresponding functionality of the underlying platform whenever possible. If some operations are not supported by the underlying platform or they should be optimized for a wireless connection, we use our own implementation instead. Since the underlying agent platform is not modified in any way, native agents can be used together with the Monads agents.

## 3. Separating agent's head and body

Every agent platform has a unique set of services and interfaces of its own. Many platforms have a special agent superclass from which all agents must be derived, for example. This means that each platform has a unique *agent programming interface* that includes everything needed in programming agents and agent-server intercommunication (see Figure 2). Such an interface defines, among other things, how agents send and receive messages, how agents register and look up services, and how agents move between agent servers, and so forth.

[1] The partners of the Monads project are Nokia Mobile Phones, Nokia Research Center, Sonera Ltd., and the University of Helsinki. The project is also funded by the National Technology Agency of Finland (TEKES). The project homepage is at `http:/www.cs.helsinki.fi/research/monads`

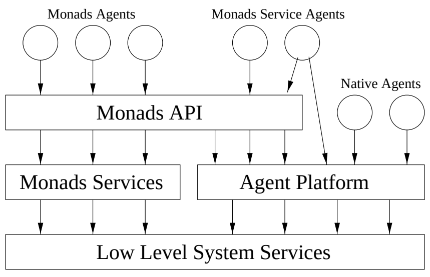

**Figure 1. Conceptual architecture of the Monads system**

Unfortunately, agent programming interfaces of different platforms are usually incompatible with each other. This is the greatest obstacle for agent migration between platforms as depicted in Figure 3. If we could transfer the state and classes of an agent to another platform and wake up the agent, even then the agent could not do anything useful in this foreign environment because it does not know how to use local services. Actually we would not even reach that point because different types of agent servers do not have common protocols for sending or receiving agents, classes, and messages.

In Monads we have solved the problems mentioned above by separating the platform independent part of an agent (the *'head'*) from the platform specific part (the *'body'*. The agent body and head communicate with each other by calling abstract methods in the classes of `AgentHead` and `AgentBody` as shown in Figure 4. The `AgentBody` provides some basic operations needed by agents, such as *send message*, *move agent*, and *find service*. The `AgentHead` defines some methods such as *receive message* that are needed when an agent server or some other agent wants to communicate with the head. The main program of an agent is in the head (class `MyAgent`), which also implements the methods in the `AgentHead`, defining what the agent will do when a message arrives, and so on. The body (class `XBody`) extends the platform specific agent superclass `XAgent`, if the platform has one, and implements the operations defined in the `AgentBody`. Whenever the head wants to operate with an agent server—send a message, for example—the head calls a method available in the `AgentBody`. The body then performs the operations necessary to fulfill the request in the particular platform. Analogously, the body handles method calls

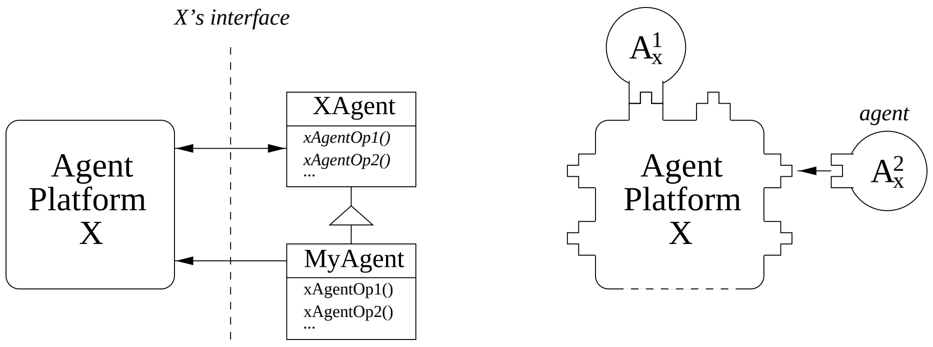

**Figure 2. Interface between agent and platform**

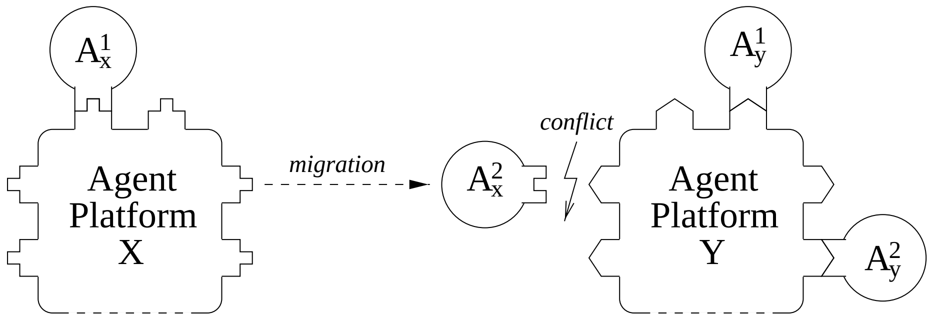

**Figure 3. Unsuccessful agent migration because of incompatible interfaces**

made to the XAgent by calling corresponding methods in the `AgentHead`.[2]

The body part of an agent must be implemented for every platform we want to support. However, it is the same for every agent, so the task needs to be done only once. The body should be kept as small as possible, containing only those operations that are absolutely necessary to survive in native servers without any Monads support. Thus, the `AgentBody` interface should contain at least the following operations:

**Move:** Move to another agent server by using the agent transfer services of the underlying platform.

**Send Message:** Send a text message—for example, a FIPA ACL message—by using native communication services. The body should also be able to receive such messages.

**Find Service:** Find a service by name and return the identifier of the service agent. This is done by using native yellow pages service.

[2]The class structure in Figure 4 can be seen as an instance of the *Adapter* design pattern [5]. In fact, the `XBody` is a *two-way adapter* since it adapts interfaces to both directions — from head to platform and vice versa.

**Get Native Proxy:** Get an object reference to an agent or its proxy. The identifier of an agent is given as a parameter and the type of the return value is `Object`, that is the superclass of all Java classes. The head must know the actual interface class of the agent and must cast the reference to the interface class. After this the head can use normal Java method calls to communicate with the agent.

Typically, when a Monads agent arrives at an agent server, it first searches for some Monads services—the Monads Naming Service, for example—and starts using them if they are available. However, the agent can also operate in non-Monads agent servers by using the operations listed above. Although the set of operations is very limited, it is sufficient for many useful tasks.

## 4. Advantages of head-body partition

In the previous section we described the basic idea of dividing agents into platform specific and platform independent parts—into the body and the head. In this section we show the advantages we can gain from this partition.

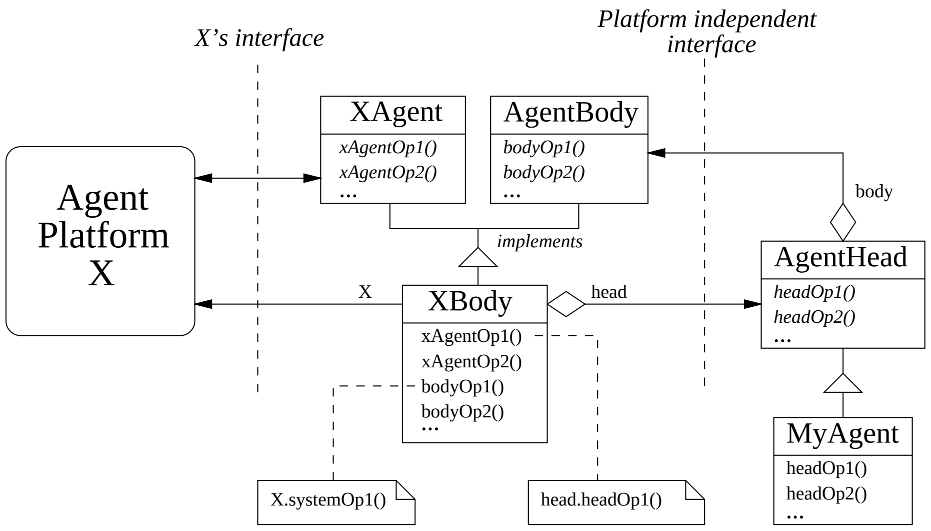

**Figure 4. Separating the platform independent part of an agent**

## 4.1 Head migration

The head-body partition allows us to use the source code of the head in several platforms. In fact, the compiled version of the head is also portable. This reflects the principal idea of this paper: Maybe we could move an instance of the head class to another platform at run time? If we use Java as the agent implementation language, the transfer is fairly easy since Java offers tools of object serialization and deserialization. In other words, we can transform an object into byte stream and vice versa. Thus, we can serialize a head, send it to another platform, deserialize it, and connect it to a new body. After these steps the agent is again ready to run but now in a different platform. However, we need a service that performs all the steps needed for the agent migration.

The *Monads Agent Gateway* (MAG) provides connections between agent servers of different platforms as depicted in Figure 5. The MAG is actually just a service agent that opens a socket connection to another MAG when needed. When an agent wants to migrate to another platform, the MAG splits the agent into the head and the body but only the head is transferred to the destination platform. At the receiving end, the local MAG creates a new body for the received head and joins them together; see Figure 6.

The body part stays alive in the original platform and waits until the head comes back and is re-connected to the body. It is important that the head is connected to the same body as before because the identity of a head-body agent is associated to the body. Thus, it would be impossible to reach an agent after destroying its body because the identifier of the agent becomes invalid by doing that. In order to find the right body for a head, identifiers of heads' bodies

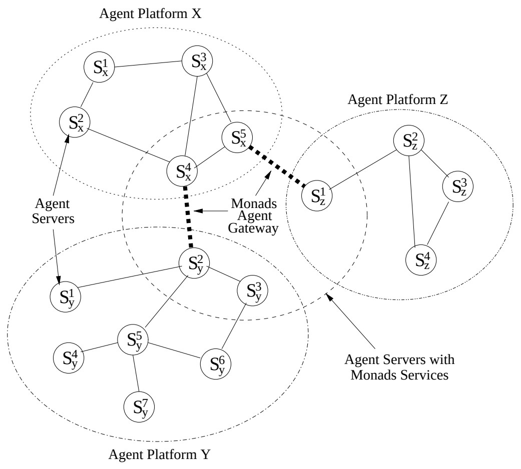

**Figure 5. Agent platforms connected with Monads Agent Gateways**

in different platforms must be stored into the head. When a MAG receives a head, it can check whether the head already has a body in the platform or a new body should be created.

## 4.2 Message delivery between platforms

A commonly used method of agent communication is to send text messages between agents. If compared to method calls, text messages have the advantage of being independent of the platform, programming language, class interfaces, and location of communicating agents. It is enough that the sender and the receiver agree on the meaning of messages, and there are some ways to deliver text messages between

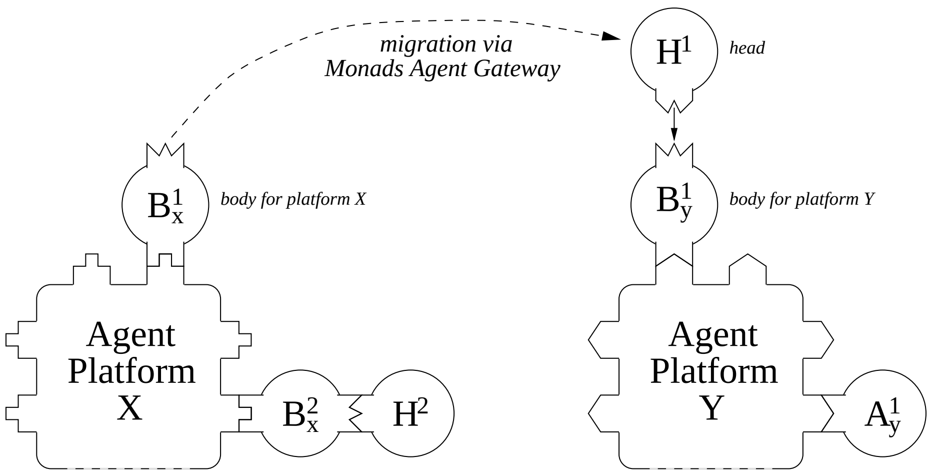

**Figure 6. Moving the head of an agent to another agent platform**

the source and the destination.

FIPA ACL [4] is a promising candidate for a common agent communication language. FIPA is also giving standards on how messages are delivered between platforms. Thus, when agent platform vendors start to use these standards in their products, we have a unique agent communication system. However, this is not going to happen in the near foreseeable future. Of course, we can already use FIPA ACL in communication between our agents but the platforms are not able to deliver messages in that form to agents in another platform. However, FIPA ACL messaging is supported by the Monads Agent Gateway.

Many platforms provide operations for sending messages. In the platforms that do not support messages, we can define a common agent interface that has the operation `receiveMessage(String message)`. Thus, agents can send messages to each other by calling this method. When somebody sends a message to a head-body agent, the message goes to the body first. If the head is present, the body delivers the message to the head. If the head has gone to another platform, the body has two choices: It can either store the message until the head comes back or ask the MAG to deliver the message to the head. In the latter case, MAG on the receiving side will forward the message to its destination by using the normal communication services of the local platform.

This kind of message delivering, which is transparent to the sender, i.e. the sender does not notice that the receiver is in another platform, is possible only when the receiving agent has a body in the sender's platform. However, agents can also send messages to other agents by using the MAG directly if they know the identifier of the receiver in the target platform.

### 4.3 Using platform independent services via method calls

The use of *Remote Method Invocation* (RMI) in agent communication simplifies the programming of agents. Instead of sending messages, we can use normal method calls to request services from agents. In practice, all method calls are handled by a *proxy object* which provides exactly the same call interface as the target agent. The proxy forwards all calls to the actual agent by using either local method calls or RMI if the target agent is in another host. The caller does not need to know the actual location of the agent called: the proxy will take care of that.

When an agent wants to use a service via method calls, it must first ask the underlying platform to create a proxy for the service agent. Typically, platforms provide an operation which uses the agent identifier or service name as a parameter and returns an object reference to the proxy. In order to use this reference, the caller must first cast the type to the interface class of the service that must be known in advance.

Let us assume that we have created a platform independent service agent using the head-body partition described in Section 3. The implementation of the service is in the head and it is to be used through a separate service interface class. Now we would like to register this service into the underlying platform so that other agents could start using the service. Unfortunately, most platforms refuse to register the head as a service agent, because it does not derive the platform specific superclass of agents. On the other hand, we cannot register the body either since it does not implement the interface class of the service. The only solution seems to be to change the body so that it will provide the necessary interface. But then we have to make a body for each head-platform combination, or do we?

Not necessarily. In the current Monads system prototype, we use a trick that takes advantage on the dynamic class handling of the Java language. We use the same class name for the body implementation in all platforms. So, instead of naming them as `VoyagerBody` or `JadeBody`, all body implementations are named `MonadsBody`. Thus, we can create a body class for our service that extends the `MonadsBody` and implements the service interface by delegating all requests to the head, as shown in Figure 7. The same service body class can be used in all platforms — only the implementation of its superclass `MonadsBody` varies between platforms. In addition, the source code for the service body class can be generated automatically when given a list of the interfaces that are to be supported.

Use of platform independent service agents gives some additional requirements for the MAG implementation. We must ensure that the body object created for a service head is always an instance of the service body class. Therefore, we store the name of the body class into the head. When a MAG receives an agent head, it checks whether the head needs a special body class or the normal body class can be used. We must also be careful with the class loading between MAGs. The service body class should always be transferred together with the head, but the `MonadsBody` must not be transferred because every platform has its own implementation of that class.

We admit that this trick of ours violates "the spirit of pure object-oriented programming" a little, because it is based on special features of Java. There are alternatives, though. Service body classes could also be obtained in the following ways:

— When a head is created for the first time or it arrives at a platform where it has not been before, it gives a list of the interface classes that should be supported by the body extension. These classes are given to a 'body generator' which creates the Java source code for the service body. This class is compiled to bytecode by a normal Java compiler.

— The head carries a template of the service body source code with it. This is just like the code created by the body generator but the name of the superclass is left open. For each platform, the appropriate superclass name—`XBody` in platform `X`, for example—is put into the source code before compilation. Thus, we do not need to use body generator but we must transfer additional data with the head.

— If all platforms into which the agent may migrate are known in advance, the necessary service body classes can be created and compiled beforehand. Class generation and compilation during migration are avoided, but all body classes must be transferred with the head.

We have not implemented the alternative methods listed above, because they all introduce more overhead than the method we are using.

## 5. Example agent

The following example illustrates our approach. We would like to make an agent that carries a message, migrates to a given destination, displays the message using a local service, and asks for a new message and destination. We know that every agent server has a *User Interface Service Agent* (UIS) that can be used for user-agent interaction. The service is registered under the name *"UserInterface"* and implements the following interface:

```
public interface UserInterface {
  void showMessage(String message);
  String ask(String question);
}
```

We implement the agent by using the head-body partition. We derive our `MessageAgent` agent from the `AgentHead` class that is the base class for all agent heads. The actual agent program resides in the `live()` method, which is called by the body when the agent is created or when the agent has migrated. As its first action, the `MessageAgent` agent searches for the identifier of a local UIS, requests a reference to it, and asks it to display the message. Then the agent asks for a new message and the address of the destination. Finally, it asks the body to move to that destination. If the destination address points to another platform, the body uses the Monads Agent Gateway for migration, otherwise it calls the move operation of the underlying platform.

```
public class MessageAgent extends AgentHead {
  private String message;
  public MessageAgent() {
      super();
      message = "Hello!";
  }
  public void live() {
      String uis_id =
          body.findService("UserInterface");
      UserInterface uis =
          (UserInterface)body.getProxy(uis_id);
      uis.showMessage(message);
      message = uis.ask("Message?");
      String destination =
          uis.ask("Destination?");
      body.moveTo(destination);
      // called again after migration
  }
}
```

This example agent has been tested successfully with Voyager, Jade, and Grasshopper. We implemented the UIS for each platform and registered it to agent servers as a local service (in Jade we used Monads Naming Service for registration). The message agent was able to migrate between agent servers of different platforms via the MAG and use all versions of the UIS without any problems.

We also made some preliminary performance measurements to see how much overhead the Monads Agent Gate-

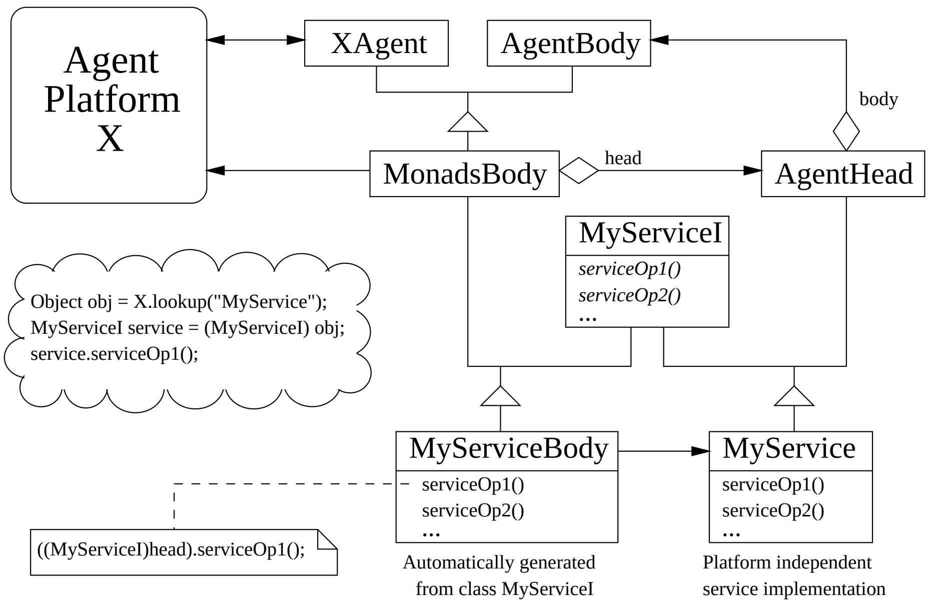
Agent Platform X
XAgent
AgentBody
body
MonadsBody
head
AgentHead
MyServiceI
serviceOp1()
serviceOp2()
...
Object obj = X.lookup("MyService");
MyServiceI service = (MyServiceI) obj;
service.serviceOp1();
MyServiceBody
serviceOp1()
serviceOp2()
...
MyService
serviceOp1()
serviceOp2()
...
((MyServiceI)head).serviceOp1();
Automatically generated from class MyServiceI
Platform independent service implementation

**Figure 7. Calling a service method**

way causes to agent migration. In these measurements, the average time needed for agent migration via the MAG was roughly the same as when using platform specific move operations. One must remember, though, that MAG is not meant to be used for agent migration inside a platform but *between* platforms. Thus, making detailed performance comparisons with other systems is impossible because alternative systems do not exist.

## 6. Conclusions

It is useful to have different kinds of agent platforms from which to choose. All platforms have their advantages and drawbacks so one can select the one which suits one's purposes best. On the other hand, having multiple incompatible systems ruins the dream of having a world-wide agent system in which agents could move between agent servers and could interact with other agents. Agent platform standards developed by FIPA and OMG may be the salvation but even if they are successful, standardization takes time.

In this paper we have taken a different approach to the problem of incompatible platforms. We have shown that some of the incompatibility problems can be solved with a special *agent architecture* in which all platform specific code is separated from the platform independent main procedure of an agent. By using our design technique it is possible to build agents that can migrate between different platforms. However, these agents can not use all features of the underlying platform. We have also described how messages can be delivered to an agent that has migrated to another platform, and how to build platform independent service agents that are used through normal method calls. The ideas presented above have already been tested successfully using Voyager, Jade, and Grasshopper platforms.

## References

[1] F. Bellifemine, G. Rimassa, and A. Poggi. JADE — A FIPA-compliant Agent Framework. http://www.practical-applications.co.uk/PAAM99/abstracts.html.

[2] G. Brasche and B. Walke. Concepts, Services, and Protocols of the New GSM Phase 2+ General Packet Radio Service. *IEEE Communications Magazine*, 35(8):94–104, 1997.

[3] S. Campadello, H. Helin, O. Koskimies, P. Misikangas, M. Mäkelä, and K. Raatikainen. Using mobile and intelligent agents to support nomadic users. In *Proceedings of ICIN 2000*. Idera, 2000.

[4] Foundation for Intelligent Physical Agents. *FIPA 97 Specification Part 2: Agent Communication Language*, October 1998.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] IKV++ GmbH. Grasshopper. http://www.ikv.de/products/grasshopper/index.html, 1999.

[7] M. Mouly and M.-B. Pautet. *The GSM System for Mobile Communications*. Mouly and Pautet, 1992.

[8] Object Management Group. *Mobile Agent System Interoperability Facilities Specification*, 1998.

[9] ObjectSpace, Inc. Objectspace voyager. http://www.objectspace.com/products/prodVoyager.asp, 1999.

[10] K. Pahlavan, A. Zahedi, and P. Krisnamurthy. Wideband Local Access: Wireless LAN and Wireless ATM. *IEEE Communications Magazine*, 35(11):34–40, 1997.