International Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'2000)

— Final Version —

TITLE OF THE PAPER: Addressing the Question of Platform Extensibility in Mobile Agent Systems

KEYWORKS: Mobile Agents, Extensibility, Software Components, Dynamic Services, JavaBeans

TYPE OF PAPER: Research

AUTHORS: Paulo Jorge Marques, Luís Moura Silva, João Gabriel Silva
CISUC, University of Coimbra, Portugal

Dep. Eng. Informática, Polo II
Universidade de Coimbra
3030 Coimbra, Portugal

{pmarques, luis, jgabriel}@dei.uc.pt

CONTACT AUTHOR: Paulo Jorge Marques

CONTACT INFORMATION: email: pmarques@dei.uc.pt
phone: +351-914144686
fax: +351-239701266

ADDRESSING THE QUESTION OF PLATFORM EXTENSIBILITY IN MOBILE AGENT SYSTEMS

Paulo Jorge Marques, Luis Moura Silva, João Gabriel Silva
CISUC, University of Coimbra, Portugal

Dep. Eng. Informática, Polo II
Universidade de Coimbra
3030 Coimbra, Portugal

{pmarques, luis, jgabriel}@dei.uc.pt

Abstract

Over the last few years, there has been a huge proliferation of mobile agent platforms, for the most different application domains. Although these platforms normally have a very interesting set of features, one common limitation found on most architectures is the lack of support for extensibility, i.e. the ability to add new features at runtime, after the platform has been written and deployed. This is an important question since it limits the usability of the platforms across application domains, and its future usefulness as software requirements change.

In this paper, we present a general mechanism for platform extensibility based on binary software components. The mechanism was implemented using the JavaBeans component model and several services where implemented and deployed. The approach undertaken is generic and can be easily adapted into existing mobile agent platforms.

Keywords: *Mobile Agents, Extensibility, Software Components, Dynamic Services, JavaBeans.*

1 Introduction

Over the last few years, a large number of mobile agent platforms have been developed for the most various application domains. Examples include Network Management [1, 2], Disconnected Computing [3], Electronic Commerce [4], and many others. These platforms normally provide a rich set of features for the most different things. There are typically several inter-agent communication mechanisms available, different types of agent tracking mechanisms, different approaches for agent migration, and so on. Nevertheless, most of today's available platforms ignore one very important point: software requirements change. The features that are available and needed today are quite different from what is required tomorrow.

Platforms are typically coded in a monolithic architecture, providing no built-in functionality for system extensibility. By system extensibility we mean the ability to add and configure new features into the platform, after it has been released in its binary form (e.g. to add a new inter-agent communication mechanism into an existing platform). This is a very restrictive limitation because:

- Once the platform is coded, the only way to add new features is by releasing a new version of the platform. If a platform is widely deployed, this can be a challenging task.
- The software requirements imposed on the platforms change, making the platform obsolete or not very useable in a short time period.
- It is not possible to incorporate solutions developed by third-party software makers, providing the features needed for the application domain, and additionally cutting development costs.

In fact, the importance of having system extensibility is well understood in today's technologies. Currently we have device drivers providing access to new hardware on our operating systems, daemon services running on top of micro-kernels, business-logic being deployed in the middle-tier of multi-tier information systems, and others.

We believe that extensibility must be properly addressed when developing new mobile agent platforms. In this paper, we present a general

architecture based on binary software components [5] for mobile agent platform extensibility. This architecture was implemented using the JavaBeans component model [6] and several services were implemented and deployed. As an exploration of the concepts, we present the implementation of one of such services – an agent tracking service.

The rest of this paper is organized as follows. Section 2 addresses the platform extensibility question by using software components. Section 3 presents the implementation of an agent tracking service. Section 4 discusses some of the lessons we learned during the project. Section 5 provides an overview of related work. Finally, Section 6 concludes the paper and presents future directions of research.

2 Platform Extensibility

Mobile agent system extensibility can be viewed in two different, but complementary aspects: the services that are made available for the agents, and the services that are made available for the applications (Figure 1).

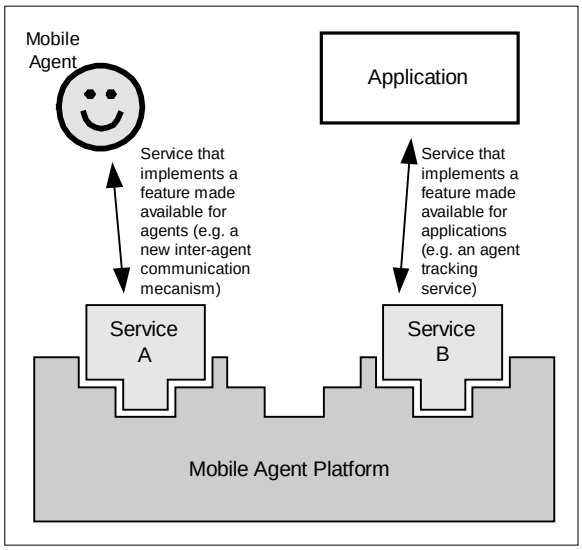

Figure 1 – Different types of services are integrated into the agent platform.

When programming an agent, the available features that an agent sees are typically fixed, defined by a well-known API. An example of such a feature is an inter-agent communication mechanism. Agents exchange messages though a well-defined interface, which is specified on the agent-programming model of the system. It is desirable that the list of available features to agents should be able to be expanded by registering new components at an agent platform, instead of being fixed by a closed API.

In the same way, the available features for the applications interacting with the mobile agents are normally fixed. For instance, it is not typically easy to add a new agent tracking mechanism to the platform, enabling external applications to monitor the agents. It should be possible to add new services at the platform level, bringing new functionalities into the system, without having to build it in a fixed way.

When considering the services themselves, several characteristics are desirable:

- The services should be self-contained modules, with clearly defined boundaries.
- It should be possible to load or unload the services at runtime, without having to shutdown the agent platform.
- The services should be able to act as an integrating part of the agent platform, as if they were integrated with it from the start.
- The services should be easy to develop and configure.
- The services should be easy to deploy on an existing agent platform.

Considering these requirements, binary software components such as JavaBeans [6] and COM/ActiveX [7] possess many of these characteristics. A binary software component is a self-contained module, with very clearly defined boundaries, which implements a well-defined functionality. The paramount objective of component-based development is to provide true reusability across application domains. For interfacing with the outside world, components typically support three features: *properties*, *events* and *methods*. *Properties* define most of the state of a component and allow configuring the runtime characteristics of the component. *Events* allow the component to be notified of changes on its surrounding environment and to notify the environment of changes in its execution state. Finally, *methods* allow operations to be performed on the component. Thus, software components provide a very interesting approach for building extensible platforms, where services are implemented as components that can be loaded and unloaded from a platform.

We will now detail how this approach was implemented, concentrating on the requirements needed to easily integrate services as components into an agent platform.

2.1 Implementing Service Components

For supporting the existence of modular services, we have defined an architecture where each service is implemented as a component. Different components implement different functionalities that are made available for the agents, for the applications or both.

The components are coupled to the agent platform through events (Figure 2). Two types of events exist: *Agent Lifecycle Events* and *Service Support Events*. These events make it possible to transparently support the existence of the two types of services previously stated: services for agents, and services for the applications.

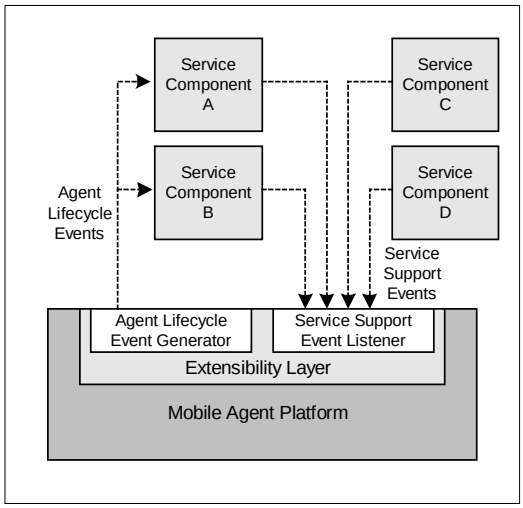

Figure 2 – Components are supported through Agent Lifecycle Events and Service Support Events.

We have written a very lightweight mobile agent platform that provides the basic means for agent mobility. On top of that platform, an extensibility layer was added that supports two functions: the generation of Agent Lifecycle Events, and the acceptance of Service Support Events.

Whenever an agent changes its execution state (e.g. arrives, departures or dies), an Agent Lifecycle Event is fired. The existing services can register their listeners with the extensibility layer, being notified whenever one of those events takes place. When such an event takes place, the listeners of the services are able to examine the agent responsible for the event and if necessary, call methods on the agent or in other modules. This is especially important for the higher-level services since they can make use of those events to accomplish their functions. For instance, a disk-persistence service can checkpoint an agent to disk after receiving an `OnAgentArrivalEvent` (one of the Agent Lifecycle Events) and remove it after receiving an `AfterAgentMigrationEvent`. For certain events like an agent arrival or an agent migration, each event listener has also the capability of vetoing the event. For instance, a security service may decide not to allow an incoming agent to migrate into the platform by vetoing the `OnAgentArrivalEvent`.

Service Support Events allow a service to be incorporated or removed dynamically from the system. A service registers itself with the extensibility layer as a source for those events and, whenever it becomes available, it fires an event that notifies the platform. If the service needs to be removed from the list of available services, it fires an event requesting to be removed from the list of available services.

One very interesting point is that since the services are written as JavaBeans components, they can be configured visually in a visual builder tool (e.g. the Beans Development Kit – BDK [8]). These components can then be serialized to disk and incorporated into the platform completely configured. The loading of the existing services is done by using the dynamic class loading capabilities of Java, which allow these beans to be instantiated at runtime.

2.2 Structure of a Service Component

We will now examine in more detail the structure of a service component.

Components that offer services at the platform level, and not for the agents, are very simple to implement. The only requisite is that they follow the JavaBeans component model [6], and that they are able to generate Service Support Events to register and un-register from the system. For simplifying this process, our system already provides a base component with this capability. The programmer only has to extend this component and implement the actual service functionality.

Most services will also implement the AgentLifecycle Event Listener in order to be able to interact with the mobile agents themselves. This is a simple interface that has several methods, each one being called whenever an agent running-state transition occurs (Figure 3).

```
public interface AgentLifecycleListener
  extends EventListener
{
  void onAgentArrival(AgentLifecycleEvent e);

  void afterAgentArrival(AgentLifecycleEvent e);

  void onAgentMigration(AgentLifecycleEvent e);

  void afterAgentMigration(AgentLifecycleEvent e);

  void onAgentFailedMigration(AgentLifecycleEvent e);

  void afterAgentCreation(AgentLifecycleEvent e);

  void afterAgentTermination(AgentLifecycleEvent e);
}
```

Figure 3 – AgentLifecycle Event Listener.

Figure 4 shows the anatomy of a service. The programmer is given a Base Agent Service Component that provides the basic functionalities in terms of firing Service Support Events and interacting with the Extensibility Layer. For creating a service, it is necessary to implement four basic classes: a *Service Descriptor*, a *Service Provider*, a *Service Instance* and finally, a *Service Interface*.

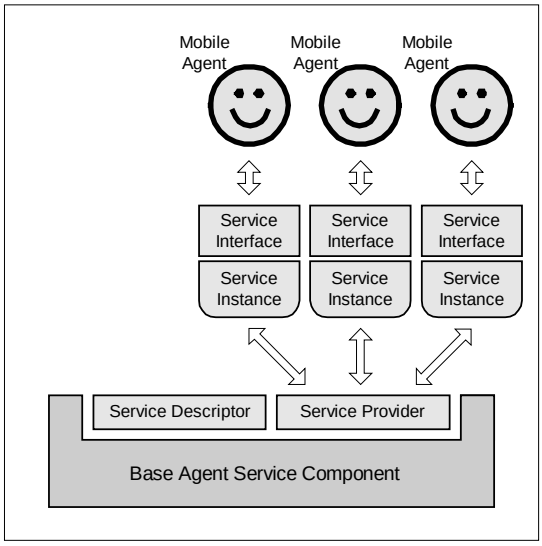

Figure 4 – Structure of a service for agents.

A Service Descriptor contains the information about the service. Its data includes: service name, version, unique identifier, description and hash-code. When a service starts running, it registers its Service Descriptor with the Extensibility Layer. Each time an agent wants to know the list of available services, it evokes the `getAvailableServices()` method on its API. The request is passed on to the Extensibility Layer, which returns the set of services that currently can be used by the agent.

The Service Provider is where the programmer actually implements the service. One important feature of this module is the `getServiceInstance()` method. Whenever an agent wants to use a service, it calls the `getService(ServiceDescriptor desc)` method of its API. The request is passed to the Extensibility Layer, and then to the corresponding Service Provider. The Service Provider may return an instance of the service (a Service Instance object), or refuse to serve the agent. This is accomplished by having the Extensibility Layer calling the `getServiceInstance()` method of the Service Provider, passing it information about the context of the calling agent.

Each agent gets a different instance object, connected to a unique Service Provider. This allows a strong separation between running agents, improving the reliability and security of all running instances. We have decided to shield the Service Provider from the agents through objects because of security reasons. The agents should only be allowed to call specific methods related to the service itself and not all the publicly available methods on the Service Provided. These proxy Service Instances allow this.

Finally, the programmer must implement a Service Interface. This interface specifies the methods that the agent is allowed to call on the Service Instance. This interface exists because an agent that gets a generic Service Instance object has to cast it to a specific service data type. Typically, an agent carries the interfaces of the services it needs, which are used to cast the base type service objects into concrete types. If an agent does not carry the interface to a specific service or arrives at a host containing a newer version of the service for which it does not have the interface, it can still use the service by evoking methods on it by using the Java introspection mechanism [6].

```
import mob.agent.*;
import mob.services.*;

public class MyAgent extends Agent {
  public void run()  {
    LogServiceDescriptor logDesc
      = new LogServiceDescriptor();

    ServiceDescriptor[] services
      = getAvailableServices();

    for (int i=0; i<services.length; i++)
      if (services[i].equals(logDesc)) {
        try {
          LogService logService
            = (LogService) getService(logDesc);

          logService.logMessage("Agent says Hello!");
          break;
        }
        catch (Exception e) {
          e.printStackTrace();
        }
    }
  }
}
```

Figure 5 – An agent using the log service.

Figure 5 shows the code of an agent requesting access to a log service, and sending a message to it. In this case, the agent carries the Service Descriptor class of the service and its interface. This is the typical case: the agents carry the identity of the service and its interface.

While we have implemented this framework in the context of a mobile agent project for network management – M&M Project, we believe the approach to be generic an easily adaptable to most mobile agent platforms.

3 Agent Tracking Service

We will now discuss the implementation of one of the services we have developed in our project, which provides agent-tracking capabilities for applications and agents.

For developing this component, two objectives were established:

- To give the capability of knowing where any agent is at a given time to applications that make use of the agent platform.
- Allow the agents themselves to know where other agents are.

It is important to distinguish these two features from what is commonly found in current mobile agent systems in terms of agent tracking. Most platforms have some sort of agent tracking built-in directly into the infrastructure. This support typically allows other internal modules to know the location of the agents. For instance, an inter-agent communication module may use the tracking support for knowing to which host to send messages, when addressing a particular agent. Agent tracking is also typically made available to the user through a Graphical User Interface. In our case, the objective was to extend a mobile agent platform with a plug-in service that allows existing applications to know about the location of the agents, and to allow the agents themselves to perform a similar task. Without rewriting the agent platform, typically this cannot be done in traditional systems.

Figure 6 shows the architecture of the tracking service. The service was implemented as a Service Provider. This service provider registers an Agent Lifecycle Event listener, being notified whenever an agent arrives, migrates or dies. Upon migration, the event listener even knows which is the destination of the agent since that information is present in the event data. Upon receiving an event notification, the component contacts a central tracking server that maintains global information about the location of all agents. The implementation of the communication channel and tracking server was kept modular, being possible to change the transport layer, and the tracking directory used at the server (native implementation or LDAP [9]).

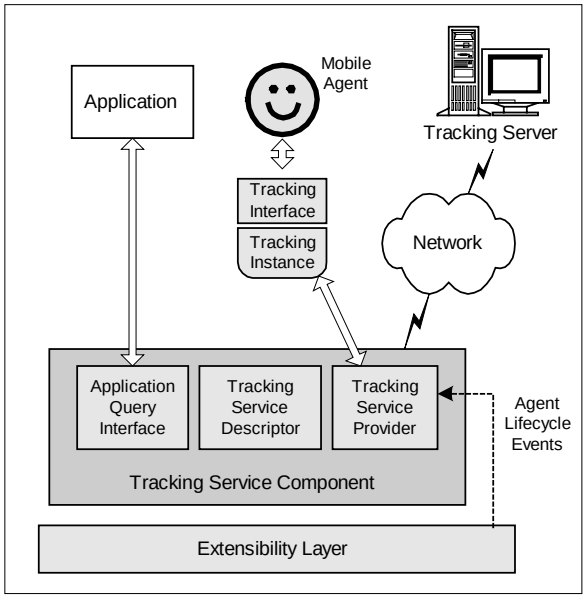

Figure 6 – Tracking service architecture.

Agents requesting access to the tracking service are given an object instance that enables them to make queries about the location of other agents. They can ask for the location of a specific agent, perform wildcard searches or ask which agents at present at a particular location. The requests are passed from the object service instance into the tracking service provider, and then the tracking server is consulted. In the end, the results are returned to the agent.

In a similar way, local applications that are not aware of the existence of the tracking server can use the Application Query Interface, which is based on Java's RMI [10], to perform similar tasks.

Thus, by using a simple mechanism like Agent Lifecycle Events, it is possible to incorporate powerful mechanisms like agent tracking into an agent platform, without having to do any re-coding and in fact, without even having to shut down the platform.

4 Lessons Learned

Building a mobile agent system with extensibility in mind, in particular by using a component-based approach, has been a rich learning experience. In this section we summarize some lessons learned while developing the system.

4.1 Two sets of events are enough

When we begun the project, we were somehow skeptical if having only two sets of events would be enough to guarantee the implementation of many different kinds of services. Our actual experience was

that in fact, having AgentLifecycle Events and Service Support Events provides a very rich framework for implementing services.

When the first few services were developed, the events had sometimes to be modified because they did not cover enough information for the several services being written. For instance, it was only when we implemented the agent tracking service that we have realized that a logical time stamp had to be included in the event information, to make events valid in a distributed computing context. Nevertheless, after writing a few services, we stopped changing the information included in the events. Since then many services have been written, and we have not found the need to add more events, or to couple the services with the platform in a tighter way.

4.2 Services can be hard to develop

In general, simple services are easy to develop. Our framework already provides base components that are very easy to extend into fully blown services. Nevertheless, sometimes services can be hard to develop.

If several services require communication with services on other platforms, or with applications running on other hosts (e.g. the tracking server), then it becomes difficult to manage and maintain the configuration of all the components. The reason is that since the services are not integrated in the core of the platform, it is necessary to explicitly configure the properties of each component in terms of client/server interface. We are currently working on an approach that eases these configuration and maintenance tasks. The idea is to use JINI [11] for dynamically downloading the services into the required platforms, without having to specifically configure the client/server interface of the services.

Versioning of components is also a hard topic. This is not a specific problem of having components for mobile agent services, but a more general problem of component-based development [5]. The problem has to do with agents having the interfaces for older versions of existing services, which are no longer compatible. Although the service descriptors clearly identify the version of the service being run, in order to maintained backward-compatibility, sometimes services have to be run side-by-side. Many times this is a necessary burden on the system. Although Java has some support for component versioning, a lot more work has still to be done.

We have found error handling also to be a complicated question. The problem is that services are running on top of the agent platforms. When developing a component, many times it is hard to decide what is the appropriate action to take, since the complete environment where the component will run is not known. For instance, if a component fails to connect to a server, what should it do? Retry, abort or ignore? In depends a lot on the running environment. The service may be critical is some applications, but only desirable in others. In general, the programmer must consider all the alterative scenarios where the service will run, and try to make the most conservative choices. Again, this is not a specific problem of ours, but of component-based development.

4.3 Extensibility is very useful

In global terms, we find that the benefits of having extensible mobile agent platforms outstand the difficulties of developing robust services. The systems become very flexible, capable of incorporating new features without having to be rebuilt. Having the capability of including new services at runtime without having to shutdown the platform is important for having zero-downtime systems. Finally, we have found to be easy to adapt existing third-party components to our system, quickly bringing new functionalities for the agents and applications.

5 Related Work

The MOA platform from the OpenGroup [12] is a component-based mobile agent platform, allowing different modules to be configured in an easy way. Since MOA was an industrial project, there is not much available information concerning its extensibility and its component-based aspects. Even so, one fundamental difference between our work and the MOA project is that we specifically address the problem of system extensibility. From what we could infer, MOA is only a configurable agent-platform, not having a specific programming framework conceived for extensibility.

The two works most related to ours are JIAC [13] from TU-Berlin and Gypsy [14] from TU-Vienna. JIAC is a component toolkit for building intelligent agent systems for telecommunication applications. In JIAC components are scripts that can be plug in into an agent backbone or into a place. Gypsy is a component-oriented mobile agent system. In Gypsy everything is a JavaBean component. Agents are components that run inside of places. Places can be assembled by connecting several components and host the agents. In our work, we specifically address the requirements needed to build an extensible mobile agent platform. We provide a clear model on how to develop services, and how these services are coupled with an agent platform by two sets of events. Both in Gypsy and in JIAC, the focus was not on studying how to build an infrastructure that generally supported adding new services, although some of the topics were addressed.

6 Conclusion

In this paper, we have presented a framework for building extensible mobile agent platforms, based on component-based development. By using such an approach, it is possible to have mobile agent platforms that incorporate new services after they have been built and deployed, without having to recompile, or shutdown the system.

Although the approach was implemented using a particular mobile agent platform, we believe it is generally applicable to any system without much effort.

We are currently working on the problems make the use and development of services hard: configuration, deployment, versioning and error handling. In particular, we are considering JINI for dynamically and transparently deploy networked services at the agent platforms.

Acknowledgments

This investigation was partially supported by the Portuguese Research Agency FCT, through the program PRAXIS XXI (scholarship number DB/18353/98) and through CISUC (R&D Unit 326/97).

References

[1] M. Breugst, S. Choy, L. Hagen, M. Hoft, and T. Magedanz, *Grasshopper - An Agent Platform for Mobile Agent-Based Services in Fixed and Mobile Telecommunications Environments*, Proceedings of the Software Agents for Future Communication Systems Workshop, 1998.

[2] L. M. Silva, P. Simões, G. Soares, P. Martins, V. Batista, C. Renato, L. Almeida, and N. Stohr, *James: A Platform of Mobile Agents for the Management of Telecommunication Networks*, Proceedings of the 3rd International Workshop on Intelligent Agents for Telecommunication Applications (IATA'99), Stockholm, Sweden, 1999.

[3] E. Kovacs, K. Rohrle, and M. Reich, *Mobile Agents OnTheMove - Integrating an Agent System into the Mobile Middleware*, presented at ACTS Mobile Summit 1998, Rhodos, Greece, 1998.

[4] T. Sandholm and Q. Huai, *Nomad: Mobile Agent System for an Internet-Based Auction House*, Internet Computing, vol. 4, issue 2, 2000, pp. 80-86.

[5] C. Szyperski, *Component Software, Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[6] Sun Microsystems, *JavaBeans Specification 1.01*, ttp://java.sun.com/beans/docs/spec.html, Sun Microsystems, 1997.

[7] D. Rogerson, *Inside COM*, Microsoft Press, 1996.

[8] Sun Microsystems, *The Bean Development Kit*, http://java.sun.com/beans/software/bdk_download.html, Sun Microsystems, 1998.

[9] W. Yeong and T. Howes, *Lightweight Directory Access Protocol*, RFC 1777, Network Working Group, 1995.

[10] Sun Microsystems, *Java Remote Method Invocation - Distributed Computing for Java*, http://java.sun.com/marketing/collateral/javarmi.html, Sun Microsystems, 1998.

[11] K. Arnold, B. Osullivan, R. Scheifler, J. Waldo, A. Wollrath, and B. O'Sullivan, *The Jini Specification*, Addison-Wesley, 1999.

[12] D. Milojicic, D. Chauhan, and W. laForge, *Mobile Objects and Agents (MOA), Design, Implementation and Lessons Learned*, Proceedings of the 4th USENIX Conference on Object-Oriented Technologies (COOTS), 1998.

[13] S. Albayrak and D. Wieczcorek, *JIAC - A Toolkit for Telecommunication Applications*, Proceedings of the Intelligent Agents for Telecommunication Applications Workshop, Stockholm, Sweden, 1999.

[14] W. Lugmayr, *Gypsy: A Component-based Mobile Agent System*, Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000), Rhodos, Greece, 2000.