Comparing Object Oriented Mobile Agent Systems*

Thomas Gschwind
tom@infosys.tuwien.ac.at
Distributed Systems Group
Technische Universität Wien

Abstract

Mobile agents are a new paradigm for distributed applications and an increasing number of mobile agent platforms emerged recently. Unfortunately, most of these systems are neither compatible to existing standards nor among each other. In this paper we compare three prominent industry platforms and promising research systems regarding their object oriented design and component model. On the basis of this comparison we identify their common design patterns. This helps us to define the requirements for an abstraction layer that allows mobile agents to move between different mobile agent platforms. Agents written on top of the abstraction layer can be moved across different mobile agent platforms.

Keywords: mobile agent systems, system architecture, design patterns

1 Introduction

Mobile Agents, programs that move from computer to computer are a promising paradigm. The principle of the mobile agent approach is that a local method call is faster than a remote procedure call. This is due to the fact that remote procedure calls have to marshal and unmarshal the arguments passed to and the value returned from the remote procedure. Additionally, each remote procedure call suffers from the latency of the underlying network. Keeping that in mind, the following conclusions can be derived:

- Mobile Agents can perform faster if multiple requests need to be submitted to fulfill the agent's task.
- Less network bandwidth is required if the agent has to request several different data items and is able to combine these items (e.g., if the agent is only interested in the sum). Thus, only a fraction of the information requested will be returned via the network. In [FPV98], this is referred to as semantic compression.
- It allows access to the remote server's data on a finer grained level. There's a smaller performance penalty, if several smaller functions need to be called to achieve the agent's goal [Fis00].

Before employing mobile agents, one has to take the pros and cons of mobile agents versus traditional approaches into account carefully. Additionally, one must not forget that the agent's code incurs some additional overhead.

While the trade-off between using mobile agents and traditional paradigms are well known, there has been little work focusing on the design of mobile agents and their execution platforms. A huge number of *home-grown* systems exists, but little work has been done in categorizing them in respect to their component architecture and object oriented design, or simply by the design patterns employed in these systems. We only have found some papers comparing those systems on a high level view like their communication mechanism, security issues, or their programming language [CGPV96, KZ97, FPV98].

Another big concern is the interoperability between different agent systems. As long as every

*This work was supported in part by an IBM University Partnership Award from IBM Research Division, Zurich Research Laboratory.

agent system designer focuses on his system without dealing with interoperability, mobile agents will never get as popular as the client/server-paradigm for instance.

Achieving interoperability between agent systems is not always possible. At least, the agent system needs to be able to interpret/execute the agent. This limits the interoperability discussion to agent systems providing the same interpreter/virtual machine to their agents. The problem, however, is still challenging since different agent systems, though written in the same language, use different approaches to identify and transfer an agent.

We have chosen to focus on agent systems written in the Java programming language [AG97] since most agent systems are written using it and it is an object-oriented programming language. It provides essential services for mobile agent systems like object serialization, a portable class format, standardized thread support, and a security manager providing basic access control to low-level operating system services.

In this paper, we will discuss three different mobile agent systems: Aglets [LO98, IBM99], Grasshopper [IKV99a, IKV99b], and Voyager [Obj98a, Obj98b]. All these systems are either prominent industry or interesting research platforms.

Section 2 will be giving an overview on the principles of mobile code, as well as a short description of the agent systems we have chosen for this paper. Section 3 will present different design choices taken by these agent systems. Based on this, we will extract the requirements for an abstraction layer that allows agents to be executed on different agent systems in Section 4. Sections 5 presents future work regarding the abstraction layer and in Section 6 we draw our conclusions.

2 Mobile Agents

Before we will analyze the design of the agent systems, we will present the terminology we will be using and will give a short overview of the candidates selected for analysis.

2.1 Basic Agent Definitions

Throughout this paper we will adopt the definitions of OMG's Mobile Agent System Interoperability Framework (MASIF) [MBB+98]. OMG defines an *agent* as a computer program that acts autonomously on behalf of a person or an organization, and having its own thread of execution so tasks can be performed on its own initiative.

In this paper, however, we will focus on *mobile agents* defined as agents that are not bound to the system where they begin execution and having the ability to transport themselves from one system in a network to another. Since this paper focuses on mobile agents we will refer to *mobile agents* simply as *agents*.

An *Agent System* is a platform that can create, interpret, execute, transfer, and terminate agents. When a mobile agent transfers itself, the agent travels between execution environments called *places*. A place is a context within an agent system that provides a uniform environment in which an agent can execute. It provides the means for managing mobile agents, enforcing security policies and accessing local resources.

The *minimal* requirements of a mobile agent system is the ability to transfer the state (excluding the execution stack) of an agent and to resume its execution on the destination host. The RMI64 agent system demonstrates this [Gsc99]. RMI64 is intended for educational purposes and consists of 48 lines of code only. Besides these *basic* requirements, an agent system usually provide one or more of the following services:

Code Transfer: Typically, the code to be executed by an agent is not available on all of the agent's destination hosts. Thus, most agent systems provide a mean for transferring the code of the agent. Either by transferring it from a centralized location (codebase) or from the host the agent previously was executing on.

Agent Identifier: Many agent systems assign a unique identifier to an agent to allow agents to address each other and to register themselves at a naming server if necessary.

Authentication/Authorization: Employed on an Internet like scale, mobile agents cannot be trusted. Thus many agent systems provide

additional information about an agent. Who has created the agent, or who is deploying it. Based on that information the agent's permissions are being set.

Before we will discuss the design of the agent systems, we present the systems, we have chosen and some excerpts of sample agents which will be used for illustration in Section 3.

One of the agents provides a distributed registry where objects, including other registries, may be registered. Two other agents provide a mean for locating objects in the registry: a mobile agent to locate information stored in the registry, and a stationary one for collecting the results collected. The interaction between the agents is shown in Figure 1.

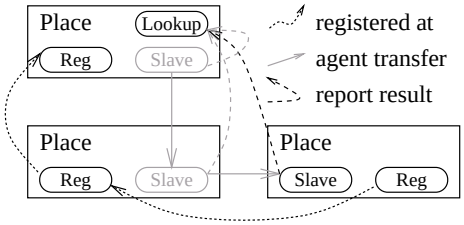

Figure 1: A sample scenario using agents

2.2 Aglets

Aglets [IBM99] is probably the most famous agent system among the systems we have chosen. The reason for Aglets' popularity might be its ease of use. Agents written with the Aglets Software Development Kit are referred to as Aglets. The name comes from the fact that Aglets tries to mimic the conventions used for Applets but with adaptations to Agents.

Aglets are executed within an aglet context, referred to as a place in OMG's terminology. A context is the agent's workspace that is responsible for maintaining and managing running agents [LO98]. Using this context, an aglet can get references to other aglets.

For loading the classes of an agent, Aglets uses its own mechanism. Classes can be loaded centrally or from the aglet's previous host. As a mean for reducing the required network bandwidth, Aglets also facilitates caching using its own cache-manager [OKO98].

```
class RegistryAglet extends Aglet {
  ...

  /** initialize the agent */
  public void onCreation(Object init) { ... }

  /** handle messages sent by other agents */
  public boolean handleMessage(Message msg) {
    ...
  }

  /** functions provided to agents residing
    * at the same place */
  Enumeration getNames() { ... }
  Object getObject(String name) { ... }
}
```

Figure 2: Registry agent implemented with Aglets

An aglet implementing a registry agent is shown in Figure 2. An Aglet inherits Aglet and is initialized by the onCreation method. Aglets residing on the same system may call each other's methods, while aglets residing on different agent systems typically communicate via a message passing mechanism. Whenever a message is sent to an aglet, its handleMessage method will be called.

2.3 Grasshopper

Grasshopper [IKV99a] is a mobile agent environment similar to IBM's Aglets system. Grasshopper, however, is the first mobile environment which is compliant to OMG's MASIF specification [MBB+98].

Like Aglets, Grasshopper uses its own class loader for the agents allowing, similar to Aglets, to load the agent's code from a central codebase or from the agent's previous location. Where the classes are to be retrieved from has to be specified upon creation of the agent.

The agents implementing the search functionality of our sample scenario are shown in Figure 3. The agents are initialized by the init method and the agents code belongs to the live method. In contrast to Aglets, Grasshopper allows to differentiate between mobile, and stationary agents. Instead of message passing, Grasshopper uses an RMI like inter-agent communication. The methods that may be called remotely have to be specified in an in-

```
class LookupAgent extends StationaryAgent
implements ICollector {
  ...

  /** initialize the agent */
  public void init(Object[] args) { ... }

  /** look for entries in registry */
  public void live() { ... }

  /** collect matching objects returned
    * by LookupAgentSlave */
  public void collect(Vector objs) { ... }
}
```

(a) The LookupAgent

```
class LookupAgentSlave extends MobileAgent {
  ...

  /** initialize the agent */
  public void init(Object[] args) { ... }

  /** look for entries in registry */
  public void live() { ... }
}
```

(b) The LookupSlaveAgent

Figure 3: Lookup agents implemented with Grasshopper

terface that has to be implemented by the agent (ICollector in our scenario) and depending on the Java version used, the stubs have to be created manually (Java 1.2) or can be created automatically (Java 1.3).

2.4 Objectspace Voyager

While IBM's Aglets and Grasshopper are *full-fledged* mobile agent system, Objectspace's Voyager [Obj98a] is not primarily designed as a mobile agent platform. Thus, Voyager itself provides only a simple approach that allows for loading classes remotely. On startup of Voyager, the user can specify a set of codebases where classes are to be loaded from. The set of codebases is the same for all the classes and does not depend on the agent to be instantiated.

```
class LookupSlave {
  ...

  /** initialize "agent" */
  public LookupSlave(ILookup master,
    IRegistry reg, String sstr) {
    ...
  }

  /** look for matching objects */
  public void lookup() { ... }
}
```

Figure 4: The slave lookup agent implemented with Voyager

An object that could provide the lookup functionality is shown in Figure 4. In contrast to Aglets or Grasshopper, no special initialization function exists. Voyager provides transparent interface and stub generation for an object, if no interface is provided explicitly. This allows to call the public methods of any object remotely, thus simplifying the interaction between different objects.

Objectspace Voyager rather provides a full ORB that can be used as an infra-structure for distributed objects and agents. This explains Voyager's focus on objects in general compared to Aglets' focus on agents.

3 Agent System Design

When designing a mobile agent system, several key issues like the creation or the type of an agent within the system need to be considered. In the following sections, we will discuss these issues and how they have been solved by the selected agent systems.

3.1 Type of an Agent

One of the most important aspect of a mobile agent system is the type it is using for the denotion of mobile agents. As we will see, this predetermines most of the other implementation issues of the agent system.

Aglets and Grasshopper use a special type to denote an object as agent. This can be done either by supplying an existing agent class and let the user

derive the agents from it or by providing an agent interface to be implemented by the agent. Aglets and Grasshopper both have chosen to use code inheritance as shown in Figures 2 and 3. While Aglets provides a single class that has to be inherited by every agent, Grasshopper provides different classes depending on whether the agent is transferable, stationary, and/or persistent (Figure 3).

As Java does not provide a mean for multiple (code) inheritance, this solution restricts the user of the agent system and is a source for problems. This gets apparent when looking at the different base classes provided by Grasshopper: MobileAgent, StationaryAgent, PersistentMobileAgent, PersistentStationaryAgent. The problem, however, gets even more complicated when the user of the agent system needs to inherit from another class. It would have been cleaner to use interfaces and to factor out the mobility and persistence aspects using two different interfaces.

Except for RMI64, none of the systems presented in this paper is using an interface to denote an object as agent. Compared to inheritance, one disadvantage is that the agent system cannot force the agents to provide some basic functionality. For instance, the Aglet superclass provides final methods like getAgletID, dispose, etc. which are the same for every agent since they cannot be overridden by the agent class. On the other hand, this functionality could be provided by other classes of the agent system, or some wrapper classes generated for each agent (i.e., in Aglets by the AgletsContext).

Another possibility is to allow the user to use any serializable object as an agent. This approach is taken by Voyager as shown in Figure 4. It is the most flexible, but also the most dangerous approach. It is flexible because it allows to convert existing objects with little or no changes into mobile agents. On the other hand, it is dangerous because objects that must not be moved might accidentally be regarded as mobile agents. This can yield obscure errors and hinders early error detection. Depending on the agent system's implementation, this approach might also require heavy use of Java's reflection API which would make the code less readable.

3.2 Agent Creation and Initialization

An agent can be created using different strategies. In the simplest case, an agent is an object created by instantiation via Java's `new` operator. Another approach is to create the agent by calling a factory method provided by the agent system. While Voyager supports both, Aglets and Grasshopper only allow agents to be created by a factory method. The advantages of using a factory method are that the agent does not explicitly have to be registered with the agent system and that the agent system has the chance to provide some agent initialization prior to instantiating the agent (e.g., creating a new class loader for the agent) as well as to update some of its internal data structures thereafter.

As we have seen in Section 2, the agent systems provide different means for initializing the agent. In Voyager this is done by the agent's constructor as one might expect it. Aglets and Grasshopper, however, instantiate the agent using the default constructor, but expect the agent to be initialized by a special initialization method (onCreation and init respectively). According to the Grasshopper manual, the reason has to do with the GUI for the configuration of the agents.

The straight-forward approach to configure an agent (e.g., its itinerary, the information it should be looking for, etc.) is to put all the agent's configuration code into the constructor, the onCreation, or the init method. While tempting, this approach has a disadvantage. When the agent is moved to another host, all the agent's configuration code will be transferred as well, though chances are small that it will be executed again within the agent's lifetime. This adds to the size of the mobile agent and should be avoided.

A different approach is to exclude the configuration code from the agent and expect the agent's creator to configure the agent appropriately. This means that the agent's configuration code must be present in all the classes where the agent is created. Besides breaking encapsulation, this leads to unnecessary code duplication and a maintenance nightmare.

The approach used by the AgentBean Development Kit [GFP99], a toolkit for the creation of mobile agents from software components, is to use a configurator class that firstly instantiates the agent

and configures the agent according to its default setup. Afterwards, only the agent is returned. When that agent needs to be transferred to another host only the agent classes, without the configuration class need to be transferred (besides the agent's state). Another advantage is that it also cuts down memory requirements since the configurator object can be garbage collected immediately after the agent has been created.[1]

This approach can be seen as a specialized version of the factory pattern [GHJV94], a factory that only instantiates a single type of agents.

3.3 Agent Revival

After transferring the agent to its destination host, several methods can be used for reviving the agent. If the agents of the agent system have to be of a well known type, a callback method defined by the agent's interface can be used.

This approach is taken by Aglets [IBM99] calling the agent's run method and Grasshopper [IKV99b] calling the agent's live method. In this case, the agent explicitly has to manage its state because the callback method always remains the same. While this makes the agent system smaller, it can increase the size of the agent itself. Frequently, this leads to huge and unmaintainable switch statements within the agent as demonstrated by some examples found in both the Aglets and the Grasshopper documentation.

Fortunately, Aglets, provides Itinerary classes that manage different kinds of tasks allowing to circumvent this problem. These classes can be used to move an agent to a new destination and execute a given task upon arrival. This is also demonstrated in one of Aglets' sample agents. The idea is to use a task-attribute within the agent that points to different task-objects encapsulating the different tasks to be performed by the agent. By setting the task-attribute to a given instance different tasks can be executed depending on the agent's state without having to use a switch statement. The downside of this approach, however, is that it increases the number of classes used by the agent and thus the overhead of transferring the agent.

Another approach, taken by Voyager, is to make use of Java's reflection API to call any method specified by the user [AG97]. Java's reflection API allows to query the method names of a class/object and to invoke a method dynamically using a string representation of its name and an object array for the arguments.

This approach allows any callback method to be specified for the agent's revival at the destination host since the method to be called can be derived from its name and the arguments passed to it.

As Voyager handles every object as agent and does not impose any restrictions on the object, Voyager has to use this approach. There is a small limitation in Voyager, however. The signature of the callback method is hardcoded into Voyager. This is to avoid overloading resolution at runtime. We think that this is due to performance and simplicity reasons.

```
class LookupSlave {
  ...
  /** create new slave and transfer to the
    * place the registry is residing on */
  public static void spawn(ILookup master,
    IRegistry reg, String sstr) {
    try {
      LookupSlave ls=
        new LookupSlave(master,reg,sstr);
      String url=Proxy.of(reg).getURL();
      Agent.of(ls).moveTo(url,"lookup");
      ...
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

Figure 5: Creating and transferring an agent using Voyager

An excerpt of the LookupSlave agent creating a new instance, moving that instance to a new place, and reviving the agent by calling its lookup() method is shown in Figure 5. Whenever Voyager needs to move an object, it is wrapped into an object implementing the IAgent-interface providing the set of methods for moving the object to a remote system and the data necessary to revive the agent. The name of the callback method and the arguments to be passed to it are specified as part of the IAgent's moveTo-method.

A third alternative would be the use of *strong*

[1] Looking at today's memory prices this advantage is of less significance.

code mobility [FPV98]. Strong code mobility means that the agent is transferred along with its execution stack to the destination system, where it is revived exactly at the point where its execution was stopped. This is also known as process migration. However, in this paper we only consider *weak mobility*, which defines the ability to transfer code and initialization data, but not the execution state. Usually, the mobile agent decides itself when to be moved to another host, thus the agent's state is well defined and can easily be stored in one of the agent's attributes. Additionally, the size of the agent's state is difficult to predict depending on the current task of the agent (e.g., computing a recursive function compared to a single loop).

3.4 Comparison

Based on the above comparison, we think that today's agent systems should use interfaces to denote an object as agent. Using inheritance is not necessary because the information that could be provided by the superclass and that must not be overridden by the agent can be easily provided by methods of agent system classes or by a wrapper class of the agent.

As demonstrated by the Voyager system, only the constructor should be responsible for initializing an agent and not a specialized initialization function. When an agent offers complex configuration options, it is also important to decouple the configuration code from the agent's logic to reduce network bandwidth. This can be achieved by using two different classes: one for the agent, and the other for the configuration. In theory, it should be even able to use a single agent class and perform a "dead code" analysis to eliminate the agent's configuration code automatically. Unfortunately, we are not aware of any system implementing this approach.

For the revival of the agent, both approaches should be provided to the user: a callback method defined by the agent's interface and the use of reflection as the advantages and disadvantages of each approach depend on the application. For instance, if the state of the agent is simple as in the LookupSlave class, or if network bandwidth is not of concern, using the callback method provided by the agent-interface is perfectly sufficient.

4 Abstraction Layer

Unfortunately, most agent systems are not interoperable. For instance the agents we have implemented for every agent system for our sample application depicted in Figure 1 cannot be interchanged with each other. The goal of the abstraction layer is to overcome some of these drawbacks and to allow mobile agents to migrate between different agent systems. To attack that problem several strategies are possible:

Our first idea was to implement an agent that can migrate between different agent systems. The agent would have to act like a chameleon whenever it migrates to a different agent system, it would have to adapt the key agent class. For instance, in case of Aglets, the agent class would have to be derived from the Aglets class. This approach, however, has a huge drawback: the agent has to adapt itself requiring the availability of the classes of all the agent systems, the agent might want to visit. Obviously, this approach is impracticable.

Another approach is to implement a layer for each agent system that abstracts the most important services of the respective agent system (at least the functionality to move an agent to and receive the agent from another agent system). Unfortunately, if each agent system has to be reconfigured to add support for the abstraction layer, its acceptance would be rather limited and questionable. But still, the benefits of using this abstraction layer (compared to the installation of another agent system) would be the reuse of the places installed at the agent system, reduced license costs, and the administrator would not have to learn a new system. Additionally, installing a new agent system is not always possible.

4.1 Design and Implementation

The approach we have chosen, is a hybrid version of the above. We suggest to implement the abstraction layer as an agent itself. This allows to send an agent to each agent system that should support the abstraction layer. The abstraction layer agent would be responsible for moving and maintaining all the agents that want to make use of the abstraction layer. The abstraction layer agent consists of two parts: an agent system independent and an agent system dependent part. The former

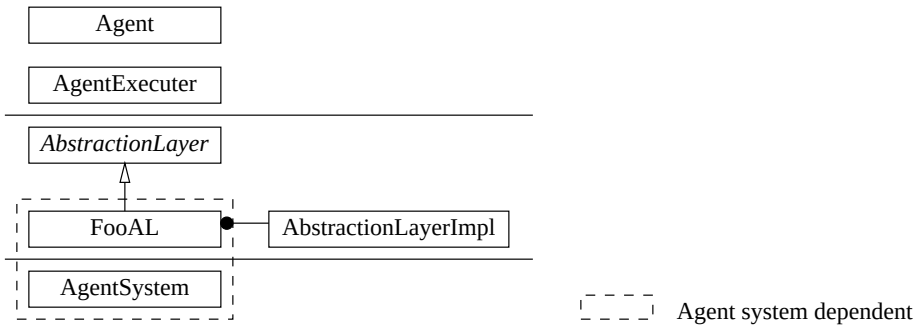

Figure 6: Classes used by the various abstraction layers.

defines how the agents using the abstraction layer are to be transferred between the agent systems. The latter provides the glue between the abstraction layer and the according agent system. It is responsible to allow the agent to communicate with other agents/services of the agent system and to simulate functionality that might be missing in the underlying agent system.

Depending on the agent system, the agents received by the abstraction layer might be injected as agents that do not differ from native agents. If this is not possible, however, the agent has to be executed as sub-agent of the abstraction layer and the abstraction layer is responsible to forward requests between the local agent system's services and the according agents.

The interaction between the agent system dependent and independent parts is show in Figure 6. On the bottom resides an agent system and one of the agents executed by the agent system is the abstraction layer agent (FooAL). The abstraction layer agent implements the abstraction layer interface used by the AgentExecuter residing above it. The abstraction layer agent also uses a support class that implements all the functionality that will remain the same across the different abstraction layer agents.

Whenever a new agent is received by the abstraction layer agent, a new class loader [Gon98] is created and within that class loader a new AgentExecuter class is created. Thus, the AgentExecuter is unique for each agent. The AgentExecuter also provides methods for querying the classes used by the agent and a reference to the agent system dependent part of the abstraction layer used to create the agent. To avoid bloating some of the methods, this functionality is implemented via static methods providing no restrictions to the agent itself, as each agent makes use of its own AgentExecuter class.

As shown in Figure 7, the abstraction layer classes make use of the Bridge [GHJV94] pattern to provide the abstraction layer for different mobile agent systems. The AbstractionLayerSupport class provides all the functionality that will be common for all the different abstraction layer agents. Since most agent systems even use different transport mechanisms, we had to define our own one. To allow agents to be received by the abstraction layer, the support class makes use of the AbstractionLayerServer class listening for incoming agents.

As mentioned before, the AgentExecuter provides the environment for the actual agent to execute. It allows the agent to obtain a reference of the abstraction layer and thus to be moved by it. The AgentDescription provides the description of an agent to be moved. It stores the serialized state of the agent, the classes necessary for its execution, and the callback method to be executed upon revival.

Additionally, the abstraction layer has been designed to put as little requirements as possible onto the agent itself. The agents written for the abstraction layer neither should be required to extend a given agent class as it is the case with the Aglets system, nor should the agents be required to implement a given interface. This eases the task of porting an agent to the abstraction layer and to allow agents to be written that work in cooperation with the abstraction layer and other agent system at the same time. As we have seen, this implies that we have to use Java's reflection API to revive

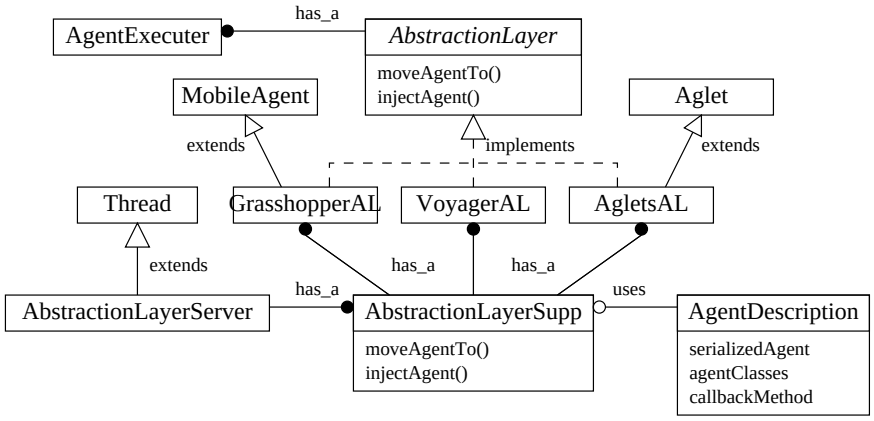

Figure 7: Classes used by the various abstraction layers.

the agent on the destination system.

5 Future Work

However, just letting the agent migrate between different mobile agent systems is not enough. There needs to be an infrastructure that allows the agent to adapt to the services available on each place since the interfaces might not be the same, even though the kind of service is the same. This gains even more importance when the agents are able to migrate between different agent systems.

So far the system presented in this paper does not provide any facilities for discovering locally available services and the agent's automatic adaptation to these services.

In future versions of the abstraction layer, we also plan to adhere to the MASIF specification. We assume that this will increase the acceptance of the abstraction layer as it would provide a simple and easy to use solution to make an agent system MASIF compliant.

6 Conclusions

We have given a description of the design issues of mobile agent systems based on three leading mobile agent/object systems. The systems we have presented form a spectrum with Aglets on one and Voyager on the other side.

Aglets represents mobile agent systems that focus on mobile agents that move from host to host and primarily interacting with local objects. Objectspace Voyager, on the other hand, represents mobile object systems that focus on the interaction of objects residing on different hosts, but also providing basic support for mobile agents. Grasshopper lies somewhere in the middle of this spectrum as it is similar to Aglets but provides some features available in Voyager. Which system to use depends on the actual use case. If the user only needs an agent system, it might be an overkill to use Voyager.

Depending on the system, we can also identify different approaches used to denote the type of an agent. We have seen that using a specific type to tag an object as agent might decrease the flexibility but increases maintainability. For instance Aglets and Grasshopper agents have to inherit a given base class. Voyager, however, regards every serializable object as movable and allows its transfer.

Based on our experiences, we have designed an abstraction layer that allows agents written for the abstraction layer to move across different agent systems. This is a first step towards interoperability of different agent systems. Unfortunately, the current implementation does not yet support the abstraction of the different services provided by other agents of the respective agent systems.

One disadvantage of the abstraction layer is that agents using the abstraction layer are not indistinguishable from native agents of the respective

agent system. This is due to the inherently different representation of an agent by the different agent systems and thus has to be attacked by the agent systems themselves. The problem is that agents transferred via a different mechanism than the agent system's native transport protocol(s) cannot be registered with the agent system. For future agent systems, however, we envision that the task of transferring and injecting an agent will be implemented as a service like any other service, thus allowing to inject an agent that is indistinguishable from a native agent.

Acknowledgements

The author would like to thank Sebastian Fischmeister for implementing the abstraction layer for the Grasshopper system and the fruitful discussions about this topic.

References

[AG97] Ken Arnold and James Gosling. *The Java Programming Language (Java Series)*. Addison-Wesley, 2nd edition, December 1997.

[CGPV96] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. A characterization of mobility and state distribution in mobile code languages. In *2nd ECOOP Workshop on Mobile Object Systems*, July 1996.

[Fis00] Sebastian Fischmeister. Building Secure Mobile Agents: The Supervisor-Worker Framework. Master's thesis, Technische Universität Wien, 2000.

[FPV98] Alfonso Fuggetta, Gian P. Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5), 1998.

[GFP99] Thomas Gschwind, Metin Feridun, and Stefan Pleisch. ADK—Building Mobile Agents for Network and Systems Management from Resuable Components. In *Proceedings of the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, October 1999.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Gon98] Li Gong. Secure Java Class Loading. *IEEE Internet Computing*, 2(6):56–61, November/December 1998.

[Gsc99] Thomas Gschwind. RMI64. http://-www.infosys.tuwien.ac.at/Staff/tom/-Projects/rmi64/, 1999.

[IBM99] IBM. Aglets Software Development Kit. http://www.trl.ibm.co.jp/aglets/, March 1999.

[IKV99a] IKV++ GmbH. Grasshopper 2: The Agent Platform. http://www.ikv.de/products/-grasshopper2/index.html, March 1999.

[IKV99b] IKV++ GmbH. *Grasshopper Programmer's Guide*, December 1999.

[KZ97] Joseph Kiniry and Daniel Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, pages 21–30, July/August 1997.

[LO98] Danny Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Computer & Engineering Publishing Group, 1998.

[MBB+98] Dejan Milojicic, Markus Breugst, Ingo Busse, John Campbell, Stefan Covaci, Barry Friedman, Kazu Kosaka, Danny Lange, Kouichi Ono, Misuru Oshima, Cynthia Tham, Sankar Virdhagriswaran, and Jim White. MASIF, The OMG Mobile Agent System Interoperability Facility. In Kurt Rothermel and Fritz Hohl, editors, *Proceedings of the Mobile Agents'98*. Springer, September 1998.

[Obj98a] ObjectSpace, http://www.objectspace.com/-products/voyager1.htm. *ObjectSpace Voyager 2.0*, 1998.

[Obj98b] ObjectSpace. *Voyager Core Technology 2.0 User Guide*, 1998.

[OKO98] Mitsuru Oshima, Günther Karjoth, and Kouichi Ono. Aglets Specification 1.1 Draft, September 1998.