Mobile Agent Interoperability Patterns and Practice

Ulrich Pinsdorf and Volker Roth
Fraunhofer Institut für Graphische Datenverarbeitung
Rundeturmstraße 6, 64283 Darmstadt, Germany

E-mail: {ulrich.pinsdorf|vroth}@igd.fhg.de

Abstract

A major setback for mobile agent technology is a lack of interoperability between systems for mobile agents which prevents them from reaching "critical mass." In this paper, we analyze the requirements for interoperability, and present design patterns which support interoperability between systems for mobile agents. We tested our patterns by adding support for Jade agents as well as for Tracy agents within our own mobile agent server SeMoA. The results of our experiments and our conclusions are summarized.

Keywords*: mobile agents, interoperability, design patterns, security, Java*

1. Introduction

A major setback for mobile agent technology is – apart from a frequently cited absence of appropriate security mechanisms – a lack of interoperability between systems for mobile agents, which prevents mobile agents from reaching "critical mass" for widespread application. Interoperability is required where systems of different vendors come into contact with each other. More precisely, we define *interoperability* of systems for mobile agents as follows:

> *Two mobile agent systems are interoperable if a mobile agent of one system can migrate to the second system, the agent can interact and communicate with other agents on this system (or even remote agents), the agent can leave this system, and it can resume its execution on the next interoperable system.*

For complex systems it is prudent to define a concise set of interfaces and protocols where agents and their hosting systems come into contact. With regard to systems for mobile agents these contact points include but are not limited to the following:

Communication: message transport and communication language

Mobility: agent transport protocols, agent encoding

Security: agent authentication and state appraisal

General: *agent setup and lifecycle, system interfaces*

At the time of writing, we are aware of only one attempt to provide means of interoperability among systems of mobile agents, which is the MASIF proposal [10]. FIPA [2] is also active in the standardization of agent mobility [8] issues, but this particular thread of FIPA's work focuses on a high level of abstraction, and, to the best of our knowledge, the document did not have much public scrutiny yet.

It is therefore fair to say that these existing standardization efforts have not yet shown to be *effective* to provide actual interoperability among systems for mobile agents with regard to all but the first category in the table given above. This lack of effectiveness also impairs the effectiveness of agent communication standards when applied to mobile agents because delivery of messages to mobile agents requires a system dependent interface.

Rather than following the top down approach to interoperability by means of standards, we chose to take a bottom up approach based on voluntary interoperability with other systems for mobile agents. Hence, we designed our own system SeMoA [13] in a way that, we hoped, would facilitate the task to provide true interoperability with other agent systems. Our goal was to let agents of two well-known systems run in our own system unmodified. We chose the Jade system and the Tracy system as targets. For simplicity, we speak of the *acme* system whenever we refer to a system other than our own system e.g., Jade [3] and Tracy [6].

The lessons we learned so far fall into two broad categories. First, we gained insight into design patterns that help to build agent systems in a way that facilitate provision of interoperability. Second, we gained insight in designs that inhibit provision of interoperability. In this paper, we wish to share our experience, and report the results of our practical experiments. We concentrate on agent setup, lifecycle, and system interfaces.

Section 2 gives an overview over the way mobile agents are set up in our own mobile agent server SeMoA, and how these agents access server facilities such as mobility. In Sect. 3 we present three case studies of varying level of progress, targeted at providing interoperability with the agent systems *Jade, Tracy,* and *Aglets.* [6, 3, 1]. Our brief conclusions are summarized in Sect. 4.

2. Agent Setup and Lifecycles

SeMoA takes a rather paranoid stance when it comes to setting up agents. A mobile agent is transported by means of a *Java Archive* (JAR), which contains the serialized state of the agent among other data. Before the agent is unmarshalled and one of its classes is linked into the server's JVM, the archive's contents have to pass a configurable pipeline of security filters. The filters we provide support agent authentication and integrity checks by means of digital signatures, selected revealing of data with detection of protocol intervleaving attacks [11], bytecode filtering, and more.

Once the agent is admitted to the server, a thread group and class loader are created for this agent. A launcher thread is spawned in this thread group, which takes care of unmarshalling the agent and later on becomes the first thread of the agent. Although a malicious class of the agent may abuse callbacks in the *Java Serialization Framework* to seize control of the thread in which the unmarshalling takes place, this does not give the agent access beyond what it would have been granted anyway.

Agents request migration by setting a ticket that points to the desired destination. However, the server transports the agent only after all threads in the agent's thread group have terminated. Again, we verify that serialization callbacks were not used to spawn new threads. The invariant we enforce in this way is that, at the time of agent transport, no class of that agent is on any thread's stack frame any more (unless the agent successfully attacked the systems e.g., it hijacks the garbage collector thread by means of sneaking a malicious implementation of `finalize()` around our byte code filters).

When the agent instance is unmarshalled, its class name is compared to one of several agent property strings that are signed along with the agent's static part by the agent's owner. In conjunction with class signing, this prevents adversaries from substituting the principal agent class with another runnable class that might be included in the agent.

The properties definition consists of a text file with key/value pairs where the key is separated from the value by an equals sign. Two more properties are mandatory. The *agent system type* property identifies the type of agent system on which the agent was created e.g., *Jade, Tracy, Aglets*, or SeMoA. The *agent type* property identifies the system--specific agent type in case multiple types are supported by that system e.g., *Java* for agents that are programmed in Java, and *shell* for agents that are programmed in a shell script language.

Based on a given agent system type and agent type, SeMoA's *lifecycle registry* is queried for a matching *lifecycle implementation.* In other words, the lifecycle registry is a factory [9] for the generation of *lifecycle instances* that handle the agent's actual lifecycle. The freshly generated lifecycle instance wraps around the agent instance and translates between SeMoA's agent lifecycle and the lifecycle of the agent's native system. In particular, it instantiates all necessary components that make the agent instance believe that it is running on its native system. Furthermore, the lifecycle instance is responsible for the marshalling and unmarshalling of the agent instance, thus completely decoupling the representation of agents from SeMoA's core.

Lifecycles are defined in terms of a deliberately simple interface declaration with the methods

```
start()
stop()
suspend()
resume(ErrorCode err)
```

Agents resume their execution in the case of e.g., a failed suspension, on migration errors, or at the end of their suspension. If applicable, an error code is passed in order to describe the reason why the execution is resumed.

Subsequent to setting up the lifecycle, the principal agent thread is annotated with four facilities by means of subclasses of `InheritableThreadLocal`:

Mobility context: provides methods to set destination tickets, retrieve the agent's name, and access more agent-specific data.

Communication context: provides methods for sending and receiving messages.

Environment: provides dictionary operations on a shared space of objects, with a hierarchical name space for the keys. All operations are subject to access control, and published objects may be wrapped into proxys that implement varying degrees of separation between callers and called objects.

Variables context: provides read access to an agent's properties, taken from the agent's archive.

These annotations are inherited by all threads that are spawned subsequently from the annotated thread. Hence, they are available to all threads of an agent. Access to the annotated facilities is granted based on an agent-specific *tag permission*. This is a permission that is unique for each agent instance in the server, and is assigned only to the classes of that agent and its initial access control context. This prevents threads of one agent from accessing facilities that are assigned to another agent's threads in the case of a direct inter-agent method invokation.

The bottom line of this is that we need not rely on a special agent class in order to make initial system hooks available to the agent, as is common in contemporary mobile agent systems. By default, native SeMoA agents must implement only the `Runnable` interface, although any other interface or class could be supported easily as well. Access to primitives such as migration and communication are provided solely by means of thread annotations. These approaches, the one based on an abstract agent class and the one we took, are juxtaposed in Figs. 1 and 3. From the perspective of interoperability, it is generally advantageous to use interfaces as types rather than abstract agent classes, because this allows an agent class to maintain type-compatibility with multiple systems simultaneously.

For instance, we implemented a lifecycle factory for Jade agents. As a side effect, programmers may write agents based on the abstract Jade agent class, use Jade behaviors, and still access SeMoA's facilities. Such Jade agents are instantly mobile, and benefit transparently from SeMoA's migration and security mechanisms.

3. Case Studies

The first step when integrating support for the *acme* system is, of course, a thorough analysis of that system's architecture and agent lifecycle. Ideally, the *acme* system is available in source code, with appropriate documentation. To some degree, reverse engineering tools are helpful, in particular those that can generate UML diagrams from Java byte code.

The primary goal of this phase is to distinguish agent support from its concrete implementation. Typically, the analysis starts at the system's abstract agent class. All places must be identified, where this class is invoked. Analysis of these places reveals salient details of the system's agent lifecycle. Special attention must be given to thread handling, and subtle assumptions that are relevant for the agent's functioning. Often, such details or not excessively documented, or the documentation abstracts from the particularities of the implementation.

Whenever a method of the abstract agent class is invoked, its parameters and return values must be analyzed for non-trivial types. Ideally, these parameters are of the following types:

- Interface classes; the use of interfaces indicates that the developers of the *acme* system anticipated alternative implementations of the system's functionality.
- Isolated helper classes without references to other *acme* classes; these classes can often be reused without modification.
- Classes that resemble entry points to self-contained subsystems that need no special adaption and can be used as a whole (in other words, modules that can be treated as a black box).
- Java standard classes

Jade's communication package is a positive example of a self-contained subsystem that can be adapted easily. Where parameters do not fall into one of the aforementioned categories, the situation becomes complicated. The analysis must recurse for these classes, and in the end a decision must be taken to the effect whether the integration is feasible and worth the effort.

The next phase deals with the mechanisms used by agents to access facilities such as migration and communication. The fewer and the more concise these mechanisms are, the easier is it to emulate the *acme* system. Systems that clearly define a limited set of interfaces to this purpose are easier to interoperate with than systems that have dependencies scattered all over the implementation.

Adaption of agent communication has the specific problem of addressing peer agents correctly. This is less troublesome for *acme* agents that are created on a SeMoA server. Many agent systems use naming schemes based on the *Uniform Resource Locator* [4] syntax, for instance something like `wombat@gwork.org:40000/strangeplace`, where "wombat" is a name that can be chosen freely by the agent's creator. However, SeMoA allows no free choice of an agent's name, instead an agent's name is computed implictly from a digital signature of its static part (see [14] for details). Implicit names consist of SHA-1 [7] digests, hence are 20 bytes long. For ease of reading, we give only 8 hexadecimal nibbles in our examples, rather than the whole 40. If the agent is created on a SeMoA server, and its implicit name is computed as `f42a1cc0` then the agent can be given the name `f42a1cc0@gwork.org:40000/strangeplace` in order to match *acme's* syntax.

If the agent has its origin elsewhere, and is assigned a human readable name such as "wombat", then a suitable mapping mechanism must be used by the lifecycle implementation in order to translate back and forth between these names as required.

Our experience up to the time of writing shows that agent communication is less of a problem when compared

to agent migration, though. Migration is often more tightly interwoven in a system's design and implementation. For instance, SeMoA's security policy requires that migration is initiated only after all threads of the migrating agent have terminated, a fact that is hardly taken into consideration by programmers of *acme* agents. However, we do not wish to sacrifice our security policy to interoperability. Termination before migration prevents agents from repeatedly spawning copies, and refusing to terminate afterwards, thus effectively flooding a network of agent servers. It is worth noting that the *full mobility protocol* defined by FIPA enables this type of attack by virtue of its specification, and requires that at least the problem of asynchronous thread termination is solved satisfactory. Consequently, a lifecycle implementation might have to defer execution of a `go()` statement to the point where it got rid of stale threads that were spawned in the agent's thread group either by the agent itself or as a consequence of a call to, for instance, Java's *Abstract Window Toolkit* (AWT).

In summary, the *acme* system is probably straightforward to adapt if it:

- is available in source code, and well documented (no surprise here);
- confines the dependencies between agents and the system to a clear and well-defined set of interfaces;
- has a modular design, and anticipates alternative implementations for its modules;
- models facilities that are required by mobile agents separately;
- provide features and services in the form of agents rather than specialized classes which must be adapted or treated in special ways;
- pays attention to security.

To some degree, security has similar requirements as interoperability – in both cases there shouldn't be too many drawbridges that lead into and out of your fortress, because you have to put guards in front of each.

3.1. Jade

We used Jade Version 2.01 beta as the basis of our experiments with Jade. Jade has a focus on agent communication and cooperation rather than mobility. Consequently, the communication support is well developed whereas migration has only marginal support. Our goal was to run Jade agents in SeMoA, without recompilation, and in a way that allows Jade agents to communicate with other Jade agents, where the peer agent can be either at the same server or at a remote server.

The adpation of Jade was surprisingly straightforward, and did not cause major problems. Jade agents are initialized with a so-called `AgentToolkit` implementation that functions, from an agent's point of view, as the principal hook into the agent system. `AgentToolkit` is actually an interface; the `JadeLifecycle` we developed for SeMoA implements this interface, and mediates between SeMoA and the Jade agent. All mappings could be handled in the `JadeLifecycle`. Figure 2 shows an UML diagram of the classes involved. Bold class names denote classes of SeMoA, all other classes were taken from Jade.

Jade supports scheduled behaviors. This allows agents to periodically repeat a specific action, or be invoked at particular times. This requires managing a global timer and dispatcher thread, which is a responsibility of the `AgentToolkit` implementation, and posed no difficulty.

Communication in Jade bases on CORBA, and is well separated in a self-contained package. Our `JadeLifecycle` reuses this package. At boot time, SeMoA activates a message stub that is responsible both for dispatching incoming messages as well as relaying outgoing messages. The stub also takes care of translating from internal to external addresses and vice versa. Our tests confirmed that intra- and inter-platform communication between Jade agents works fine. Furthermore, inter-platform also works fine between agents at Jade servers and SeMoA servers with Jade support.

There is hardly any criticism we could raise on Jade's design with regard to our aims. A minor nuisance was caused by some classes that were declared as `protected` or package private without obvious reason, among them `AgentToolkit`. We changed the access modifiers to `public` and recompiled these classes. Apart from this, we had to make no changes.

3.2. Tracy

The integration of Tracy has been done based on the current version (which is 0.54 *alpha*). At time of writing, Tracy agents are able to run, communicate, and migrate in a network of SeMoA servers.

The architecture of the Tracy adapter classes is similar to Jade. Figure 5 illustrates the design of our `TracyLifecycle` and related classes as an UML diagram. Again, bold class names denote classes of SeMoA. The diagram shows only a subset of the classes we developed in order to support Tracy in SeMoA. Tracy distinguishes between *mobile agents* and *system agents*. The first ones are able to migrate, whereas the latter have special privileges e.g., to open a graphical user interface. Both inherit from the abstract base class `Agent`, which we access from the lifecycle class in order to control the agent.

The implementation of the inter-agent communication mechanism was straight forward. Tracy agents communicate by means of a *blackboard*. The blackboard acts as a hierarchical name space where agents can deposit objects. We wrapped a single `Blackboard` instance in a SeMoA service and published it at boot time in SeMoA's shared object environment. All instances of `TracyLifecycle` access this single blackboard service whenever an agent wants to read or write messages.

Tracy also uses an intra-agent communication mechanism which allows an agent to send messages to itself. This is used e.g., to control the agent's state of execution. In order to adapt this behavior, the lifecycle registers itself as listener of the agent's message queue. Consequently, the lifecycle is able to intercept and process all messages of that particular agent instance.

More difficult than the implementation of communication support was to adapt Tracy's migration concept. Whenever a Tracy agent wants to migrate, it throws a `WantToMigrate` exception. By assumption, this type of exception may not be caught by the agent. Instead, it is caught by a server thread which processes the agent's request. In order to support this behavior, the `run` method of `TracyLifecycle` is implemented as a loop that catches the exception and waits until all threads of the agent terminated. In accordance with SeMoA's security policy, an agent has to throw the exception and terminate all spawned threads as well. This also holds for the `WantToDie` exception, which is used by Tracy, and which indicates that the agent wants to terminate.

The adaption of the migration process itself was straightforward. SeMoA's architeture supports different transport mechanisms, which are distinguished by means of the protocol identifier of the target URL. Since Tracy agents always use the protocol `tracy` for migration, we simply published a handler instance for this protocol. This handler actually uses a simple socket connection for transport. This allows Tracy agents to migrate between different SeMoA hosts. We intend to provide a protocol handler for the real Tracy migration protocol as well, which is developed in cooperation with the Tracy authors. Tracy's transport layer is currently redesigned [5], and we wish to provide interoperability with the most recent version.

3.3. Aglets

Our experiments with Aglets were based on the Aglets Software Development Kit Version 1.1 Beta. At first sight, the Aglets framework has a highly modular design. Major parts of the Aglets framework are modeled by means of interface classes and abstract classes (package `com.ibm.aglet`). This approach yields a generic system structure that accounts for alternative implementations of the core system's functionality.

Concrete implementations of the abstract and interface classes are found in package `com.ibm.aglets`. Unfortunately, our code inspection revealed that frequently the types of parameters were based on the concrete implementations rather than the interfaces and abstract classes defined in `com.ibm.aglet`. For instance, the code referred to `LocalAgletRef`, a concrete implementation of the abstract class `AgletStub`, where a reference to `AgletStub` would have been more appropriate. On other occasions, the code reads

```
void setMessageManager(MessageManagerImpl)
```

rather than

```
void setMessageManager(MessageManager)
```

as one would have expected. This turned out to be a major setback for our task. The reason appeared to be added functionality that couldn't be accessed properly by means of the interface types. Using the concrete implementations as parameter types worked around these limitations, although this is generally not a desirable approach.

Figure 4 illustrates the design of the `AgletLifecycle` and related classes as an UML diagram. Class names that are typeset in bold denote classes of SeMoA. Only a subset of the classes we developed in order to support Aglets in SeMoA are shown. The border line between SeMoA and Aglets runs along the interfaces `AgletStub`, `AgletContext`, and `AgletProxy`. The base class `Aglet` makes use of the two classes `AgletID` and `AgletInfo`, which are actually just helper classes that could be reused without modification.

The most difficult part posed the implementation of `AgletProxyImpl`. Its corresponding implementation in the Aglets system turned out to refer to a considerable number of classes from package `com.ibm.workbench`. These classes were so specific that reusing them did not appear to be a viable solution. This complicated the implementation of `AgletProxy`, which is involved in agent migration, communication, and remote control.

At the time of writing, we can run Aglets in SeMoA, and these Aglets can dispatch messages to themselves. However, we cannot migrate Aglets at this point. This is subject of future work.

4. Conclusions

In this paper, we presented a bottom-up approach towards interoperability of mobile agent systems, which is based on voluntary interoperability between selected agent systems, rather than a top-down approach driven by standards, which are not available in the first place.

In particular, we presented a number of design approaches that facilitate the transparent support of agents of other systems in our own mobile agent server SeMoA. One of the key features of our design is the modeling of agent lifecycles by means of specialized lifecycle implementations that translate between a native lifecycle and the lifecycles of adapted systems. Lifecycle implementations have the task to make agents believe that they are running in a native environment although they actually do not.

In the course of pursuing interoperability between different mobile agent systems we gained considerable insight both in the particularities of Java as well as in the do's and do not's of mobile agent system design.

At the time of writing, our work is far from complete, yet we can already demonstrate a successful integration of Jade and Tracy agents, as well as initial results from our efforts to provide interoperability with Aglets. Our future work will address migration protocols of the *acme* system, so that agents are able to migrate back and forth to and from SeMoA and *acme* servers.

References

[1] Aglets Software Development Kit. Software project at SourceForge, Internet resource at URL `http://sourceforge.net/projects/aglets/`.

[2] Foundation for Intelligent Physical Agents (FIPA). Internet Web page at URL `http://www.fipa.org`. Version current on November, 2001.

[3] F. Bellifemine, A. Poggi, and G. Rimassa. Jade programmers guide, June 2000. Available at URL `http://sharon.cselt.it/projects/jade`.

[4] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). Request for Comments 1738, Internet Engineering Task Force, December 1994.

[5] P. Braun and J. Eismann. Personal communication, 2001.

[6] P. Braun, C. Erfurth, and W. R. Rossak. An introduction to the Tracy mobile agent system. Technical Report No. 2000/24, Friedrich Schiller University of Jena, Computer Science Department, September 2000. Available at URL `ftp://ftp.minet.uni-jena.de/ips/braun/bericht-00-24.pdf`.

[7] FIPS180–1. Secure Hash Standard. Federal Information Processing Standards Publication 180–1, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, April 1995. supersedes FIPS 180:1993.

[8] Foundation for Intelligent Physical Agents. *FIPA Agent Management Support for Mobility Specification*, August 2001. Document PC00087B.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns*. Addison Wesley Longman Publishing Co., December 1994. ISBN 0201633612.

[10] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Omo, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF – The OMG Mobile Agent System Interoperability Facility. In K. Rothermel and F. Hohl, editors, *Proceedings of the Second International Workshop on Mobile Agents (MA '98)*, volume 1477 of *Lecture Notes in Computer Science*, pages 50–67. Springer Verlag, Berlin Heidelberg, September 1998. The MASIF specification is available from URL: `http://www.fokus.gmd.de/research/cc/ecco/masif/doc/97-10-05.pdf`.

[11] V. Roth. On the robustness of some cryptographic protocols for mobile agent protection. In *Proc. Mobile Agents 2001*, volume 2240 of *Lecture Notes in Computer Science*. Springer Verlag, December 2001. Revised version of [12].

[12] V. Roth. Programming Satan's agents. In *1st International Workshop on Secure Mobile Multi-Agent Systems*, Montreal, Canada, 2001.

[13] V. Roth and M. Jalali. Concepts and architecture of a security-centric mobile agent server. In *Proc. Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, pages 435–442, Dallas, Texas, U.S.A., March 2001. IEEE Computer Society. ISBN 0-7695-1065-5.

[14] V. Roth and J. Peters. A scalable and secure global tracking service for mobile agents. In *Proc. Mobile Agents 2001*, volume 2240 of *Lecture Notes in Computer Science*. Springer Verlag, December 2001.

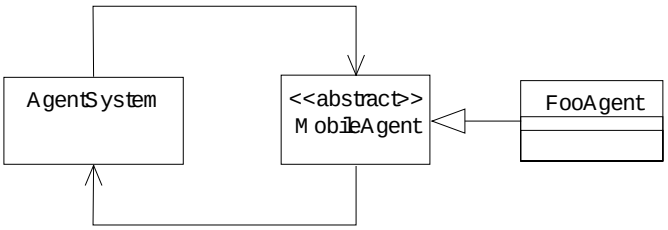

Figure 1. Agents extend a well-known abstract class that provides the basic hooks into the system.

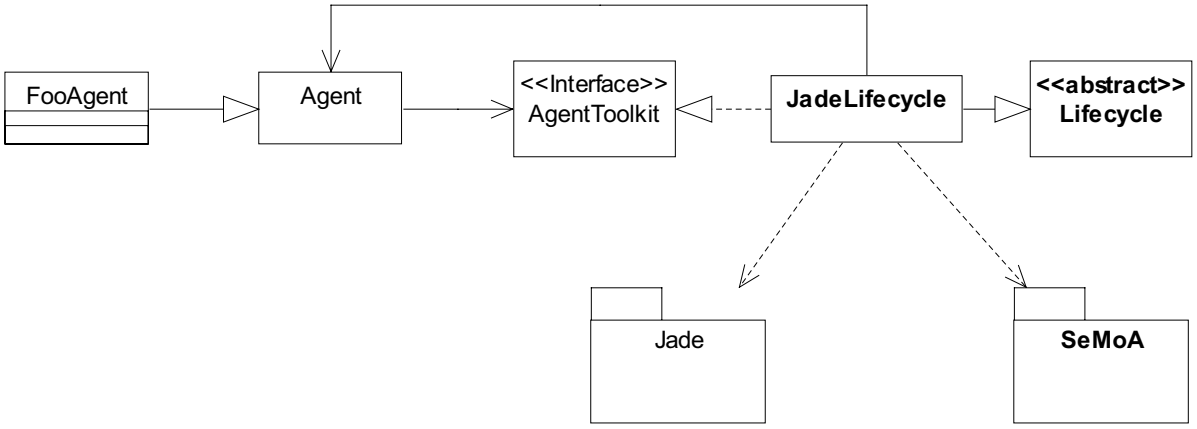

Figure 2. UML class diagram showing an implementation of the lifecycle pattern for interoperability with Jade agents.

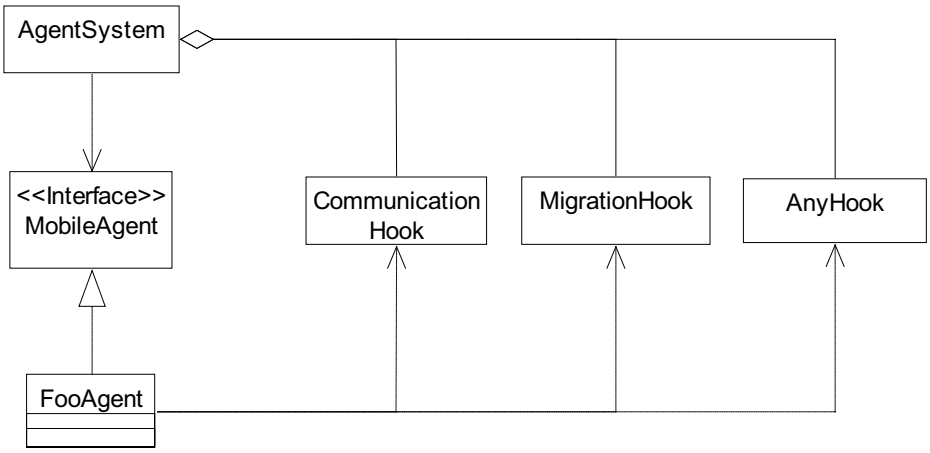

Figure 3. Agents implement an interface as the primary type. Hooks into the hosting system are modelled as separate facilities. In the case of SeMoA**, an agent's threads are annotated with facilities such as agent mobility, communication, and access to shared object instances.**

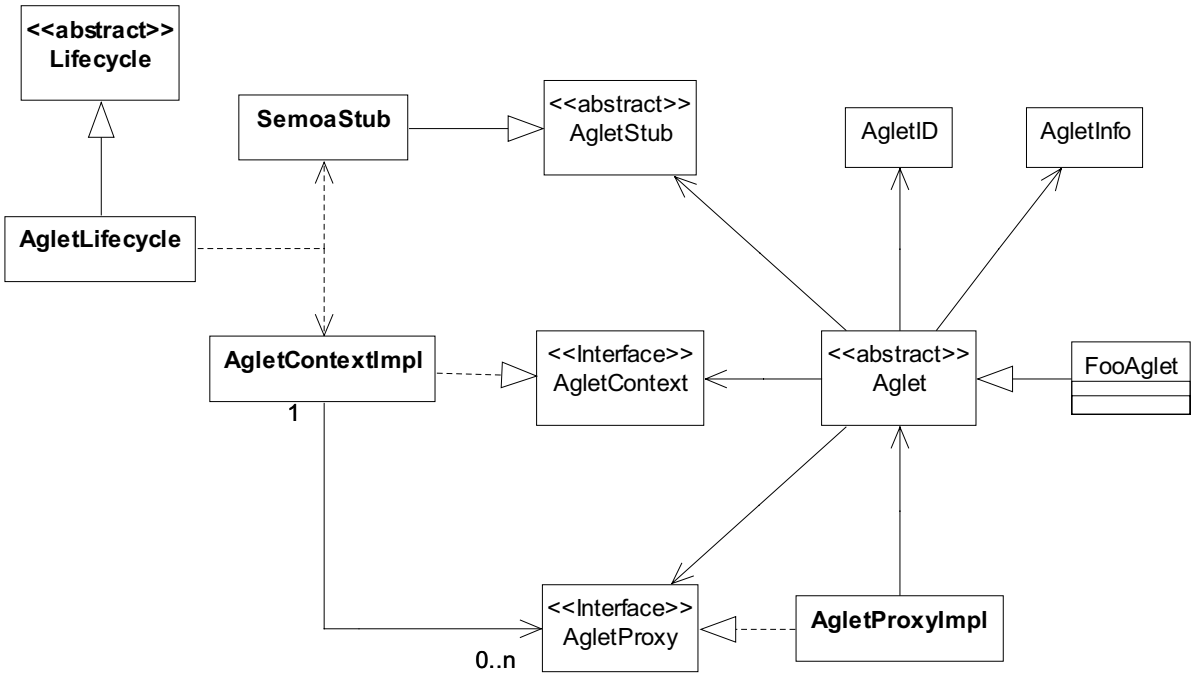

Figure 4. The UML class diagram shows an implementation of the lifecycle pattern for the interoperability with Aglets.

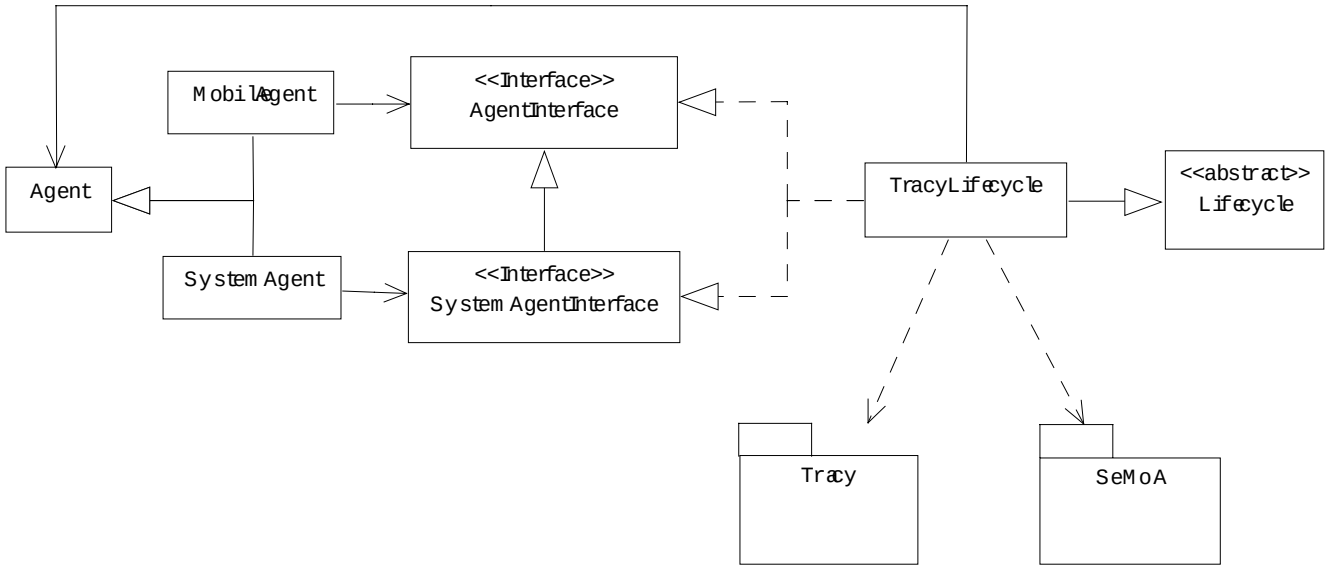

Figure 5. UML class diagram showing an implementation of the lifecycle pattern for interoperability with Tracy agents.