Tracy: An Extensible Plugin-Oriented Software Architecture for Mobile Agent Toolkits

Peter Braun, Ingo Müller, Tino Schlegel, Steffen Kern,
Volkmar Schau and Wilhelm Rossak

Abstract. In this chapter we propose a software architecture for mobile agent toolkits and describe our Tracy toolkit as a reference implementation of this architecture. Agent toolkits mainly consist of a software system that forms an agency, which is responsible to host mobile and stationary software agents. In contrast to most architectures developed so far, which already define a large set of services for agent migration, communication, and security, we propose to employ a kernel-based approach. The kernel only provides fundamental concepts common to all agent toolkits and abstracts from any of these services. In particular, although Tracy was developed as a mobile agent toolkit, its kernel abstracts from all issues related to agent mobility, delegating this to an optional service implementation. This makes it possible to replace Tracy's migration service with another implementation and even to have two different migration services in parallel. Service implementations are developed as plugins that can be started and stopped during run-time. We have already developed almost a dozen plugins for agent migration, communication, authentication and authorization, and security solutions, only to name a few. We believe that this architecture is a useful foundation for research on agent-related topics as it allows research groups to implement their own results as a service which can be used by other groups running an agent system based on the same architecture.

1. Introduction

Mobile agents have been introduced as a design paradigm for distributed applications [35]. A mobile agent is a program that can migrate from host to host in a network of heterogeneous computer systems and fulfill a task specified by its owner. It works autonomously and can communicate with other agents and host systems. During the self-initiated migration, the agent carries its code and some kind of execution state with it. What comprises the execution state depends on the

underlying programming language and is, in the case of most Java-based toolkits for example only the serialized agent (an agent is an object of a specific class) and does not contain information about the state of the Java virtual machine. On each host they visit, mobile agents need a special software that we name *agency*, which is responsible to execute agents, provides a safe execution environment, and offers several services for agents residing on this host. A *mobile agent system* is the set of all agencies together with agents running on these agencies as part of an agent-based application. To refer to a specific project or product, for example Aglets [23] or Grasshopper [3], we use the notion *agent toolkit*.

For some years, mobile agents have been a hot topic in the domain of distributed systems. Reasons were problems more traditionally designed distributed systems, especially client/server systems, might have to handle work-load, the trend to open large numbers of customers direct access to services and goods, and user mobility. Mobile agent technology can help to design innovative solutions in these domains by complementing other approaches, simply by adding mobility of code, machine based intelligence, and improved network- and data-management capabilities. We have seen tremendous research effort, for example, in the area of mobile agent security, where sophisticated security protocols were developed to solve the problem of malicious agencies (that try to attack visiting agents) and malicious agents (that try to attack hosting agencies) [17]. Other research topics are, for example, performance aspects of mobile agents [6], in which inherent drawbacks of mobile agents are tackled using sophisticated migration strategies, mobile agent communication [2], or control issues [1], which targets at the development of algorithms to trace mobile agents while roaming the Internet.

However, the interest in mobile agents as a design paradigm for distributed systems seems to have dwindled over the last years. The number of research groups working on mobile agent related topics is becoming smaller, some conferences and workshops cease to exist or are aligned with more general topics regarding mobility of code or mobility of devices. It is argued that mobile agents were not able to satisfy some of the main expectations, for example regarding their ability to reduce network traffic overhead. Vigna [34] states that mobile agents are very expensive and provide worse performance in the general case than other design paradigms, as for example remote procedure call or remote evaluation. Compare Vigna's early work regarding network load of different design paradigms [33] and a discussion of his approach in [6]. He also points to security problems, which are unlikely to be solved completely and will, therefore, impede the acceptance of mobile agents. Other authors try to get to the bottom of the problem of decreasing acceptance and discuss whether mistakes of the research community have caused the current disappointing situation. For example, Roth [29] questions Java to be the best programming language to cope with unresolved security problems. Johansen [19] points out that far too many groups have focused on the development of yet another prototype of a mobile agent toolkit with only small contributions to the fundamental research questions. In fact, the current situation is characterized by a few tens toolkits. Although this number reflects an enormous research output

by different groups all over the world, it also reveals premature status of research and a not-existent coordination between projects.

We agree with Johansen about the high number of research prototypes and their negative effect to the research community. However, many research groups were obliged to develop their own prototype because of the lack of any reference architecture for mobile agent toolkits as well as the absence of an open and extendable implementation of a mobile agent toolkit. Therefore, each research group working on core research problems rather than application development was compelled to develop its own prototype and due to the high complexity and limited resources this prototype is more a proof-of-concept implementation focusing on a single research issue and leaving out elementary functional components necessary for a full mobile agent toolkit. Only to name two examples, we mention Semoa [30] as a system with a very strong focus on security issues and our first implementation of Tracy [7] which was designed around early ideas regarding high-performance migration protocols. At first, agent communication was not in the focus of neither of these systems–but added later in both of them. Other groups focused on agent tracing and communication problems, for example [25], but their algorithms are not adopted in other toolkits so far. We see these isolated islands of research as another obstacle for the acceptance of mobile agents as there is no single mobile agent toolkit available that provides at least an almost up-to-date set of features and research results.

To amend this situation, one of the most important challenges of our Tracy project is to develop a reference architecture for mobile agent toolkits. This architecture is leveraging off of previous work done by the Tracy team in designing the first Tracy architecture [7] and benefits from experiences learned when porting it to mobile platforms and investigating feasibility to use Tracy within an electronic commerce application [22]. Our model for agencies consists of a very small kernel which defines only imperative functions of an agency and the concept as well as the life-cycle of agents. As part of our model, we define the concept of services, which implement additional functionality on top of the kernel. The model does not define anything related to specific services, for example agent communication, agent migration, or agent tracing, except of an interface for services to communicate to the kernel.

The Tracy toolkit, which is the reference implementation of our new agency model, has a plugin-oriented software architecture. Each service in the meaning of our model is implemented as a plugin, which can be dynamically started and stopped at runtime. As part of the Tracy implementation, we have developed more than a dozen plugins for various services. An interesting result regarding mobile agent toolkits was that it is possible to design an agency without considering mobility of agents–and later to plugin this new service without any modification of the kernel.

The remainder of this book chapter is structured as follows: We start by giving a state-of-the-art overview of other approaches to build interoperable and component-oriented mobile agent toolkits. After that we introduce our agency

model and describe the Tracy mobile agent toolkit, which is a reference implementation of our model. We will also introduce some of the most important software components that come along with Tracy already. Finally, we describe our experience with our agency model and the Tracy toolkit in building a mobile agent based application for mobile users.

2. Related Work

In this section, we will give a concise overview of the development of software architectures of mobile agent toolkits in the past few years. We choose this presentation style to demonstrate our kernel-based software architecture as the evolution of current mobile agent toolkit design.

In our opinion, the timely development of mobile agent toolkits can be roughly distinguished into an early, a current, and a next generation phase. The early phase from the mid 1990s till the end of the 1990s is characterized by the development of first complex mobile agent toolkits, such as AgentTcl [15], Aglets [23], and Grasshopper [3]. The main goal of these systems was to provide a basic set of functions and features for the development of mobile agent based applications. Therefore, early toolkits offer only a small but inflexible interface via a mobile agent class, which encapsulated access to the functionality of the toolkit. It is difficult to add (e.g. a tracking mechanism) or to change functionality (e.g. replace the existing mobility or communication model), or to adapt the toolkit to specific requirements (e.g. to downsize it for resource-limited mobile devices).

After the mobile agent community recognized the drawbacks from the early mobile agent toolkits, the current phase begun at the end of the 1990s [18,30]. Now mobile agent toolkits are developed using a component-oriented architecture. Most systems still use a layered architecture, with a migration layer and communication layer below, a layer for agent management functionality in the middle, and a basic service layer on top [5, 20]. Each layer consists of a set of components, which implement the layer's functionality. Following this, the problems of first mobile agent toolkits with respect to flexibility, extensibility, and adaptability were improved. However, only for each mobile agent toolkit itself. For, the problem of exchanging functionality and interaction between different mobile agent toolkits still remains unresolved. The main reason for this problem is a high coupling between internal components and different definitions of component interfaces in these mobile agent toolkits.

Some mobile agent toolkits even implement standards to amend the interoperability problems, such as MASIF [24] or FIPA [26], originally developed for multi agent systems. Both standards do not solve the given problems in a suitable manner. MASIF has not been accepted because it relies too much on further OMG specifications such as CORBA and IDL, increasing the effort for the development of a mobile agent toolkit with aspects not belonging to the basic principles of mobile agents. FIPA, implemented for example by Jade [4], is also too complex because

it defines a large set of system internal architectural and design constraints, for example regarding agent communication. This is a problem especially with respect to researchers who have their main focus on a single problem in the mobile agent domain not willing to implement all mandatory parts of the specification.

Thus, we are still looking for an approach that might solve the problems identified in the introductory section. This is the reason why we believe that mobile agent toolkits should enter a next generation phase in which the focus changes from trying to uniform system design to specify only a very small set of kernel functionality accessed through a lean interface, reducing the coupling between system components. Functionality can be added in a very flexible way with software components.

3. JAM – A New Model for Agencies

In this section we will introduce *JAM* (Java Agency Model), which is a new Java-based model for agencies for mobile and intelligent software agents. Our goal is to define a model that consists of the smallest number of interfaces and classes and only defines an imperative set of functional and non-functional requirements in order to execute software agents. In the following, we restrict ourselves to Java-based mobile agent toolkits, because Java has become the de-facto standard for programming mobile agents; almost all new toolkits developed in the last six years are programmed in Java. The reasons for this are many built-in functions, for example object serialization, dynamic class loading, and the sand-box security mechanism, which simplify the development of mobile agent toolkits. The restriction to Java makes sense, since we focus on the exchangeability of service components and the exchangeability of software agents on the level of implementations. Therefore, our model must be seen orthogonally to other models and standards, for example FIPA [26] and MASIF [24], which specify communication protocols and migration protocols, respectively.

Current models for agencies and collections of design issues for mobile agent toolkits [16, 21], already include design decisions for several high-level *services*. This notion will be used in the rest of this book chapter as synonym for optional functions offered by an agency and used by agents in order to fulfill their task. Services are for example migration, communication, agent tracking, persistency, region management, etc. We see as major drawback of these current agency model that they create many dependences between actually independent services. The resulting agency implementation is more monolithic, because services are not represented by independent software components. In fact, our model abstracts from these services and in contrast attempts to move as many functional requirements of a mobile agent toolkit into optional services.

As motivated in the introduction, we want to address the possibility to exchange research results between research projects in form of software components and software agents. Therefore, we propose to split an agency into a single *kernel*

(compare [11] for an introduction to kernel-based software architectures) and several optional *service components*, each implementing a single *service*. The kernel only provides the basic environment for running agents, maintaining a directory of agents currently residing at this agency, defines and monitors an agent's life-cycle and defines a basic interaction model for agents and services. It might even be argued that maintaining an agent directory should not be part of the kernel but provided as a service. However, the kernel must maintain such a directory in any way, because it has to wait cooperatively for all agents to terminate, if the agency shuts down.

Upon that kernel, each software component provides an additional service that extends the functionality of the agency. The resulting model for an agency has, therefore, a layered design, where the kernel can be seen as lowest layer. On top of the kernel, all service components form the middle layer. Finally, agents are executed as part of the agency and form the topmost layer, where they can only access services but not the kernel anymore. Actually, it must be stressed that our model does not distinguish between mobile and stationary software agents. As a consequence, our system could be seen to be more a multi-agent toolkit instead of a mobile agent toolkit.

In the rest of this section we will define the basic concepts and functions of a kernel and we will define interfaces for services to communicate to the kernel. Finally, we will also explain how agents can communicate to services. We start with the description of the basic abstractions that form an agency, i.e. *kernel*, *services*, and *agents*. For the following, compare Fig. 1.

3.1. Kernel

We assume that every agency has a name, which comprises of a *logical agency name* and the full qualified domain name of the underlying host. For example, if the logical agency name is *goldeneye* and the host name is *fleming.cs.uni-jena.de*, then the *full agency name* is *goldeneye.fleming.cs.uni-jena.de*. The name of an agency must not change during runtime. It must be allowed to start multiple agencies on a single host but the model does not define how this is done in practice.

The kernel consists of two classes. Class *Kernel* is responsible to maintain a directory of all agents currently residing at this agency and all services currently connected to the kernel. Class *Context* is used as mediator for agents to access the agency and services (compare Sec. 3.2 and Sec. 3.3 for more information). The kernel retains several information about agents, for example name, time of creation, permissions, and owner.

The kernel is responsible to start and stop agents and to control an agent's life-cycle (which is explained later). The kernel is also responsible to generate agent names, which must be globally unique in the whole agent system. The model does not define *how* agent names are computed but enforces that they can be represented as *String* objects. The kernel must create a new class loader for each agent to ensure that agents cannot access each other via direct method calls and to assign each agent an individual set of permissions. In Java terms this is named

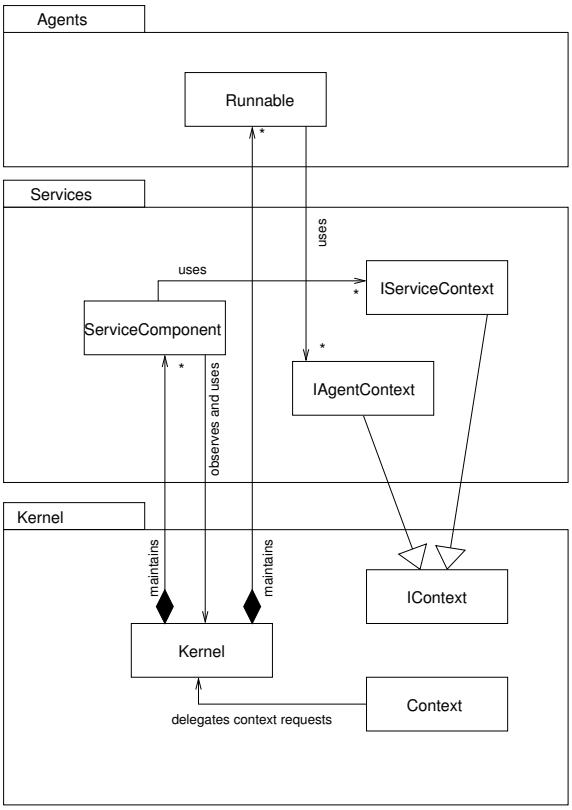

FIGURE 1. Design of our agency model as UML class diagram. For sake of simplicity, we omit methods and stereotypes.

a *sandbox*. Each agent is assigned an own thread of control when it is executed, which must be placed in an own thread group in order to prevent illicit access of agents to each other.

3.2. Services

On top of the kernel there are several *service components* that can be plugged into an agency. Each service component provides a specific service (which is identified by a *service name*) and extends the functionality of the agency. It bears a unique *name*, which is used to unambiguously identify a service component, if necessary. Usually, agents and other service components only use the *service name* to select a single component that provides this service. Assume for example the case, that an agency provides two migration components, both accessible under service name *migration*. To disambiguate components with the same service name, it is also allowed to address a service by its *full service name*, which is *name.servicename*.

When a service component is started, it is loaded using a new class loader in order to separate it from other components and agents. Note that the model does not define *when* a service component is started. It is possible to launch an agency with a predefined set of services or to allow to start dynamically on demand. Each service component obtains a reference to the kernel.

Service components can start and stop agents (simply by calling the corresponding kernel methods), request a list of all agents currently residing at this agency and a list of all services and service components. Our agency model also defines the reverse communication direction using the observer pattern. A service component can register with the kernel to become notified in case of specific events. A description of the most important events follows in the next section.

Our agency model defines a loose coupling between services components among each other and between agents and services. Direct method calls are prohibited and replaced by so-called *context* objects, comparable to proxies. Each service component has to provide two interfaces: an *agent context interface* and a *service context interface*. The first interface must extend interface *IAgentContext* (defined as part of the kernel package) and defines methods of this service to be used by agents, whereas the second interface must extend interface *IServiceContext* (part of the kernel package) and defines methods to be used by other service components. For example, a service for agent migration defines an agent context interface with methods to set the migration destination and a communication service defines an agent context interface with methods to post a message. The communication service also enables other service components to send messages to agents and, therefore, defines a service context interface, which includes a method to send a message.

The main advantage of this concept is that agents and service components do not hold strong references to another service component, which makes it possible to invalidate a context object, if a client must not use this service any more and to even exchange a service during runtime without giving notice to the agent or other services. In contrast to other agency models, we propose to use method calls rather than asynchronous messages as convenient means of communication between service components as well as between agents and service components. Our agency model is open to enable communication via asynchronous messages using an appropriate service component.

Finally, we have to describe how services and agents can obtain such a context object to access a service. For services this is straightforward, because the kernel provides a method to request a service context object of another service. In case of agents this is more complex, because agents do not have any reference to the kernel. We give an example of an agent requesting a service context object in the following section.

3.3. Agents

As common to all agency models, each agent must have a globally unique name or id which must not change during the life-time of an agent. As already mentioned

above, our model does not define how to create such a name, but we recommend to use a combination of user given *agent nick names* and implicit names that are computed by the agency and that guarantee uniqueness. The full agent name must be representable as a *String* object.

Agents are represented by objects of a specific class, as in every agency model. However, in our model, agent classes must only implement interface *Runnable*, i.e. they must provide at least a single public method *run*, which serves as central starting point. Mobile agents must implement the *Serializable* interface too. We abstain from defining any base class for all agents, as for example class *Agent* or *MobileAgent* as done for example in the Aglets model. Such a base class already defines several methods to access services, for example methods to communicate or methods to initiate the migration process. Although this might simplify agent programming, it also creates a dependence between agents and services that we want to avoid. Such a base class also prevents to add new services for agents as this would entail to modify the base class which would then lead into an incompatibility between agents migrating to other agencies. The consequence of agents as *Runnable*s is of course, that an agent is unaware of itself and its environment. It does know neither its name nor its hosting agency and must use services to obtain these information.

The life-cycle of an agent actually consists only of two states. The first one is *Running* which is characterized by assigning a thread to this agent that executes agent's *run* method. After this method has terminated, the agent might switch to state *Waiting* where no thread is assigned to this agent or the agent's thread is waiting to become activated again. Thus, the agent is now only a passive object that is waiting for a message or another external signal or event. Details about the life-cycle will be presented in the following section, when we introduce the basic functions of an agency.

If an agent wants to communicate to a service, a similar technique is used as for services. As an agent object does not have a reference to any object of the agency, there must be a static method which is provided by class *Context* and which is named *getContext*. Consider the following example:

```
public class MyAgent implements Runnable
{
  public void run()
  {
    IAgentMessageContext cxt;

    cxt = (IAgentMessageContext)Context.getContext("message");

    cxt.sendMessage( ... );
  }
}
```

The agent requests a context object of a service that has registered itself under service name *message* and uses it to send a message. We omit to discuss problems that arise from many service components providing the same service and we also omit to print parameters of method *sendMessage* for sake of simplicity.

Method *getContext* delegates the task of requesting an agent context object to class Kernel, which first identifies the agent that has requested a context by determining the current thread (the kernel maintains a directory for this). Second, the kernel asks the agency, which itself selects the service component that provides the requested service *message* and asks this component for an agent context object, which is then returned to the agent. If the agent requests a context object from this service component for the first time, it is created. In the other case, the component must return the same object as before.

We face two problems with this first example. First, if there is no service component providing a service under the given name, then class *IAgentMessageContext* does not exist. Second, the service registered under the given name *message* does return a context object that is not assignable to variable *cxt*. The first problem can be solved by Java's dynamic class loading concept. The class of type *IAgentMessageContext* is not loaded until it is accessed, which is in the above example not before the type cast. If the agent first verifies that a service with the given name exists, then this problem can be solved. The second problem can be solved by comparing class names. The following example shows the resulting code sequence:

```
public class MyAgent implements Runnable
{
  public void run()
  {
    IAgentMessageContext cxt;

    if( Context.existsContext("message", "IAgentMigrationContext" ))
    {
      cxt = (IAgentMessageContext)Context.getContext("message");

      cxt.sendMessage( ... );
    }
  }
}
```

We omit to print full qualified package names in this example. First, the agent verifies that there is (i) a component providing service *message* and (ii) this component has created a context object which is assignable to an object of class *IAgentMessageContext* (we omit package names here).

3.4. Functions of an Agency

In this section we will define how kernel, service components, and agents interact with each other. We will describe the basic functions according to the agent's life-cycle.

3.4.1. Registering an Agent. Registering an agent can be done by services in two ways. In the first case, an agent (as Java object) has not been instantiated yet. To register an agent, a nick name, the name of the agent's main class and a URL where the agent's classes can be found must be specified. The agent object is instantiated and a full agent name is computed. In the second case, the agent has already been instantiated by a service component and must now be registered with the kernel. In this case, the agent already has a full name. In both cases, the agent is finally enrolled with the agent directory and then started (explained below).

Before an agent is registered, all service components are informed that have been registered a listener for this event. Each service can access all information about the agent and is now able to vote against registering. For example, a service that scans an agent's code for pattern of malicious behavior, is able to prevent registering and starting of this agent. If no service has voted against, the agent is registered with the local agent directory. Finally, a second event is fired, by which registered services are informed about the finalization of the registering process. For example, a graphical user interface can now update its list of agents.

Finally, the agent switches to state *Running*, which is described in the next section.

3.4.2. Running an Agent. When an agent is started, its *run* method is invoked within an own thread of control. The model does not define, whether it must be same thread that executes the agent during its whole life-time, or thread-pooling is allowed. The latter technique is preferred due to performance reasons. As agents must be strongly separated from each other, no two agent threads must be member of the same thread group. While the agent is running, it can request context objects from services as shown above.

3.4.3. Termination of Agent's Main Method. Every time, an agent's method *run* terminates, it is decided whether the agent should be killed (i.e. deleted from the agent directory and finally garbage collected) or remain as passive object. It is important to note that this decision is not only up to agents but is influenced by the service components. The protocol that is proposed for this can be seen as a voting protocol that works as follows (compare Fig. 2). The kernel asks all service components sequentially about the local status of the agent by calling method *getState*. A component might announce that it wants the agent to be immediately restarted again (return value *restart*), or to continue to live without restart (*passivate*), or raise no objection to kill the agent (*terminate*).

If there is at least one service component that wants the agent to be re-started, then agent's method *run* is invoked immediately again. If no service component wants the agent to be restarted, but there is at least a single one that wants the

agent to continue to live, then the agent's thread might terminate or wait and the agent continues to live as passive object waiting to become started again. For example, as long as an agent has pending messages, a communication service should prevent the agent from being terminated. If no component raises an objection to kill the agent, the agent is removed from the agency and is eventually garbage collected.

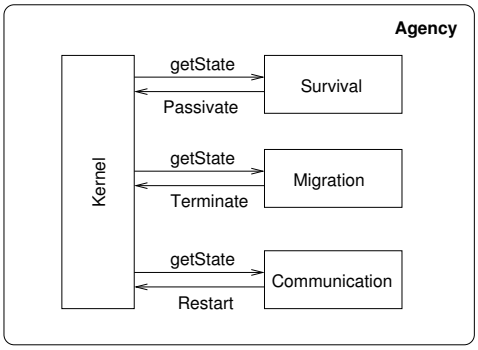

FIGURE 2. This figure illustrates the voting protocol that is used by the kernel to decide on the next state of an agent. The figure shows three service components to show the three different agent status results. In this case, the result of the voting protocol will be *restart*.

We want to mention two consequences of this protocol. First of all, if an agent wants to survive termination of its *run* method, then it must have registered with some service by requesting a context object. Second, as long as an agent possesses at least a single context interface, it cannot die.

3.4.4. Agent Termination. If the voting protocol results in terminating an agent, all registered observers are notified about this event to perform some final clearance and then de-register the agent. This notification process is implemented as a transaction using a two-phase commit protocol. In the first phase, each service component is *preparing* to delete the agent's context object. Only if all service components are ready to delete, the agent's context is deleted and the transaction is *committed*. Otherwise, if any component raises an exception, the transaction is *rolled back* and the agent is re-started again.

Figure 3 shows as an example the case of three service components, amongst others a migration component. During the first phase of the two-phase commit protocol, the migration component starts the migration process, if the agent has defined a migration destination in its context object. It uses the information stored in the agent's context object in order to open a network connection to the migration destination. After that, the agent's code and data are sent to the destination agency. On the destination site, the migration component might inform the kernel

(named *lock* in the figure) about the in-migration, for example to let the kernel validate the agent name. During this first phase of the protocol, the migration process is not finalized and the network connection between sender and receiver agency is not closed. In case of any error during the migration process, this service throws an exception. When the transaction is then committed, the migration component sends a command to the destination which finalizes the migration process and starts the agent. Otherwise, in case of an error during the first phase of the protocol, a different command is sent to roll back the whole migration process at the destination agency.

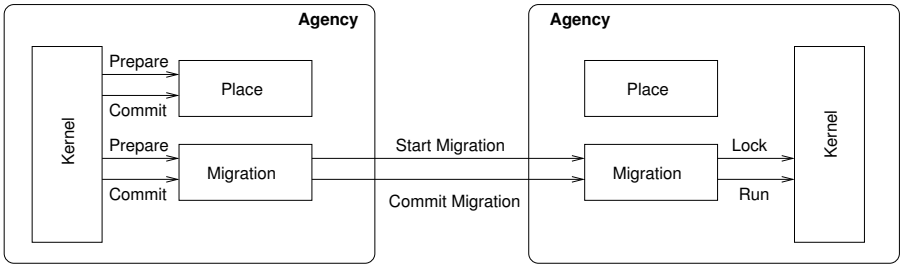

FIGURE 3. This figure illustrates the two-phase commit protocol that is used to stop agents.

3.5. Summary

In this section we have defined basic principles of our new agency model. Our aim was to identify the smallest common denominator of typical functions of an agency. The model specifies that an agency consists of three entities, i.e. the kernel that is responsible to execute agents and control their life-cycle, service components that provide high-level functions of the agency, and finally defines the main interface that all agents have to implement. Further, the model defines how these entities communicate to each other. The model does deliberately not define any high-level functions of an agency, as for example migration or communication and it does define only basic requirements related to security.

We see as main advantages of our new approach the following aspects.

1. Agent toolkits that conform to our new agency model are *compatible* to each other which actually has two aspects. First, it means that service components developed for one toolkit are applicable in all other agent toolkits too. We believe that it will be possible to enable exchange of research results on the basis of such software components in future. Second, agents and therefore complete agent-based applications are executable on every other toolkit too. The only requirement is that both toolkits provide the same set of high-level services.
2. A second advantage of our approach is that agent toolkits become very modular, as there is only a very small imperative kernel which forms the basis for

many service components to be added on. This modular architecture makes it very easy to port an agent toolkit to other devices, as components no longer needed can be simply removed. If it is too heavy-weighted for a resource limited mobile device for example, it can replaced by another component with less functionality and of less size.

3. Finally, every research group working on core research problems of mobile agents can implement their research results, for example new protocols for location-transparent communication as a service component and distribute it with other research groups. It is not necessary for them to implement other service components or even a complete agent toolkit by themselves.

4. The Tracy Toolkit

Tracy is a mobile agent toolkit designed according to the model defined in the last section. In this section, we will mention details of the Tracy implementation and especially describe several service components that have already been implemented.

4.1. The Tracy Kernel

The kernel of Tracy mainly consists of an implementation of the *Context* class mentioned in the previous section, classes for agent and service management (ASM), and a thread pool. In total, the kernel only consist of about 3000 lines of Java code.

If an agent switches from state *Waiting* to state *Running*, a sleeping thread from the thread pool is activated to execute the agent. After agent's method *run* has terminated, this thread is responsible to carry out the voting protocol. If the agent is not started immediately again (transition back to state *Waiting*), the thread is given back to the thread pool. The thread pool is initialized with a predefined number of threads (this number can be configured) and adapts dynamically to load variances.

The ASM classes are responsible to maintain a directory of all agents and service components. These classes communicate with the thread pool in order to execute agents. Agent names are defined to consist of the agent's nick name and a hash value that is computed over the agent's classes, the home agency's name, and the start time, to which the name of the home agency is appended. The ASM classes are also responsible to initiate the start of service components, which are named *plugins* in Tracy, because they can be started and stopped dynamically during runtime. Important functions of the kernel are guarded using Java permissions, so that it is possible, for example to prohibit plugins to start or kill agents.

4.2. Tracy Plugins

As part of the Tracy project, we have already defined several high-level service components and correspondent interfaces for context objects. They range from very simple plugins that provide agents access to the hosting agency to complex

services to manage overlay networks of agencies. In our opinion, this makes Tracy a comprehensive agent toolkit that can be used for the development of real-world applications yet. For each of these services, Tracy provides a default implementation as plugin. In Tracy, a plugin is deployed as JAR file, which contains a manifest to define the service name, version and author information, and dependences on other plugins, which are then started automatically before. In the following, we describe some of the most important plugins.

4.2.1. Place. The Place plugin provides agents an interface to the hosting agency to obtain information about themselves, their environment, and other agents. As already mentioned above, agents are innately blind, i.e. they are not aware of their environment and do not even know their name. Using this plugin, an agent can retrieve its name, name of its home and current agency and a list of all other agents and services currently residing on this agency.

4.2.2. Survival. The most important feature of the Survival plugin is to prevent an agent from being disposed after its *run* method has terminated. This plugin can also be used to schedule agent execution in the future. For example, an agent can define that it wants to be started once at a specific time or after some time interval. Additionally, an agent can also define that it wants to be started periodically.

4.2.3. Migration. The migration plugin that is used in Tracy is based on the Kalong migration component that is the result of a research project on high-performance mobility models and migration protocols [6, 9]. The main difference of Kalong as compared to other mobility models is that it provides a flexible and fine-grained migration protocol, which leads to a higher migration performance of mobile agents and to a flexible implementation of security protocols.

Current mobility models only offer a single *migration strategy*. For example, in Grasshopper the agent migrates only with its data but without any code, which is then dynamically loaded on demand (pull migration strategy). Other systems always transmit the agent as a package of code and data to the next destination agency (push migration strategy). In [8], we proofed that none of these simple migration strategies leads to an optimized network load and transmission time and proposed in the thesis that mobile agents should be able to adapt their migration strategies according to specific environmental parameters, as for example, the code size of each class, the probability that a class is used at the next destination, network bandwidth and latency, etc.

Therefore, Kalong defines a virtual machine for agent migration with a small set of methods to fully conduct the migration process by the agent or by the agent programmer. A program for this virtual machine is called *migration strategy* and we have already implemented several migration strategies, where the simplest ones just implement the simple migration techniques of Aglets and Grasshopper and the most sophisticated ones take several parameters into account, for example code execution probability, which is determined by static program analysis, and network bandwidth and latency information. In [6] we presented results of several

experiments, where mobile agents migrate in wide-area networks using Kalong and need about 30% to 50% less execution time for a complete round-trip than using simple push or pull-based strategies.

The Kalong component defines hot spots within the migration protocol, where each message that is sent or received via network is processed by a pipeline of so-called *protocol extensions*. Every protocol extension can modify network messages, for example compress, sign, or encrypt it. In [6] we already presented first protocol extensions for agent authentication, code signing and protecting data items against illicit tampering. We are currently working on adapting more sophisticated security protocols, especially path history [27], execution tracing [32], environmental key generation [28], and state appraisal [14].

The migration plugin also provides the possibility to use various network transmission protocols and we have already implemented support for TCP and SSL. As a feature provided in order to make programming of mobile agents more convenient, we have implemented transparent agency name resolution. The programmer of a mobile agent can address a destination agency using the full agency name *without* a port number on which the destination agency is listening for incoming request–instead of the full qualified domain name together with a port number as it is used in most other agent toolkits.

4.2.4. Agent Authorization. Security is a non-functional requirement of software systems and therefore cannot be implemented in a single software component, but is distributed over the kernel and several services. Some basic requirements concerning agent security were already introduced in the *kernel* and we mentioned that some security protocols can be implemented as extensions of the migration component.

Another component that is considered with security is *agent authorization*, which defines which permissions are granted to an agent. This service is an example of a component that might vote against starting an agent, if, for example, the agent is included on some black-list of possibly malicious agents. Otherwise, this service assigns permissions to the agent with regard to the agent's name and owner, its path history [12] and the result of code inspections done in the Kalong component.

4.2.5. Communication. The communication service developed as part of the Tracy toolkit supports the transfer of asynchronous messages between agents and services. Every agent has a message box in which new messages are stored. The agent can decide on its own how to handle these messages. It can decide to accept messages or not by closing its message box (even temporarily). So, the autonomy of an agent can be preserved. To send a message, the agent needs to know the name of the receiving agent.

This service does not support any kind of remote communication, i.e. an agent cannot send messages to another agent residing on a different agency. Even if both agencies were to reside on the same host sending messages between them would not be possible. We are currently working on an extension of this plugin

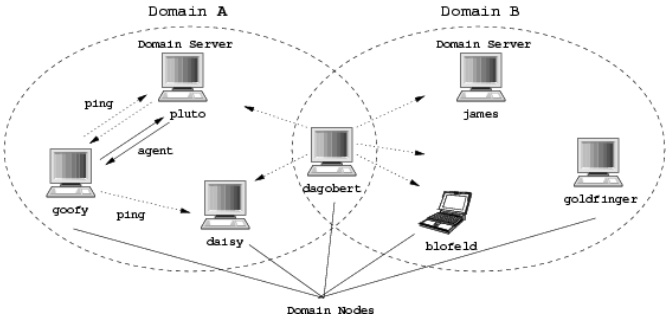

FIGURE 4. Example of two Tracy domains: Each contains a domain server and several domain nodes. Each node is registered at exactly one domain server. Agent server `dagobert` resides in the intersection of two domains and could be either registered at `pluto`, or at `james`.

which also allows remote communication using a forwarding pointer concept as already proposed by [25].

4.2.6. Tracy Domain Service. To manage logical Tracy networks we implemented the *domain management service*, which is completely implemented using stationary and mobile agents. A *domain* is a set of agencies that are connected in a local subnetwork. All agencies in one domain must be pairwise reachable by a UDP multicast. However, not all pairs of agencies in one local subnetwork must be member of the same domain–several logical Tracy networks can exist in one subnetwork independently. The domain management service is the basis for a comprehensive directory service in which each agency publishes services it offers for mobile agents [13].

A logical Tracy network consists of several agencies, which can either be in the role of the unique *domain server*, or in the role of a *domain node*. A domain server is responsible to manage a list of all registered domain nodes, whereas a domain node only knows its domain server. When a domain service is started, it first sends a UDP multicast message to all computers in the local subnetwork, see Fig. 4. If there already exists a domain server, this one will answer by sending a UDP package to the sender. This package contains the name of the agency on which the domain server resides. In a second step, the new agency sends a mobile agent to the domain server with the task to register it over there. If no domain server exists in the local subnetwork the new agency will become a domain server itself. As can be seen in Fig. 4, an agency (`dagobert`) can be in the intersection of two domains. The UDP multicast would be answered by more than one domain server. In this case the new agency is registered at the domain server which has answered the UDP multicast first. This makes sense, because Tracy domains should

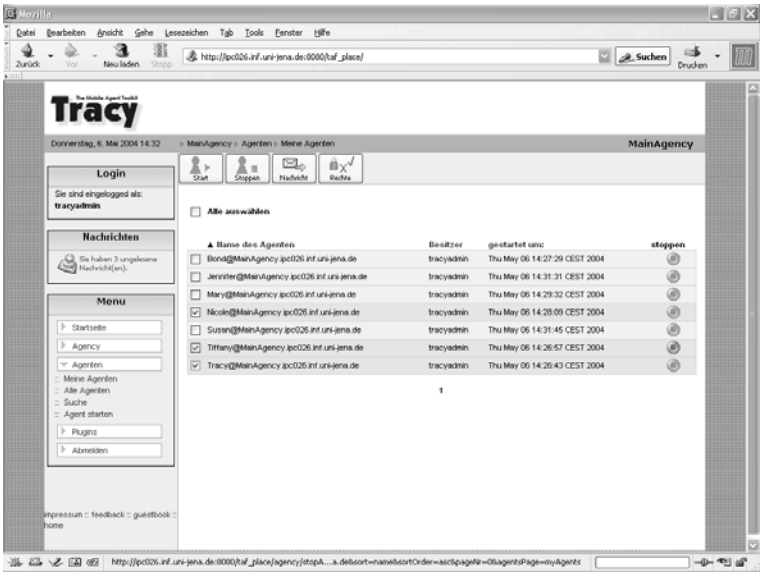

FIGURE 5. The screen shot shows an example of the Tracy Web interface, showing the list of agents currently residing at agency *MainAgency*.

contain servers that are situated locally and request time is a good indicator for this metric.

To build larger Tracy networks, domain servers can be connected dynamically. At the time, this process must be performed manually. For the future, we plan to build up larger Tracy networks by using mobile agents that explore the agent system and connect domain servers according to quality of services and quality of network connections. Note, that for an agent it is not necessary to follow this domain server connections to migrate to another agency–this can be done directly, as in a peer-to-peer network. The structure of a Tracy network is only important to explore new agencies in an unknown environment. The Tracy system contains all agencies, even if they could not find each other because they are in different logical Tracy networks–nevertheless it is possible for them to exchange mobile agents.

4.2.7. Other Services. Other services that are already defined and implemented as part of the Tracy project allow

- to send emails to arbitrary users using the SMTP protocol,
- to send short messages via a SMSC (using a SOAP Web service),
- multi-user management including dynamic permission management,
- to administrate an agency using a text-based user interface via Telnet protocol,

- to load public keys from a LDAP server (this is used by some security enhancements of the migration plugin),
- to launch agents automatically when an agency starts,
- to administrate an agency using the SOAP protocol.

As an application of the last service, we have already implemented a Web-based user interface, where Java servlets communicate to an agency using SOAP. Fig. 5 shows a screen shot of the Tracy Web interface.

5. Proof-of-Concept

This section describes an application scenario to mainly express two things: First and most significant to give evidence that our service-based approach does work and second to emphasize the applicability of well-structured mobile agent frameworks in production.

Thus, we have designed and implemented an application offering mobile services for customers of a fictive railway company on top of our mobile agent toolkit Tracy. The main goal of that application is to provide a passenger while traveling with in-time information about schedules and state of potential or prospective trains he is going to use during his journey. That passenger can use mainly two functions with our application directly from his mobile device. On the one hand he is able to compose complete time tables including departure and arrival times, transfers, platform numbers, etc., and on the other hand the application can be configured to observe train states to provide the passenger with latest information regarding train delays or breakdowns giving a passenger an opportunity to react on those incidents and to keep his journey efficient.

When we have a closer look at our application it can be recognized that two different types of agencies are needed. On the railway company's side we need a high performance and high scaling agency able to host thousands of concurrent agents, each represents a single user. Additionally, the company's agency offers services for mobile agents to access the time table database and to compose schedules. Finally, it offers a service for mobile agents to register with some kind of notification manager in order to retrieve information about train latencies.

In contrast, a small agency is needed on the customer's side, which easily adapts to the restricted resources, such as low processor performance and limited memory. That client agency has to provide also additional services, i.e. to connect to other local applications (e.g. PIMs) and to enable communication with the user via a graphical interface. We have chosen a PDA as basic hardware for our application in the first phase because it offers sufficient resources for running Java programs and hosting mobile agents. In a next step our client agency could be further downsized to another version which can be executed on even smaller Java-capable devices, for example mobile phones.

Mobile agents are used as information carriers between passenger and company agency in order to transmit schedules and delay information from the company agency to the client agency and to transmit schedule requests and observation configurations from the client agency. Additionally, mobile agents offer additional functionality to passengers, such as the opportunity to re-plan a journey if a train delays. Following this, both agencies need to implement a migration service.

At this point the advantages of a fully service-based approach for designing agent toolkits are brought to bear. Obviously we have two different implementations of our agency model both able to run the same migration service. The only service running on the PDA agency is the migration service that we were able to adopt to the restrictions of the mobile device and new Java version successfully.

Let's have a look at a concrete scenario. Imagine a salesman to plan a business trip by train from Jena to Hamburg. We presume he is already a customer of our fictive railway company and has yet a Tracy agency installed on his PDA. A day before traveling the salesman assigns a mobile agent via the graphical interface of our application to obtain the schedule for his journey. Therefore, he parameterizes the mobile agent with details about start location and destination, date, and prospective departure and arrival time. Then the mobile agent migrates to the railway company's agency, interacts there with appropriate services for composing the schedule, and returns back onto the PDA for presenting the information to its owner. We assume the salesman has got a connection with two transfers.

The salesman creates a mobile agent and configures it to observe the state of all trains involved in his journey and starts it in the morning before the travel begins. The mobile agent determines the unique train identifiers and migrates again to the railway company's agency. This time the mobile agent subscribes to a certain service for being provided with information regarding to all incidents influencing the salesman's trip. If an event occurs, for example the second train delays 15 minutes, the mobile agent receives a notification from the service, migrates back onto the PDA and presents that information audio-visually. Thus, the salesman can react on this event, for example by notifying his business partners about his delay or even by looking for other transport opportunities to keep the time loss to a minimum. Of course, the scenario is a simple one. However, behind that idea there is much potential for further more sophisticated services for mobile users.

6. Conclusions

The main motivation for the work presented in this chapter was the lack of any widely accepted implementation of a mobile agent toolkit that can be used by researchers to implement and test their own research results. Our thesis is that research on mobile agents will benefit from our kernel-based approach, in which we only define basic concepts and functions common to all toolkits. Core services, for example agent migration, communication, management of logical agency networks, and parts of security issues are implemented as software components. All research

groups still working on core research topics related to mobile agents are invited to contribute to our idea by implementing their own research results as plugins for Tracy.

In our opinion, our Tracy approach differs from already existing models for mobile agent toolkits in the following aspects: Current models for agencies [3, 16] include design decisions for several core services already. We see as major drawback of such an agency model that it creates many dependences between actually independent services. In fact, our model abstracts from these services and in contrast attempts to move as many functional requirements of a mobile agent toolkit into such services. To extend the discussion of related work started in Sec. 2, we mention two mobile agent toolkit that also claim to employ a kernel-based approach, namely JavaSeal [10] and MobileSpaces [31]. In both toolkits a kernel is defined comprising of core functionality and additional basic services. In JavaSeal these services are migration, communication, and security–in MobileSpaces it is mainly migration, as services are implemented as mobile agents in this toolkit. Our approach goes a step further by defining core services such as migration and communication as exchangeable components that are not part of the kernel. Besides, services are clearly distinguished from agents.

Finally, we mention Semoa [30] as an example of another extendable toolkit. Agents are also represented as *Runnable*s in Semoa and the concept to decouple agents and services using context objects is comparable to our approach. Agents offer application-specific services by registering a service object with the single *environment*. We see as main difference to our approach that Semoa handles application-specific services and the two *core services* for agent migration and communication differently. Whereas the first class of services can be plugged into the system during runtime, core services seem to be strongly coupled into the design of the whole toolkit. In Tracy, we do not distinguish between these two classes of services and migration and communication are both optional services.

References

[1] Joachim Baumann. *Mobile Agents: Control Algorithms*, volume 1658 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[2] Joachim Baumann, Fritz Hohl, Nikolaos Radouniklis, Kurt Rothermel, and Markus Straßer. Communication concepts for mobile agent systems. In Kurt Rothermel and Radu Popescu-Zeletin, editors, *Proceedings of the First International Workshop on Mobile Agents (MA'97), Berlin (Germany), April 1997*, volume 1219 of *Lecture Notes in Computer Science*, pages 123–135. Springer-Verlag, 1997.

[3] Christoph Bäumer, Markus Breugst, Sang Choy, and Thomas Magedanz. Grasshopper — A universal agent platform based on OMG MASIF and FIPA standards. In Ahmed Karmouch and Roger Impey, editors, *Mobile Agents for Telecommunication Applications, Proceedings of the First International Workshop (MATA 1999), Ottawa (Canada), October 1999*, pages 1–18. World Scientific Pub., 1999.

[4] Fabio Bellifimine, Giovanni Caire, Agostino Poggi, and Giovanni Rimassa. Jade – A White Paper. *EXP in search of innovation*, 3(3):6–19, 2003.

[5] Diego Bonura, Leonardo Mariani, and Emanuela Merelli. Designing modular agent systems. In *Proceedings of Net.ObjectDays, Erfurt (Germany), September 2003*, pages 22–25, 2003.

[6] Peter Braun. *The Migration Process of Mobile Agents–Implementation, Classification, and Optimization*. PhD thesis, Friedrich-Schiller-Universität Jena, Computer Science Department, May 2003.

[7] Peter Braun, Jan Eismann, Christian Erfurth, and Wilhelm R. Rossak. Tracy – A Prototype of an Architected Middleware to Support Mobile Agents. In *Proceedings of the 8th Annual IEEE Conference and Workshop on the Engineering of Computer Based Systems (ECBS), Washington D.C. (USA), April 2001*, pages 255–260. IEEE Computer Society Press, 2001.

[8] Peter Braun, Christian Erfurth, and Wilhelm R. Rossak. Performance Evaluation of Various Migration Strategies for Mobile Agents. In Ulrich Killat and Winfried Lamersdorf, editors, *Kommunikation in verteilten Systemen (KiVS 2001), 12. Fachkonferenz der Gesellschaft für Informatik (GI), Fachgruppe Kommunikation und verteilte Systeme (KuVS) unter Beteiligung der VDE/ITG, Hamburg (Germany), February 2001*, Informatik Aktuell, pages 315–324. Springer Verlag, February 2001.

[9] Peter Braun, Ingo Müller, Sven Geisenhainer, Volkmar Schau, and Wilhelm R. Rossak. Agent migration as an optional service in an extendable agent toolkit architecture. In Ahmed Karmouch, Larry Korba, and Edmundo Madeira, editors, *Proceedings of the First International Workshop on Mobility Aware Technologies and Applications (MATA 2004), Florianopolis (Brazil), October 2004*, volume 3284 of *Lecture Notes in Computer Science*, pages 127–136. Springer Verlag, 2004.

[10] Ciaran Bryce and Jan Vitek. The JavaSeal mobile agent kernel. In Dejan S. Milojicic, editor, *Proceedings of the First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99), Palm Springs (USA), October 1999*, pages 103–116. IEEE Computer Society Press, 1999.

[11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture: A System of Pattern*. John Wiley and Sons, 1996.

[12] Guy Edjlali, Anurag Acharya, and Vipin Chaudhary. History-based access control for mobile code. In Jan Vitek and Christian D. Jensen, editors, *Internet Programming – Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 413–432. Springer-Verlag, 1999.

[13] Christian Erfurth, Peter Braun, and Wilhelm R. Rossak. Migration Intelligence for Mobile Agents. In *Artificial Intelligence and the Simulation of Behaviour (AISB) Symposium on Software mobility and adaptive behaviour. University of York (United Kingdom), March 2001*, pages 81–88, 2001.

[14] William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Authentication and state appraisal. In Elisa Bertino, Helmut Kurth, Giancarlo Martella, and Emilio Montolivo, editors, *Proceedings of the Fourth European*

Symposium on Research in Computer Security (ESORICS 1996), Rome (Italy), September 1996, volume 1146 of *Lecture Notes in Computer Science*, pages 118–130. Springer-Verlag, 1996.

[15] Robert S. Gray, George Cybenko, David Kotz, and Daniela Rus. Agent Tcl. In William R. Cockayne and Michael Zyda, editors, *Mobile Agents: Explanations and Examples*, pages 58–95. Manning Publications, 1997.

[16] Dieter K. Hammer and Ad T. M. Aerts. Mobile Agent Architectures: What are the Design Issues? In *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems (ECBS'98), Maale Hachamisha (Israel), March/April 1998*, pages 272–280. IEEE Computer Society Press, 1998.

[17] Wayne A. Jansen. Countermeasures for mobile agent security. *Computer Communications: Special Issue on Advances in Research and Application of Network Security*, 23(17):1667–1676, 2000.

[18] Mehdi Jazayeri and Wolfgang Lugmayr. Gypsy: A component-based mobile agent system. In *Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing (PDP), Rhodos (Greece), January 2000*, 2000.

[19] Dag Johansen. Mobile agents: Right concept, wrong approach (panel). In Anupam Joshi and Hui Lei, editors, *IEEE International Conference on Mobile Data Management (MDM'04), Berkeley (USA), January 2004*, pages 300–301. IEEE Computer Society Press, 2004.

[20] Neeran M. Karnik. *Security in Mobile Agent Systems*. PhD thesis, Univeristy of Minnesota, Department of Computer Science, 1998.

[21] Neeran M. Karnik and Anand R. Tripathi. Design Issues in Mobile Agent Programming Systems. *IEEE Concurrency*, 6(6):52–61, 1998.

[22] Ryszard Kowalczyk, Bogdan Franczyk, Andreas Speck, Peter Braun, Jan Eismann, and Wilhelm R. Rossak. InterMarket: Towards Intelligent Mobile Agent-based e-Marketplaces. In *Proceedings of the 9th Annual Conference and Workshop on the Engineering of Computer-based Systems (ECBS-2002), Lund (Sweden), April 2002*, pages 268–275. IEEE Computer Society Press, 2002.

[23] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[24] Dejan S. Milojicic, Markus Breugst, Ingo Busse, John Campbell, Stefan Covaci, Barry Friedman, Kazuya Kosaka, Danny Lange, Kouichi Ono, Mitsuru 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 Second International Workshop on Mobile Agents (MA'98), Stuttgart (Germany), September 1998*, volume 1477 of *Lecture Notes in Computer Science*, pages 50–67. Springer-Verlag, 1999.

[25] Luc Moreau. A Fault-Tolerant Directory Service for Mobile Agents based on Forwarding Pointers. In *The 17th ACM Symposium on Applied Computing (SAC'2002) — Track on Agents, Interactions, Mobility and Systems, Madrid (Spain), March 2002*, pages 93–100, 2002.

[26] Paul O'Brien and Richard Nicol. FIPA – towards a standard for software agents. *BT Technology Journal*, 16(3):51–59, 1998.

[27] Joann J. Ordille. When agents roam, who can you trust? In *Proceedings of the First Conference on Emerging Technologies and Applications in Communications, Portland, Oregon (USA), May 1996*, 1996.

[28] James Riordan and Bruce Schneier. Environmental key generation towards clueless agents. In Giovanni Vigna, editor, *Mobile Agents and Securtiy*, volume 1419 of *Lecture Notes in Computer Science*, pages 15–24. Springer-Verlag, 1998.

[29] Volker Roth. Obstracles to the adoption of mobile agents (panel). In Anupam Joshi and Hui Lei, editors, *IEEE International Conference on Mobile Data Management (MDM'04), Berkeley (USA), January 2004*, pages 296–297. IEEE Computer Society Press, 2004.

[30] Volker Roth and Mehrdad Jalali. Concepts and architecture of a security-centric mobile agent server. In *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001), Dallas, (USA), March 2001*, pages 435–442. IEEE Computer Society Press, 2001.

[31] Ichiro Satoh. An architecture for next generation mobile agent infrastructure. In *Proceedings of International Symposium on Multi-Agent and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'2000)*, pages 281–287, 2000.

[32] Giovanni Vigna. Protecting mobile agents through tracing. In Christian Tschudin, Joachim Baumann, and Marc Shapiro, editors, *3rd ECOOP Workshop on Mobile Object Systems: Operating System support for Mobile Object Systems, Jyvälskylä (Finland), June 1997*, 1997.

[33] Giovanni Vigna. *Mobile Code Technologies, Paradigms, and Applications*. PhD thesis, Politecnico di Milano (Italy), February 1998.

[34] Giovanni Vigna. Mobile agents: Ten reasons for failure (panel). In Anupam Joshi and Hui Lei, editors, *IEEE International Conference on Mobile Data Management (MDM'04), Berkeley (USA), January 2004*, pages 298–299. IEEE Computer Society Press, 2004.

[35] James E. White. Mobile agents. In Jeffrey Bradshaw, editor, *Software Agents*, pages 437–472. The MIT Press, Menlo Park, CA, 1996.

Information about Software

Software is available on the Internet as
( ) prototype version
(x) full fledged software (freeware), version no.: 1.0.1-40
( ) full fledged software (for money), version no.:
( ) Demo/trial version
( ) not (yet) available
Internet address: http://www.mobile-agents.org
Description of software: Tracy mobile agent toolkit
Download: http://www.mobile-agents.org
Contact point for question about the software:
Name: Peter Braun
Email: braun@mobile-agents.org

Peter Braun
Faculty of Information and Communication Technologies
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia
e-mail: pbraun@ict.swin.edu.au

Ingo Müller
Faculty of Information and Communication Technologies
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia
e-mail: imueller@ict.swin.edu.au

Tino Schlegel
Faculty of Information and Communication Technologies
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia
e-mail: tschlegel@ict.swin.edu.au

Steffen Kern
Friedrich Schiller University Jena, Computer Science Department
Ernst-Abbe-Platz 2, 07743 Jena, Germany
e-mail: steffen.kern@informatik.uni-jena.de

Volkmar Schau
Friedrich Schiller University Jena, Computer Science Department
Ernst-Abbe-Platz 2, 07743 Jena, Germany
e-mail: volkmar.schau@informatik.uni-jena.de

Wilhelm Rossak
Friedrich Schiller University Jena, Computer Science Department
Ernst-Abbe-Platz 2, 07743 Jena, Germany
e-mail: rossak@informatik.uni-jena.de