A Service-oriented Software Architecture for Mobile Agent Toolkits

Peter Braun, Ingo Müller, Sven Geisenhainer, Volkmar Schau, Wilhelm Rossak
Friedrich Schiller University Jena
Computer Science Department
Ernst-Abbe-Platz 2, D-07743 Jena, Germany
P.Braun@computer.org

Abstract

In this paper we propose a new software architecture for mobile agent toolkits. 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 high-level services for agent migration and communication, we propose to employ a kernel-based approach. The kernel only provides fundamental concepts and functions and abstracts from any of these high-level services. We propose a new model for agencies and present our Tracy mobile agent toolkit acting as a reference implementation of our new model.

1. Introduction

Mobile agents are currently a hot topic in the domain of distributed systems. Reasons are the 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 this domain by complementing other approaches, simply by adding mobility of code, machine based intelligence, and improved network- and data-management possibilities.

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 communicates with other agents and host systems. During the self-initiated migration, the agent carries all its code and the complete execution state with it. Mobile agents need on each host that they visit a special software that we name agency, which is responsible to execute agents, provide an environment, and offer several services for agents residing on this host. A mobile agent system is the set of all agencies of the same type together with agents running on these agencies as part of an agent-based application. To refer to a specific project or product, e.g. Aglets [14] or Grasshopper [3], we use the notion agent toolkit.

In the last years, research and development in the area of mobile agents made a great leap forward - we have seen tremendous research efforts especially in the area of mobile agent security [10], performance aspects of mobile agents [4], mobile agent communication [2] or control issues [1].

However, the current status of mobile agents research is still considered to be immature. We see as main reason that there is no single mobile agent toolkit available which implements a large set of research results. In fact, the current situation is characterized by a few tens toolkits developed by research groups and industrial companies during the last years. 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.

The reason for this is the lack of any reference architecture for agencies as well as the absence of an open and extendable implementation. The research community has to admit that in the area of distributed artificial intelligence such a reference architecture has been developed in form of the FIPA [15] standard.

We also face disparate perceptions of basic concepts of mobile agents, e.g.:

- What should a mobile agent be from the programmer’s point of view: an object of a specific type or just any serializable object?
- What level of communication is necessary: only simple local or location-transparent (global) communication?
- What level of security is necessary?
- What kind of mobility is necessary?

The disadvantageous consequence of these isolated islands of research is that findings cannot be transferred between projects in form of definite implementations, e.g. software components that can be installed in other agent
We describe our experience porting Tracy to mobile devices.

We present the Tracy toolkit, which acts as the reference implementation of our agency model. Within this project we have specified several interfaces for high-level services and provide first implementations. With Tracy we proof the possibility to design an agency without regarding to the mobility of agents – and later to plug-in this new service without modification of the kernel.

We describe our experience porting Tracy to mobile devices and lessons learned by developing a small agent-based application for mobile users.

We see the following main advantages. Agent toolkits conforming to our agency model are compatible to each other which actually has two aspects. First, 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. Another advantage would be 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 architecture makes it very easy to port an agent toolkit to other devices, as components can easily be adapted or simply removed. As a third advantage every research group would be able to implement their research results, e.g. new protocols for location-transparent communication as such a service component and distribute it.

The rest of this paper 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 systems. We will point out some shortcomings of current agency models and introduce our own model. After that we describe the Tracy mobile agent toolkit. Finally we will shortly summarize some of the most important software components that come along with Tracy already.

2 Related Work

In the eyes of the authors the timely development of mobile agent toolkits can be roughly distinguished into an early, a current, and a next generation phase. The early phase in the late 1990s is characterized by the development of less flexible layered mobile agent toolkits, such as Aglets [14] and Grasshopper [3] heading the main goal to provide a complete set of functions for the development of mobile agent based applications.

After the community recognized drawbacks from the early phase toolkits, the current phase begun with the new millennium [11] creating component based toolkits. Thus, toolkit flexibility, extensibility, and adaptability were improved. However, interoperability between toolkits still remains unresolved.

This problem has been recognized by several members of the Special Interest Group on Mobile Agents of the European AgentLink network trying to lead toolkits into a next generation phase. Currently, we see related research within the SeMoA project, especially concerning mobile agent interoperability. SeMoA offers some design and implementation issues which are nearly identical to our approach, for example agent life-cycle and pluggable services [16].

However there are differences as well. The interoperability concept of our approach focuses not only on agents but also on plugins with the goal that all implementations of our agency model de facto ensure interoperability with each other. A second issue is our different understanding of contexts. We consider a context to be a common shared resource based on interfaces, instead of a pur proxy object\(^2\), which is fixed bound to its service component.

Another fairly similar approach for an agent-toolkit architecture is proposed in [17] by Satoh. He also figured...
out a micro-kernel comprising of a minimal and common set of functions. However, any additional functionality is implemented using mobile agents accepting mobility as a core service of the whole architecture. In contrast, in our approach, mobility should be only an optional service component.

3 JAM – A New Model for Agencies

In this section we will introduce JAM (Java Agency Model), which is our model for agencies for mobile and intelligent software agents written in the Java programming language. Our goal is to define a model that consists of the smallest number of classes resp. interfaces and only defines an imperative set of functions in order to execute agents. In the following section, we will then introduce our mobile agent toolkit Tracy, which can be seen as an instance of this agency model.

Current models for agencies and collections of design issues for mobile agent toolkits [9, 12] include design decisions for several high-level services already. This notion will be used in the rest of this paper 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. An agency model simply made up of several services risks the creation of many dependences between actually independent services, resulting in more monolithic agencies with strongly-coupled components.

Our model does deliberately not define any high-level functions of an agency, as for example migration or communication and it does only define basic requirements related to security. We see an advantage in the possibility to split the core functions of an agency from all high-level services and we argue that it is even possible to drop all specific requirements according to the mobility of agents from the kernel.

We propose to split an agency into three layers: a single kernel, several additional software components, where each component provides a single service, and the agents. The kernel just establishes the basic environment for running agents, maintains a directory of agents currently residing at this agency and all services currently connected to the kernel. The kernel is responsible to start and stop agents and to control an agent’s life-cycle. The kernel also generates agent names, which must be globally unique in the whole agent system and are enforced to be representable as String.

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 JAM model defines two Java classes for the kernel. Class Kernel is responsible to maintain a directory of all agents currently residing at this agency, defines an agent’s life-cycle and a basic interaction model.

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.
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. This principle is also known as a sandbox. Furthermore 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. If multiple service components register themselves using the same service name (and thus providing the same service), 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 classloader in order to separate it from other components and agents. Please note, that the model does not define when a service component is started. It is possible to start an agency with a predefined set of services or to allow to plug-in dynamically on demand. Each service component obtains a reference to the kernel.

Services are allowed to access the kernel by calling appropriate kernel methods. The reverse communication direction is achieved using the observer pattern, which makes it possible to propagate kernel events to the registered service components.

Our agency model defines a loose coupling between service components among each other and between agents and components. Direct method calls are prohibited and replaced by so-called context objects, comparable to proxies. A service component must assure that each client holds just a single context object in order to prevent DOS attacks and to eliminate inconsistencies between different context objects.

The main advantage of this concept is that clients do not hold a strong reference to a service, which eases maintenance of services during runtime and invalidation of stale context objects.

We now describe only briefly how service components or agents can obtain such a context object from another service. Each service component has to provide two interfaces which extend interface IAgentContext and IServiceContext. Both interfaces extend interface IContext which is defined as part of the kernel. For example, a service for agent migration defines a context with methods to define the migration destination and a communication service defines an interface with methods to post a message. The model enforces that each agent must hold only at most a single context object of every service. If a client wants to communicate to a service it must request an appropriate service context object. In the case of services, this must be done using a method defined in class Kernel. The kernel now works as a mediator to other service components. For agents, this is explained in the following section.

3.3 Agents

As common in all agency models, each agent must have a globally unique name or id which must not change during the life-time of an agent. Our model does not define how to create such a name, but we enforce 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 Java based agency model. However, in our model, agent classes are just required to implement interface java.lang.Runnable, whose single public run method serves as central starting point. We abstain from defining a base class for all agents, defining several methods to communicate to the agency. Although this simplifies agent programming, it also creates a dependence between agents and services that we want to avoid. It 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 Runnables is of course, that an agent is unaware of itself and its environment. It knows 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 waits to become activated again, for example waiting for a message or another external signal or event.

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 getApplicationContext. Consider the following example:

public class Agent implements Runnable {
    public void run() {
    
}
The agent requests a context object of a service that has registered 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.

MethodgetContextdelegates this task to the 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 searches for a component that provides service message and requests an agent context object, which is then returned to the agent. Further requests for a message context must return the same context object.

We face two problems with this first agent. First, if there is no service component providing the requested service, then class IAgentMessageContext does not exist. Second, the service registered under the name message does not return a context object implementing the IAgentMessageContext interface. The first problem can be solved due to 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 handled by comparing class names. The following example shows the idea:

```java
public class Agent implements Runnable {
    public void run() {
        if (Context.existsContext("message", "IAgentMigrationContext")) {
            // ... now get the context
        }
    }
}
```

We omit to print full qualified package names in this example. Before requesting a service context, 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.

### 3.4 Functions of an Agency

In this section we will define how agents, agency, and services can interact with each other. We will describe the basic functions according to the agent’s life-cycle.

#### 3.4.1 Registering an Agent

Agent registration 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 object agent is instantiated and a full agent name is computed. In the second case, the agent has already been instantiated, e.g. 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 started.

Before an agent is registered, all services are informed that have been registered a listener for this event. Each service can access all information about the agent and are now able to vote against registering. For example, a service that scans the agent’s code for pattern of malicious behavior is able to prevent starting of this agent. If no service has voted against, registration proceeds and a second event is fired, informing registered services about 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. An arbitrary thread may be choosen to execute the agent. As agents must be strongly separated from each other, no two agent threads must be member of the same thread group.

#### 3.4.3 Termination of Agent’s Main Method

After agent’s method run has terminated, 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 decided by the kernel resp. the service components. The protocol that is proposed for this can be seen as a two-phase commit protocol that works as follows: The kernel asks all service components sequentially about the local status of the agent. 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 at least one service component that wants the agent to continue to live, then the agent’s thread might terminate or wait and the agent remains on the agency as passive object waiting to become restarted. If no component raises an objection to kill the agent, the agent is removed from the agency and is eventually garbage collected, which is reported to the observers to make some final clearance possible.
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. In the following section, we will describe our Tracy toolkit, which can be seen as a reference implementation of this model.

4 The Tracy System

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

4.1 The Tracy Kernel

The kernel of Tracy is further distinguished into a micro kernel and the agent and service management component (ASM) and consists in total only of about 1500 lines of code. The micro kernel mainly consists of an implementation of a thread pool.

The ASM component is responsible to maintain a directory of all agents and service components. It communicates with the micro kernel in order to execute agents and is also responsible to initiate the start of service components, which are named plugins in Tracy as they can be started and stopped dynamically during runtime. Important functions of the kernel are guarded using Java permissions for the sake of security.

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. For each of these services, Tracy provides a default implementation as plugin. In Tracy, a plugin is deployed as JAR file, which should contain a manifest to define the service name, version and author information, and dependences on other plugins. The latter has the effect that whenever a plugin is started, all dependent plugins are started before.

4.2.1 Place

The Place plugin provides agents an interface to the hosting agency and to obtain information about themselves and other agents. Using this plugin, an agent can retrieve its name, name of its home and current agency and a list of all other agents currently residing on this agency.

4.2.2 Migration

The migration service component that was developed as part of the Tracy project is named Kalong and is the result of a thesis about high-performance mobility models and migration protocols [4]. The main difference of Kalong as compared to mobility models is that Kalong 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. In [6], 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.

4.2.3 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. A special component that is considered with security is agent authorization, which defines which permissions an agent should have. This service is an example for a component that might vote against starting an agent, if, for example, the agent is enlisted 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 [7] and the result of code inspections done in the Kalong component.

4.2.4 Communication

The communication service developed as part of the Tracy project 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. However, this service does not support any kind of remote communication.

4.2.5 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 [8].

5 Conclusions

The main motivation for the work presented in this paper 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. Today's toolkits are either closed systems that are difficult to extended, or prototypes that are tailored to specific research issues, where certain elementary functions in order to develop mobile agent systems are missing.

Our thesis is that research on mobile agents might benefit from our kernel-based approach, where we only define basic concepts and functions common to all agencies. High-level services are implemented as low-coupling software components. All research groups are appealed for contributing to our idea by implementing their own research results as plugins for Tracy.

References