The Migration Process of Mobile Agents

Implementation, Classification, and Optimization

D i s s e r t a t i o n

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.),

vorgelegt dem Rat der Fakultät für Mathematik und Informatik der
Friedrich-Schiller-Universität Jena

von
Diplom-Informatiker Peter Braun,
geboren am 22. Juni 1970 in Neuss.

Gutachter

1. Prof. Dr. Wilhelm R. Rossak, Friedrich-Schiller-Universität Jena
2. Dr. Bill Buchanan, Napier University, Edinburgh, Scotland

Tag der letzten Prüfung des Rigorosums: 30. April 2003

Tag der öffentlichen Verteidigung: 13. Mai 2003

Abstract

Mobile agents provide a new and fascinating design paradigm for the architecture and programming of distributed systems. A mobile agent is a software entity that is launched by its owner with a user-given task at a specific network node. It can decide to *migrate* to other nodes in the network during runtime. For a migration the agent carries its current data, its program code, and its execution state with it. Therefore, it is possible to continue agent execution at the destination platform exactly where it was interrupted before. The reason for a migration is mainly to use resources that are only available at remote servers in the network.

This thesis focuses on the migration process of mobile agents, which is to our knowledge not considered in literature so far, although the performance of a mobile agent based application strongly depends on the performance of the migration process. We propose a general framework an an innovative set of notions to describe and specify the migration process. By introducing the concept of a *migration model*, we offer a classification scheme to describe migration issues in existing mobile agent systems. As an example, the migration feature of two well-known mobile agent systems, Aglets and Grasshopper, are characterized. A detailed analysis of network load of mobile agents as compared to client-server approaches in several typical application scenarios shows the potential benefits of mobile agents. However, the analysis also shows drawbacks of the simple migration techniques that are used in today's mobile agent systems. The main drawback of these simple techniques is the lack of adaptability, which causes the superfluous transmission of code and data.

We present a new migration model named Kalong, which overcomes these drawbacks. It provides the agent resp. the agent programmer with a very flexible way to migrate an agent. Using Kalong, a migration is no longer a monolithic transmission of code and data. It is possible to send only those pieces of code and data that are used at the next destination platform with high probability. The agent can of course dynamically load missing code or data items from appropriate servers. The Kalong migration model was implemented in the Java programming language and can be used as an independent software component. It should be usable in almost all existing mobile agent systems. This Kalong software component is extendable, which is shown by implementing basic security solutions.

We conducted experiments to show the performance of our new Kalong model in real network environments. The result of our experiments is that Kalong's new

features increase the performance of mobile agents as compared to all other existing migration techniques. Measurements also point to the influence of many parameters on the performance of mobile agents, as for example network quality, code size, transmission protocol, and security enhancements.

Zusammenfassung

Mobile Agenten stellen ein neues faszinierendes Design-Paradigma für den Aufbau und die Programmierung von verteilten Systemen dar. Ein mobiler Agent ist eine Software-Entität, die von ihrem Besitzer mit einem Auftrag auf einem Knoten eines verteilten Systems gestartet wird und dann zur Laufzeit auf andere Knoten des Netzwerkes *migriert*. Bei einer Migration nimmt der Agent seine derzeitigen Daten, seinen Programmcode und seinen aktuellen Ausführungszustand mit, so dass auf dem Zielknoten die Ausführung genau an der Stelle fortgesetzt werden kann, an der sie auf dem letzten Knoten unterbrochen wurde. Der Grund für eine Migration liegt bei mobilen Agenten vornehmlich darin, Ressourcen zu nutzen, die nur auf anderen Knoten im Netzwerk angeboten werden.

Diese Arbeit konzentriert sich auf den Migrationsprozess für mobile Agenten, dem in der Literatur bisher wenig Aufmerksamkeit geschenkt wurde, obwohl er die Ausführungsgeschwindigkeit eines Agenten entscheidend beeinflusst. Es wird ein allgemeines Rahmenwerk für den Ablauf einer Migration vorgestellt und erstmals einheitliche Begriffe zur Beschreibung des Migrationsprozesses definiert. Unter dem Begriff des Migrationsmodells wird ein Klassifikationssystem zur Beschreibung der Migrationseigenschaften eines mobilen Agentensystems vorgestellt und die beiden am weitesten entwickelten Systeme Aglets und Grasshopper in diesem Schema beschrieben. Eine detaillierte Analyse der Netzbelastung von mobilen Agenten im Vergleich zum traditionellen Client-Server Ansatz in mehreren typischen Anwendungsszenarien zeigt das Potential von mobilen Agenten zur Verringerung von Verarbeitungszeiten. Allerdings zeigt die Analyse ebenso die Nachteile der in heutigen Agentensystemen verwendenten sehr einfachen Migrationstechniken. Der Hauptnachteil dieser Techniken ist die fehlende Anpassbarkeit, wodurch vielfach Programmcode und Daten des Agenten überflüssig übertragen werden.

Es wird ein neues Migrationsmodell mit Namen Kalong vorgestellt, das diese Nachteile beseitigt und dem Programmierer eines mobilen Agenten eine sehr flexible Technik für die Migration zur Verfügung stellt. Mit Kalong kann getrennt von der Geschäftslogik des Agenten die Migration im Detail beschrieben werden. Bei Kalong wird die Übertragung von Code und Daten im Gegensatz zu anderen Migrationsmodellen nicht mehr als untrennbare Einheit gesehen. So ist es möglich, bei einer Migration nur solche Codefragmente zu übertragen, die mit hoher Wahrscheinlichkeit auch auf der Zielplattform gebraucht werden. Ähnlich können die Daten eines

Agenten getrennt übertragen werden, so dass einzelne Daten nicht zu solchen Plattformen mitgenommen werden müssen, auf denen sie mit hoher Wahrscheinlichkeit nicht gebraucht werden. Der Agent kann jederzeit fehlenden Code oder Daten von dafür geeigneten Server-Plattformen nachladen. Kalong wurde als Software-Komponente implementiert und kann in nahezu allen derzeitigen Agentensystemen verwendet werden. Die Komponente ist erweiterbar und an konkrete Anforderungen des Agentensystems anpassbar. Die Ergänzung der Komponente um grundlegende Sicherheitstechniken wird am Beispiel gezeigt.

Zahlreiche Messungen in realen Netzwerken zeigen die Vorteile des neuen Migrationsmodells. Er stellt sich heraus, dass durch die neuen Funktionen von Kalong die Ausführungsgeschwindigkeit von mobilen Agenten gegenüber bisherigen Migrationsmodellen gesteigert werden kann. Die Messungen zeigen außerdem den Einfluss verschiedener Parameter, wie beispielsweise Netzwerkqualität, Größe des Agenten, Übertragungsprotokoll und verwendete Sicherheitstechniken.

Acknowledgments

First, I would like to thank my adviser, Prof. Dr. Wilhelm Rossak, for his wonderful way of supervising my thesis. Working with him was a great pleasure. He gave me the chance to choose a fascinating new research topic and to initiate a new project for it. He always motivated me to try out new things and gave invaluable advice, if some of these activities got slightly off course. I esteem his capability and feeling for the situation, when he restrained me in the right moment, if I, not for the first time, took more than I could handle. Many thanks for all the faith, the useful discussions, and for all the support. I think, I've learned a lot.

Thanks to Dr. Bill Buchanan from Napier University, Edinburgh, Scotland, for being second reviewer of this thesis and the first external user and tester. Many thanks for his positive feedback and comments reassured us in our work and pointed out many important aspects.

This thesis is the first one published by a member of the Tracy project. Since I have started to work on this topic, several students could be raved for mobile agents – and became new colleagues later on. Without their help, it would have been clearly impossible to set up a project like Tracy. I thank my colleagues Christian Erfurth, Jan Eismann, and Sven Geisenhainer for their work, for interesting and very helpful discussions over the last years, and for helping to get Tracy a useful mobile agent system. I owe a great deal of thanks to former students Chris Fensch, Volkmar Schau, and Ingo Müller who helped by implementing parts of Tracy.

I thank the University of Jena, namely Dr. Karl Brosche and Dipl.-Ing. Elke Pohle, to gave us the opportunity of presenting the Tracy project at the CeBIT fair twice in the years 2001 and 2002. Thanks to my colleagues Uli Pinsdorf from Fraunhofer Society Darmstadt, Corinna Flüs from University of Essen, and several other colleagues from the Special Interest Group on Mobile Agents which is part of the AgentLink European network of excellence in agent-based research, for useful discussions and meetings.

Finally, I would like to thank my friends from the *agent factory* team, where we are currently planning a university spin-off company: Susann Wipper, Ingo Müller, Sven Geisenhainer, Jürgen Frotscher, Volkmar Schau, and Alex Pauls.

Last but not least, I would like to express my warm thanks to my family. This thesis would not have been written without the unconditioned support of my parents. I wish to thank my wife, Eva, for being the most wonderful person in the world, for

her generous love, indefatigable support, and admirable patience during this process and for always being there for me.

Contents

List of Figures

List of Tables

1. Introduction

Until some years ago, the term *distributed system* was mainly used for a network of several computer systems with separated memory which are connected to each other by a dedicated network. The computers used in such a distributed system are almost homogeneous, which means that they have the same type of processor and same type of operating system. The network is more or less static: computers are only rarely switched off, network connections between hosts are always reliable and provide constant bandwidths. Each computer has a fixed IP address and network packet routing is done via local switches. Up until today, this type of network is typical for most applications.

Currently, we see rapidly evolving network and computer technologies. The Internet as a *network of networks* with heterogeneous computers has become widely accepted as a very important medium for any kind of information exchange. The number of people and companies providing services in the Internet increases continuously and is even surpassed by the mere number of Internet users. Many different types of services are offered in the Internet, first of all electronic mail and electronic file exchange. Without any doubt, the most successful Internet service is the World Wide Web. Whereas, in the beginning, the Web was only a medium to publish information on so-called Web sites, we now see the dissemination of novel applications in the Web.

Most of these applications are part of the electronic commerce domain, for example online shops or electronic marketplaces. They are built using traditional design technique called client-server, where a single powerful computer system (server) holds data to be shared over the network, and less powerful computer systems (clients) access the server using a network. In Internet applications the server not only holds data but also executes application code in form of Java servlets or some other kind of server-based language. In this paradigm, the client is only responsible for the graphical user interface, which is in Internet applications some kind of Web interface using HTML pages.

Due to the success of the World Wide Web, the notion of what we know as a distributed system shift outwards. The Web can be considered as a predecessor of future distributed systems, as we notice an exponential growth of services available on the Internet already today. In the future we will see hundreds of million of people being on-line by different means of communication using hundreds of million services in the Internet. Only in the center of the network, connections will be of copper or

fiber – on the edges of the network, wireless connections based on new standards like Bluetooth, WLAN, and UMTS will become popular. Bandwidth in the center of the network will increase dramatically in the next years – and it will cover upcoming demands for transmission of large amount of data as needed, for example, in video streaming. On the outskirts of the network, however, available bandwidth will not increase as fast as in the center. People still use and perhaps will use for the next years Internet connections via ISDN or xDSL with not more than 128 Kb/sec. Therefore, the bandwidth gap between backbone and end-user connection will increase. Since backbone connections are fairly often renewed compared to the local bandwidths, this trend will continue over the next years.

Two major trends can already be seen entering the main-stream of interest: *Pervasive computing* means that everything might become a node in a distributed system. As computers become more and more tiny, computers can be found not only on desks but also in cars to regulate speed control, at wrists to show the time and control pulse, and in refrigerators to monitor the temperature.

The second trend we want to mention here is *nomadic computing* which means that users move during their work physically from place to place, logging into the system from very different computer systems, e.g. first from a system in the office over the company-wide local area network, later from home over an ISDN dial-up connection. Nevertheless, users want to see nearly the same working environment, the same applications and, above all, the same data. In addition, nomadic users demand for a seamless integration of different devices, making it possible to change the working environment from a desktop computer to a PDA in a few seconds.

All these new trends require new network-centric programming techniques. The client-server design pattern, successfully used for distributed systems in local area networks, is not able to face all the challenges of future distributed systems described above.

One very promising approach is *mobile code* resp. *mobile agents*. With *mobile code* we name a technique where code is transferred from the computer system that stores the code files to the computer system that will execute the code. A well-known example of mobile code are Java applets which are small programs available in a portable and interpretable byte code format. Applets are transferred from a Web server to a Web browser in order to be executed as part of a HTML page.

Mobile agents are a special type of mobile code. A mobile agent is a program that can migrate from a starting host to many other hosts 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 data, and the complete execution state with it.

The difference between mobile code and mobile agents is the fact that mobile agents initiate the migration process by themselves, whereas the migration of mobile

code is initiated from other software components, e.g. the Web browser in the case of Java applets. The second difference is that mobile code migrates only from a server to a client and does not leave the client to migrate back to the server or another client. Mobile code's lifetime is bound to the lifetime of the Web page it is part of and dies when the browser terminates or another Web page is requested. In contrast to this, mobile agents usually migrate more than once. Think of a mobile agent that travels to several hosts in order to collect prices for a desired product.

1.1. Motivation for This Thesis

The employment of mobile agents is justified by several advantages of this new paradigm as compared to the traditional client-server paradigm[1].

An important technical argument in favor of mobile agents is the *network load argument* (sometimes also called *performance argument*) on which we will focus in this thesis[2]. According to this argument, mobile agents are able to save network load and, therefore, decrease execution time to some extent by shipping code close to the data instead of shipping data to the code as it is done in the client-server paradigm.

To make this clear, we consider an example, where in a distributed system a server holds an image database and clients must filter this database for interesting images by analyzing image content. To analyze image content, the client has developed an algorithm for which the code is currently stored at the client. Using the client-server paradigm the client will request images from the server using remote procedure calls or remote method invocations. The server only offers an interface to filter images according to name, modification date, author, etc., but not image content, of course. The result of the request might be a huge number of images and all images must be transferred over the network to be analyzed locally at the client. The client filters the received images using the given algorithm. As the server interface does not offer filtering according to image content, it may happen that many images are transferred that are not interesting to the client. In this case, the client has to repeat its request several times to request other images (images of another author or older images), if the last request was not successful.

In the mobile agent paradigm, the client sends a mobile agent to the image server and the agent already contains the algorithm to analyze image content. The mobile

[1] Perhaps the first qualitative comparison of both paradigms was done by Harrison et al. in 1995. The paper was later published as Chess et al. [1997]. The authors discuss several advantages of mobile agents against client-server based techniques and conclude: "While none of the individual advantages of mobile agents given above is overwhelmingly strong, we believe that the aggregate advantage of mobile agents is overwhelmingly strong, because: ... b. While alternatives to mobile agents can be advanced for each of the individual advantages, there is no single alternative to all of the functionality supported by a mobile agent framework. ..." [Harrison et al., 1995, p. 17].

[2] We will discuss other important advantages later in Section 2.3.

agent can then communicate with the server locally, without producing any network traffic. After the agent has filtered the database, it will only take those images with it that passed the filter and thus are interesting for the client.

The advantages of mobile agents that can be concluded from the example above are that mobile agents can reduce network load

1. by reducing the overhead of network protocols (e.g. repeated requests), and
2. by filtering data at the server side.

It is obvious to see that this argument is only valid if, simply speaking, the mobile agent's code that has to be transmitted is not larger than the amount of data that can be saved by the use of a mobile agent. Transcribed into our example above, a mobile agent will produce less network traffic if its size is smaller than the amount of image data that is not transmitted due to filtering at the server side. In the other case, the client-server paradigm will produce less network traffic. It might be argued that it is inadmissible to restrict the comparison to the number of bytes not including the overall execution time. Using agents, execution time can be higher even if the number of bytes to transmit is smaller due to a large number of agents, all sharing server's processing load. For the moment, we assume to have a highly scalable server, which allows to neglect processing time.

It depends on the size of the request, the size of the reply, the code size and some other parameters to decide how much network load can be saved. For example, if the client requests *all* images, using mobile agents would be better in all probability. In the other case, if no image fits the request, the client-server technique would have been the better choice. As some of these parameters can be determined at runtime earliest, the decision between mobile agent and client-server paradigm must be done dynamically at runtime to achieve always lowest network load.

However, a dynamic decision between sending code to a server and shipping data to the client is very difficult to implement, because it requires to implement both paradigms and it requires an algorithm to forecast the reply size *before* having sent the request to the server. As, in general, it is not possible to develop such an algorithm, several approaches were published in the last years where the decision is done *before* designing the whole application. These approaches are based on mathematical models for network load where several parameters, as for example request size and reply size, are estimated by the software designer. The result of this network load analysis is either a mixture of agent migrations and remote procedure calls [Straßer and Schwehm, 1997], or a design decision for either the client-server, or the mobile agent paradigm [Carzaniga et al., 1997]. Looking into the future this decision might be wrong, if parameters of the mathematical model turned out to be different[3]. Apart

[3]We will discuss both approaches in detail in Section 3.1.

from these approaches, a detailed analysis of the reasons for mobile agents producing more network load than remote procedure calls is still missing.

1.2. Contribution of This Thesis

The main thesis of our work is that mobile agents need a sophisticated migration technique that allows fine-grained and flexible agent migration.

As described in the last section, mobile agents have the ability to save network load as compared to client-server techniques, but this advantage depends on the application scenario. In contrast to the two approaches described in the last section, we first investigate the reasons, which in total are responsible for higher network load of mobile agents in specific situations. The result is a collection of inherent drawbacks, which mostly are related to the migration process. It turned out that most of today's mobile agent migration techniques do not differ from mobile code technology, which is used for Java applets for example. We analyze the migration process in detail and we compare different approaches implemented in current mobile agent systems with regard to these disadvantages. We show that none of these systems has implemented techniques to avoid the possible drawbacks. Above all, the typical behavior of mobile agents to migrate several times between different hosts and the fact that in real-world agent-based applications many agents of the same type are alive in parallel, are not considered in current mobile agent systems.

We present a new migration technique which allows the programmer (or the agent itself) to adapt the agent's migration process during runtime. Using our migration technique it is possible to develop mobile agents that produce lower network load than with all existing mobile agent systems and to avoid typical drawbacks of mobile agents.

Incidentally, using our new migration technique, we are able to propose a new solution for implementing a dynamic decision between sending code or sending data. Our approach does not use a static network analysis between both paradigms, but uses mobile agents in any case. The mobile agent works in two phases, where in the first phase the agent filters the database locally at the remote server. According to the size of the result, the agents decides whether to stay at the server and analyze the result locally, or to ship the result to the client and process it there. The advantage of this procedure comes from the fact that the code for processing the request is much smaller than the code for data filtering and the latter is only transmitted to the server, if it is really needed at the server.

The following are the contributions of our work in particular:

- Analysis of drawbacks of mobile agents as compared with client-server.
- Description of the state-of-the-art in agent migration. Identification of the

design issues and design alternatives. Classification schema for the migration of mobile agents to assess an existing mobile agent system.

- A new mobility model which provides a fine-grained and flexible agent migration. Semi-formal description of the network migration protocol.
- The Kalong migration component, in which the new model is implemented. By providing a migration component we also present a new idea in building mobile agent systems, namely by using components-off-the-shelf, at least for agent migration.
- Implementation of some migration strategies and comparison regarding performance. Empirical evaluation in a real-world network.

1.3. Outline of This Thesis

This thesis is divided into four parts.

In the first part, we introduce the concept of mobile agents as a new paradigm to design and program distributed systems and compare mobile agents with more traditional techniques, as for example client-server techniques. We motivate focusing on effective migration as a key factor to increase the overall performance of mobile agent based applications. In the second part (pp. 47) we give an overview of state-of-the-art migration techniques in existing mobile agent systems. The main contribution of this part is a classification scheme to describe the design issues according to agent mobility.

The third part (pp. 113) is the core contribution of this thesis. Here, we motivate and introduce our new mobility model. We describe the architecture of a software component, named Kalong, which implements this new mobility model and we specify the new migration protocol SATP. The last part (pp. 201) of this thesis contains evaluation of our new migration component.

The enclosed CD-ROM contains the source code, the API documentation of Kalong, and an electronic version of this thesis.

Part I.

Introduction to Mobile Agent Technology

In the first part of this thesis we introduce mobile agents as a new design paradigm for distributed systems, present main advantages, and describe application domains that would benefit from the use of mobile agents. We analyze the performance of mobile agents in terms of network traffic and response time as compared to client-server based techniques and motivate why we focus on the migration aspect of mobile agents.

2. Mobile Agent Systems As a New Paradigm for Distributed Computing

In this chapter we provide an overview of traditional techniques for programming of distributed systems and introduce the mobile agent paradigm. We present the history of mobile agents beginning with early approaches using mobile code in the 1970's up to the latest mobile agent systems developed during the last two years. At the end of this chapter, we motivate their use in some specific application domains and enumerate important advantages of this new paradigm.

2.1. Traditional Techniques for Distributed Computing

In this section we will summarize three very important design paradigms used for the development of today's distributed systems. We will abstract in this presentation from implementation and language details and we will only describe the architecture of systems based on these paradigms.

In particular, we use the following abstractions[1]. A *site* represents the notion of location in a distributed system, for example a single computer as part of a network. A site hosts *resources*, which are any kind of immovable files, data bases, or any external devices. A site also hosts and executes *code*, for example by using a *virtual machine* or simply a micro-processor. We assume virtual machines to be immobile too[2], although moving processes is possible in distributed operating systems. The code contains the know-how to perform a specific computation. Note that a computation can only be successful, if code and necessary resources are located at the same site. At last, we have *interactions* between code, resources, and virtual machines on the same or on different sites.

In the following, we write S_A for a site with name A, R_A^M for a resource with name M at S_A, C^N for code with name N, C_A^N for code with name N at S_A, and M_A^N for a virtual machine executing code C_A^N at site S_A.

[1] For the following, we were inspired by Vigna [1998b, pp. 36] and Picco [1998, pp. 38].

[2] Vigna and Picco define a *computational component* as "active executors capable to carry out a computation", which are allowed to migrate to other sites, whereas in our approach *virtual machines* are immobile.

2.1.1. Client-Server Paradigm

Client-server is the most common paradigm of distributed computing at present. In this paradigm (Figure 2.1(a) on the facing page), there is code C^S executed by a virtual machine M_B^S (server) offering a set of services (for example access to resources R_B^x) at S_B and code C^C executed by virtual machine M_A^C (client) that needs this services in order to accomplish its task. Therefore, it sends a request to the server using an interaction, in which it asks for execution of a specific service, supplemented by some additional parameters. M_B^S executes the requested service using resources located at S_B and sends the result back to M_A^C using an additional interaction.

In this paradigm, no component is mobile, except of the request that is sent from the client to the server. The request usually contains the name of the service along with some additional parameters. This concept is comparable to a procedure call in programming languages, and, therefore, several programming concepts were developed that offer convenient use of the client-server concept in programming languages, for example *Remote Procedure Call* (RPC) [Birrell and Nelson, 1984; Nelson, 1981] or *Remote Method Invocation* (RMI) [Sun, 2002].

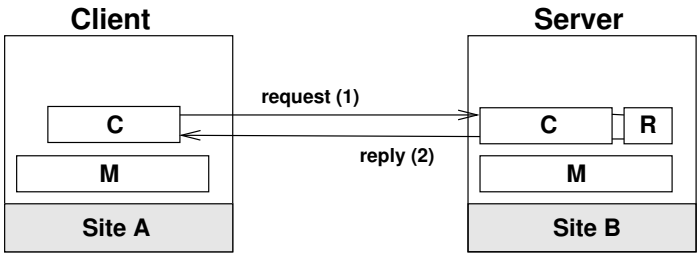
2.1.2. Remote-Evaluation Paradigm

In the remote-evaluation paradigm the same distinction is made between server and client as in the client-server paradigm (Figure 2.1(b)). Thus, there is code C^S executed by virtual machine M_B^S at site S_B having access to local resources, and code C^C executed by virtual machine M_A^C at site S_A. Important resources are located at site S_B. In contrast to the client-server paradigm, virtual machine M_B^S does not offer a suitable application specific service the client M_A^C could use. Instead, the client sends code fragment C^F (which is not executed so far) to the server to be executed there. Virtual machine M_B^S executes this piece of code, for example by simply initiating a new virtual machine M_B^F. During execution, local resources at site S_B are used and, afterwards, the result is sent back to the client using an additional interaction.

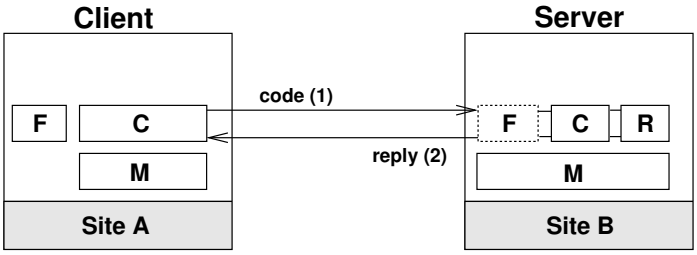
In this paradigm, the code fragment C^F is mobile and sent from the client to the server. The type of code depends on the concrete implementation of this paradigm and might be either some kind of script language that is transmitted as source code, or some intermediate code format that can be easily interpreted at the server. This technique is described by Stamos [1986], similar approaches were already published earlier. Examples are described in Section 2.2.3.

2.1.3. Code-on-Demand Paradigm

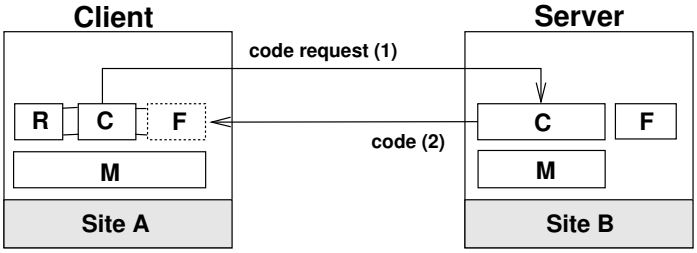
In the code-on-demand paradigm, roles are switched as compared to the remote-evaluation paradigm (Figure 2.1(c)). Here, virtual machine M_A^C has access to some resources R_A^x but lacks the know-how to access them. The code to access the resources



(a) Client-Server.



(b) Remote-Evaluation: Code fragment F, which is not executed at site A, is sent to site B and executed there. Dashed lines indicate that a component is dynamically loaded at site B.



(c) Code-on-Demand.

Figure 2.1.: Examples for traditional design paradigms. We use the following symbols: M stands for a virtual machine, R for a resource, C for a code component, and F for a code fragment. Lines between components indicate interactions, numbers indicate the order. If numbers are missing, it should be understood as a simple request/reply interaction.

is currently located at S_B. Thus, M_A^C interacts with M_B^S by requesting the know-how in form of code C^F. The code is executed at S_A by M_A^F.

In this paradigm, the code fragment F is mobile and sent from the server to the client. Concerning the type of code, the same remarks as given in the last section are valid. Java applets are a very prominent example of this design paradigm.

2.2. Mobile Agents

2.2.1. Software Agents

Let us start with the simpler notion of *software agents* first, because, as we will show later on, each mobile agent is also a software agent to a certain degree.

The word *agent* derives from the Latin word for actor and stands for a person that acts on behalf of another. In different languages the notion *agent* is used with different meanings. In English-speaking countries, for example, the word agent is used very often in a more general context, while in German speaking countries an agent mostly works for the secret service. In physical science, an agent can be an active substance that causes a reaction. Usually, to rent or buy a house a *real estate agent* is employed, or to plan a holiday, a *travel agent* is visited. Other sciences do also use the term agent. For example, in legal sciences an *agent provocateur* is a person hired to incite suspected persons to commit some illegal action that will make them liable for punishment.

In computer science, the term *agent* is known since the mid 1970's and it was introduced in the area of artificial intelligence. Most authors refer to a paper written by Hewitt [1977] as origin of the term agent. According to Foner [1997], the first reference can be traced back to Vannevar Bush and Douglas Engelbart in the late 1950's and early 1960's.

A software agent is a software entity which continuously performs tasks given by a user within a particular restricted environment. The involved software entity can be a computer program, or a software component, or, in the meaning of object-oriented programming languages, just a simple object. The definition of what exactly constitutes a software agent has been intensively debated in the research community for several years. Although this debate is not over yet, there is a common understanding that a software entity has to exhibit certain minimal features to qualify as an agent:

Autonomy Agents operate and behave according to a self-made plan that is generated in accordance with the user-given task. Agents do not receive every step of this plan stipulated by their owner in advance and they do not ask their owner for confirmation of every step.

Social behavior Agents are able to communicate with other agents or human beings by means of an agent communication language. Communication can be

restricted to pure exchange of information, or can include sophisticated protocols for negotiation, for example, when trading the price for a good or joining an auction. An separate branch of research deals with the problem of multiple agents working together on a single task in so-called multi-agent-systems. In this case, benevolent behavior is necessary for a successful undertaking.

Reactivity Agents perceive their environment by some kind of sensors and are able to react to identified events.

Proactivity Agents do not only react to stimuli from their environment, but they are able to take the initiative and do active planning. B. Le Du explains this with the following metaphor: *"The difference between an automaton and an agent is somewhat like the difference between a dog and a butler. If you send your dog to buy a copy of the New York Times every morning, it will come back with its mouth empty if the news stand happens to have run out of this specific newspaper one day. In contrast, the butler will probably take the initiative and buy a copy of the Washington Post, since he knows, that sometimes you read it instead."* [Bradshaw, 1996, p. 16]

Nowadays, the term *agent* has (unfortunately) become a buzzword that is used to signal innovative system characteristics, even if only a single feature of our list has been slightly touched. For example, some electronic mail clients are called *mail agents*, although they do nothing special aside from the usual task of delivering and collecting e-mails from your mailbox.

True software agents must be seen as an extension of the more general concept of objects or software components. Whereas software objects are passive, agents are active entities according to the so-called Hollywood principle: "Don't call us, we call you!".

2.2.2. Mobile Agents

A mobile agent is a software agent as described in the last section. However, it has a single additional, and very important, property that makes it unique: It is capable to move through a network of computer systems, hopping from node to node while fulfilling its task.

Software agents as described in the last section are called *stationary* to express that they are executed during their whole life-time on the same computer system, i.e. the one on which they were started on. In contrast, a mobile agent is not bound to a single computer system but it is free to migrate to other computer systems that are reachable on basis of the available network infrastructure.

Many different definitions for the term *mobile agent* exist. We want to give our own two definitions here. The first one targets the viewpoint of end-users:

> Mobile software agents are computer programs that act as a representative in the global network of computer systems. The agent knows its owner, knows his/her preferences and learns by communicating with its owner. The user can delegate tasks to the agent that is able to search the network efficiently by moving to the service or information provider. Mobile agents support nomadic users, because the agent can work asynchronously while the user itself is off-line. Finally, the agent reports results of its work to the user by different communication channels such as electronic mails, Web sites, pagers, or short messages via mobile phones.

In this definition many characteristics of software agents can be found, as we have introduced them in the last section. A mobile agent acts on behalf of a user, it knows its user, and gets to know him better over time. It has social behavior because it is able to communicate with the user, services, or even other agents. It works pro-actively, because it can, for example, contact its owner by many means of communication. The additional property of mobility can be seen as a very straightforward extension, at least from a human point of view, as it goes well with our natural understanding of how to search for information in a distributed environment.

The second definition that we want to present here draws more attention on the technical aspects of agent mobility.

> Mobile agents refer to self-contained and identifiable computer programs, bundled with their code, data, and execution state, that can move within a heterogeneous network of computer systems. They can suspend their execution on an arbitrary point and transport themselves to another computer system. During this *migration* the agent is transmitted completely, i.e. as a set of code, data, and execution state. At the destination computer system, an agent's execution is resumed at exactly the point where it was suspended before.

In this definition is nothing left from the characteristics of a software agent. We simply talk about computer programs or processes in the meaning of operating systems that are able to freeze themselves, move to other computer systems and resume execution over there. This more technical definition can be seen as a complement of the end-user driven one, simply targeting a lower level of abstraction.

There are three characteristics of mobile agents that must be stressed in any case:

1. Mobile agents are employed in wide-area and heterogeneous networks, in which no assumptions can be made concerning either reliability of the connected computers or security of the network connections.

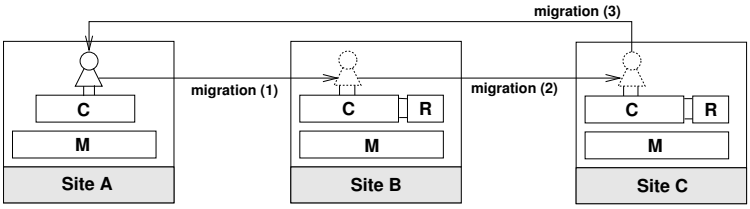


Figure 2.2.: The mobile agent paradigm. Agents are represented as small figurines like pieces of a board game and are shown above other, stationary code components (agencies) to indicate that they are dynamically bound to this code.

2. The mobile agent's migration is initiated by the agent (resp. its programmer) itself in contrast to mobile object systems, where object migration is initiated by the underlying operating system.

3. Migration of mobile agents is done to access resource only available at other servers in the network and not just for load-balancing, as in mobile object systems.

4. Mobile agents usually migrate more than once – this characteristic is sometimes called *multi-hop* ability. After a mobile agent has visited the first server, it migrates further to other servers to continue its task, whereas mobile code is only transferred once in the remote-evaluation paradigm resp. code-on-demand paradigm.

Let us have a closer look at this technical definition before moving on: By the term code we mean some kind of executable representation of computer programs. In case of script languages, like Perl or TCL this could be the source code, in case of the Java programming language [Arnold and Gosling, 2000] it is the portable intermediate Java byte code format, and in case of the C programming language it could be the executable machine language format for a single processor. By the term data we mean all variables of the agent – in case of object-oriented languages it is the set of all attributes of the corresponding object. Finally, by the term state we mean information about the execution state of the agent, e.g. information from within the underlying processor about current register values and instruction pointers. In case of the Java Virtual Machine [Lindholm and Yellin, 1999], which is the execution environment for Java-based programs, the state of an agent comprises of the operand stack, the instruction pointer, etc.

In the notion we introduced in Section 2.1 on page 9, we can describe the mobile agents paradigm as follows (compare Figure 2.2). At site S_A a virtual machine M_A^T

has the know-how in form of code, which is currently executed. During this execution the code realizes that it needs access to some other resources currently located at site S_B. Thus, M_A^T interacts with M_B^U to transmit the code, together with some more information about the current execution state. At site S_B virtual machine M_B^U executes the code providing access to the resources located at this site. Later, the code may decide that it needs other resources at other sites, e.g. S_C, and so the code migrates to another computer again.

A *mobile agent system* provides the infrastructure that implements the agent paradigm. Each computer system that wants to host mobile agents must provide an agent execution environment, called an *agent server* or *agency*. The agent server is responsible to receive and execute mobile agents and provides basic facilities for them, e.g to communicate, to protect them from malicious agents, and of course to migrate.

In this case, mobile agents are seen from the viewpoint of software engineering and distributed systems. They can be considered to be a new design paradigm in the area of distributed programming and a useful supplement of traditional techniques like the client-server architecture.

As almost all other mobile agent research groups, we have a rather pragmatic notion of the term mobile agent. To our understanding, *a mobile agent is simply any kind of software entity that is able to initiate a migration on its own within a network of heterogeneous computer systems and serves a specific task that has been specified by its owner.*

In fact, as we will show in the next section about the history of mobile agents, there is currently a clear separation between the two research communities - people working on intelligent software agents and people working on mobile agents. Unfortunately, people from the non-mobile agent community do claim very often that mobility is a pretty useless feature. This view can be best described by the statement that mobile agents are a "solution in search of a problem"[3]. On the other hand, people from the mobile agents community are sometimes proud of not working with *intelligence*.

2.2.3. A Short History of Mobile Agents

As we have pointed out earlier, the mobile agent paradigm relies heavily on the idea of mobile code. Thus, to some extent, we have to consider mobile code as an ancestor of mobile agents.

The idea to send code in an architecture-independent format to different hosts via a network was mentioned, probably for the first time, by Rulifson [1969]. He and his colleagues introduced the Decode-Encode-Language (DEL) which was published as RFC 5[4]. The idea was to download an interpretative program at the beginning of a

[3]Stated by John Ousterhout during an interview that is published online [IEEE IC-Online, 1997].
[4]Request For Comments, see `http://www.rfc-editor.org/` for more information about RFCs.

session, while communicating to a remote host. The downloaded program, written in DEL, could then control the communication and efficiently use the small bandwidth available between the user's local host and the remote host. Later, Michael Elie improved this concept and proposed the Network Interchange Language (NIL) as RFC 51 in 1970.

About 10 years later, a group at Linkoping University in Sweden had the idea to build a packet-oriented radio network they called *Softnet*. Each packet sent over the network was a program written in the FORTH programming language and each network node that received a packet immediately executed this FORTH program. Using this technique, every user was able to instruct every network node to provide new services. More information can be found in a paper by Zander and Forchheimer [1983].

Joseph R. Falcone faced the problem of providing client-specific interfaces to remote services across a heterogeneous distributed system [Falcone, 1987]. In contrast to offering a single interface with many small functions to satisfy the possibly high number of clients, Falcone proposed to enable clients to program their specific interfaces themselves, using a well-defined new programming language NCL (Network Command Language). In NCL a client sends an NCL expression to a server which, in turn, executes this expression using standard functions provided in form of a library. The server sends the result, which is an expression again, to the client which can start a computing process again. Thus, what we have here is primitive mobile code in both directions. Independently of Falcone, Stamos developed the *Remote Evaluation* (REV) approach, which extends the idea of *Remote Procedure Calls* introduced by Birrell and Nelson [1984]; Nelson [1981]. The motivation for REV is quite the same as for NCL described above. In REV, a client sends a request to a server in the form of a program. The server executes the program and sends the result back to the client. Other examples of mobile code sent within networked computer systems are remote batch job submission [Boggs, 1973] and the PostScript language used to control printers [Adobe Systems, Inc., 1999].

A second step towards mobile agents was then done by adding a minimal kind of autonomy to the messaging concept. We will refer to this technique, as is usual, as *mobile objects*, although nowadays the term *mobile objects* is often associated with Java RMI. The idea was that of active messages, i.e. messages that are able to migrate to a remote host. A message contained data and some program code that was executed on each server. However, the data portion was still dominant in this concept, while the active portion, i.e. the code, was more or less an add-on. As opposed to the mobile code approach, a mobile agent typically migrates more than once in its life-time and migration is initiated by the agent itself.

The MESSENGERS project [Fukuda et al., 1996] proposed the concept of so called *autonomous objects*, which were called *Messengers*. Messengers are able to migrate autonomously through a local area network of dedicated servers that accept these

objects. The difference to the techniques described above is that a Messenger is not only transferred to a single remote server but is able to autonomously roam a complete network. However, the concept was limited to static local area networks and did not include any notion of application level intelligence. A messenger's autonomy was limited to the level of technological and system level needs and not targeted at solving a user's problem.

A third predecessor of mobile agents are *mobile processes* from which mobile agents inherited the ability to capture the actual execution state of the processor or virtual machine they currently use. The idea was developed in the area of distributed operating systems in the late 1980's. In this framework, a process which is currently executed on a single computer system, can be moved to another system in order to balance the load of the distributed system as a whole. An example for operating systems with process migration is Sprite [Douglis and Ousterhout, 1991]. One technique to implement process migration is, for example, *checkpointing*. In regular time periods an image of an active process is captured and stored permanently. In order to migrate a process to another host, the last checkpoint is transmitted and the process is reactivated. If we compare, it has to be noted again that in mobile agents the motivation for migration derives not only from load balancing or other low level technical goals, but is typically driven by the demand to facilitate via the agent the use of various available services on the network's application layer.

It was in 1994, when James E. White, affiliated with General Magic Inc. at that time, published a white paper that initiated dedicated research on what we call *mobile agents* today. This paper was later republished in a book edited by Bradshaw [1996]. In this paper, White introduced the *Telescript* technology which comprises of a runtime environment and a dedicated programming language for mobile agents. This language already offered most of the very important aspects and abstractions of all current mobile agent systems. The further development of Telescript was nevertheless dropped when it became clear that this technology would not be able to stand against Java as the common basis for most mobile agent systems. For their work on mobile agents, General Magic received a U.S. patent in 1997 [White et al., 1997].

Since General Magic's initial project, the research community interested in mobile agents has been steadily growing. Some major conferences have been established that address solely this topic and a lot of workshops deal with specific subareas, like security or communication. A lot of alternative mobile agent systems have been developed since then. An almost up-to-date list counts about 70 different systems[5].

Today, nearly all systems use the Java programming language as basis for their development, only few systems additionally support other languages, as for example TCL or Scheme. Some of the mobile agent systems developed in the last years

[5] See `http://mole.informatik.uni-stuttgart.de/mal/mal.html`.

are Aglets by IBM[6], Voyager by ObjectSpace[7], and Concordia by Mitsubishi[8]. Two mobile agent systems are real commercial products: Grasshopper by IKV[9] and Agent Development Kit by Tryllian[10]. Systems developed for university-based research are for example Mole [11], Tacoma [12], D'Agents [13], Ajanta [14], Semoa [15], and our own system Tracy[16]. See Kiniry and Zimmerman [1997], and Wong et al. [1998] for a comprehensive review of Java-based mobile agent systems. A comparison of object oriented mobile agent systems was done by Gschwind [2000]. We omit here a detailed introduction to these systems and refer to literature for more information.

A first standardization approach to mobile agent systems was published by Milojicic et al. in 1999. The *Mobile Agent System Interoperability Facility* (MASIF) was backed by companies that were active in mobile agent research, e.g. IBM, General-Magic, and GMD Fokus and was published as OMG standard. MASIF bases on CORBA as system infrastructure. Aglets and Grasshopper are the only two systems that support the MASIF standard today.

2.2.4. Some Typical Applications for Mobile Agents

In the last years many research groups and companies have participated in the advancement of mobile agent systems. However, as the technology is new and radical

[6] Aglets [Lange and Ishima, 1998] is perhaps the most famous mobile agent system. The project became an open source project at Sourceforge (`http://aglets.sourceforge.net/`). The latest version of Aglets is 2.0.2, February 2002.

[7] The product was purchased by Recursion Software, Inc. (USA). The latest version of Voyager is 4.5 and is available from `http://www.recursionsw.com/products/voyager/voyager.asp`. Unfortunately, no white paper or any other documentation is available online. Two white papers [ObjectSpace, 1997, 1998] are related to older version of Voyager (1.0 and 2.0).

[8] The project is not alive anymore. See `http://www.merl.com/projects/concordia/` for more information. The main publication about Concordia is by Koblick [1999].

[9] The latest version of Grasshopper is 2.2.4b, July 2002, see `http://www.grasshopper.de` for more information. The Grasshopper mobile agent system is currently redesigned to become part of a new IKV product, named enago, see `http://www.ikv.de` for more information. Information about Grasshopper can also be found in Bäumer et al. [1999] and the manuals [IKV, 2001a,b].

[10] See `http://www.tryllian.com` for more information.

[11] Mole [Baumann et al., 1998; Straßer et al., 1997] was one of the first Java-based mobile agent systems. The project was completed in year 2000, but its home page is still available at `http://mole.informatik.uni-stuttgart.de`.

[12] For more information about Tacoma, see Johansen et al. [1995] and `http://www.cs.uit.no/forskning/DOS/Tacoma/`.

[13] D'Agents [Gray et al., 2002] is the successor of AgentTCL [Gray, 1997a], which was one of the first mobile agent systems. For more information, see `http://agent.cs.dartmouth.edu`.

[14] Ajanta [Karnik and Tripathi, 2001] focuses on secure and robust mobile agent execution, see `http://www.cs.umn.edu/Ajanta/` for more information.

[15] Semoa [Roth and Jalali, 2001] focuses on mobile agent security, see `http://www.semoa.org` for more information.

[16] An overview of Tracy is given in the appendix (pp. 267).

in its concepts, some type of *proof* has been asked for that would show that mobile agents, as a technology, are indispensable. (That this was never done for other technologies that are now widely used seems to be of no interest to those asking for the ultimate *killer application*).

What is accepted today is that mobile agents will not make any applications possible that would not have been possible before, using other, more traditional techniques. However, that holds also for other technologies, e.g. high-level programming languages: We could still develop all our systems by sticking to plain object code, even though nobody doubts any more that it was a good idea to develop higher level languages and introduce design and requirements phases into the software life cycle.

Thus, and this holds for mobile agents as well, when we talk about a new technology today, in most cases we talk about improved quality and management of complexity, the efficient use of resources in projects and the adequacy of concepts and tools. The point is not that something would not have been possible before, but how it can be achieved – which, of course, sometimes means the same: It might be possible in theory to build something similar to the Empire State Building without the use of cranes, steel and concrete, but who would ever want to do that?

On the same basis we argue that mobile agent systems provide a single framework and a very convenient abstraction, the mobile agent, to build distributed applications very efficiently. The point is not to look at one specific application, but to look at the whole set of possible applications and to understand that this new technology will enable a new level of networked software by delivering a sound basis to understand, handle and implement them despite their complexity and risks.

Nevertheless, its is possible to identify some application domains where mobile agents have already shown to be highly valuable and that seem "to ask for" that type of technology:

Electronic commerce, be it business to business or business to customer, suffers from the fact that it currently simply translates real-world business into electronic processes and data. Neither the advantages of the Web, nor the capabilities of software driven systems are fully utilized. To achieve that, a much higher degree of support for automation and a much better coverage of information sources must be offered: The customer simply wants to state what he/she wants, and is not interested to direct a system over hours manually to actually implement how this is done. Interfaces need to be unified, but a general standardization has shown to be nearly impossible. Huge amounts of data are shipped, and that often very slowly, and then thrown away after the most primitive evaluation. In all of these cases mobile agents can help, as they offer delegation and asynchronous task execution, are able to simulate unified interfaces to widely differing sources, and, last but not least, actually were born out of the need to send the evaluation process to the data, and not vice versa.

Information retrieval is another popular application domain for mobile agents.

Instead of moving large amounts of data to a single point where data is searched to extract the needed pieces of information, the code for searching the data is moved to the location where the data is. Nice examples are large graphical data bases, e.g. the data warehouses of satellite pictures that will charge for the downloaded data, not the information extracted. These systems also suffer from the problem of the overly simplified and non-standardized interface for multiple clients, as discussed above. Again, mobile agents will be able to unify that interface from the client's perspective and offer a higher and well adapted level of functionality.

Another typical application for mobile agents in the domain of information retrieval are multiple distributed sources. If the relevant information sources cannot be centralized, either because of technical reasons, e.g. in a network of fast updating sensors, or because of business-driven necessities, e.g. if the information at each node is proprietary and the owner does not agree to a centralized solution, mobile agents offer the only chance to develop a flexible solution that accepts the distributed nature of the given environment and offers a solution that is as distributed and scalable as the problem itself.

2.3. Technical Advantages of Mobile Agents

Although, mobile agents provide a new and interesting approach to distributed systems, there must be clear qualitative and/or quantitative arguments in favor of mobile agents to substitute them for more traditional techniques. However, while we believe that mobile agents are the most promising technology to solve most of the problems of the networked future, it should be said that we also believe that mobile agents will rather supplement many older techniques than killing them completely.

We will now present four major technical advantages, which are suitable for an introductory chapter as this one is, in some more detail. It is this set of basic technical advantages that opens the chance for improved and typical applications, as discussed above.

1. **Delegation of tasks.** As mobile agents are simply a more specific type of software agent, a user can employ a mobile agent as a representative to whom the user may delegate tasks. Instead of using computer systems as interactive tools that are only able to work under direct control by a user, autonomous software agents aim at taking care of entire tasks and work without permanent contact and control. As a consequence, the user can devote time and attention to other, more important, things. Thus, mobile software agents are a good means to cope with the steady information overload we experience.

2. **Asynchronous processing.** Once mobile agents have been initialized and set up for a specific task, they physically leave their owners' computer system and

from now on roam freely through the Internet. Only for this first migration a network connection must be established. This feature makes mobile agents suitable for nomadic computing, where mobile users can start their agents from mobile devices that offer only limited bandwidth and volatile network links.

3. **Adaptable service interfaces.** Current techniques in distributed systems that offer application service interfaces, usually as a collection of functions, constitute only the least common denominator of all possible clients. As a consequence, most of the interface functions are more or less primitive and clients will probably have to use a workflow connecting these functions in order to execute a complex, user-driven operation. If the communication overhead for exchanging messages between client and server is high, as compared to the execution time of each function, it would make sense to offer aggregated and more advanced functions as combinations of the primitive ones. However, since it is very difficult to track down every possible scenario in advance or even during runtime, this is usually not offered by the server's multi-purpose interface. Mobile agents can help in this situation by offering a chance to design a client-driven interface that is optimized for the point of view of the client (user), but is adaptable to different server interfaces. The key is to use a mobile agent to translate the more complex and user-driven functions of the client interface, at the server node, into the fitting primitive functions offered there. Thus, the mobile agent simulates a constant and highly specialized interface for the client (user), while talking to each server in its own language.

4. **Code-shipping vs. data-shipping.** This is the probably most cited advantage of mobile agents and it stands in tight relationship to the last one. For the same reason as mentioned above, service interfaces frequently offer only primitive functions to access data bases. A single call can, therefore, result in a huge amount of data sent back to the client, due to the lack of precision in the request. Instead of transferring data to the client where it will be processed and filtered and probably cause a new request (data-shipping), this code can be transferred to the location of the data (code-shipping) by means of mobile agents. In the latter case, only the relevant data, i.e. the results after processing and filtering is sent back to the client. This reduces network traffic and saves time, if the code for filtering is smaller than the data that must be processed. This advantage has been scrutinized in the last years by many different research groups and it has been verified in general.

3. Effective Migration As a Core Feature of Mobile Agent Systems

In this chapter we will motivate why we stress on the migration aspect of mobile agents in this thesis. As already introduced in Chapter 1, we consider the *network load argument* as one of the main advantages of mobile agents as compared with other design paradigms for distributed systems. According to this argument, mobile agents are able to save network traffic by shipping the code close to data, instead of shipping data to the code as it is done in the client-server paradigm. Although this argument can be proofed by experiments, there are cases where mobile agents produce higher network load than client-server techniques. This leads to the very important question for software designers: "Which paradigm produces lower network load?".

In the first section, we will describe two approaches from literature to answer this question. The first approach proposes a design decision between mobile agents and client-server on the basis of mathematical models. The second approach argues that only a mixture between mobile agent migrations and remote procedure calls can achieve smallest network load. Both approaches do not deliver a detailed analysis of the reasons for the mobile agents' bad performance.

Our thesis is that mobile agents suffer from several drawbacks that are all related to the technical bases used to provide the ability to migrate. Therefore, we will carry out a detailed network load analysis using mathematical models in Section 3.2. We will compare both paradigms in typical scenarios to identify reasons for mobile agents producing higher network load. The result is an enumeration of inherent drawbacks of mobile agents as compared to client-server techniques, which all are entailed with the details of the migration process. In the last section, we will discuss our results and mention a few other papers that also focus on network load analyzes of mobile agents.

3.1. Mobile Agents vs. Client-Server

3.1.1. Static Decision between Mobile Agents And Client-Server

This approach proposes a static decision between the two paradigms according to a mathematical analysis of the network load for a concrete application. The approach

was published several times for different applications. Carzaniga et al. [1997] and Vigna [1998b] discuss this approach for an application from the information retrieval domain, whereas Baldi et al. [1997], Picco [1998], and Baldi and Picco [1998] use this approach for an application from the network management domain. This approach compares network load for client-server, remote-evaluation, code-on-demand, and mobile agents.

The main thesis of the authors is that no paradigm is better than the others in every application scenario, but "The choice of the paradigm to exploit must be performed on a case-by-case basis, according to the specific type of application and to the particular functionality being designed within the application." [Vigna, 1998b, p. 42].

We describe their approach using the example of a distributed information system. To give an expression on the mathematical model, we also mention some of the most important parameters here. In the distributed information system N servers hold D documents each. The client's task is to download the *relevant* documents which are identified using keywords. The server offers for each document a header that also contains the keywords for this document. For sake of simplicity, the authors allow the following constraints. The relation between relevant documents and all documents equals i for all servers. The header information has length h bits for each document, and each document has length b bits. Requests sent from the client to the server have length r bits. Then, the authors model each approach using these parameters and finally gain an expression for network load for each paradigm, e.g. the network load using the client-server paradigm equals $((D + iD)r + Dh + iDb)N$ and the network load for the mobile agent approach equals $(r + C_{MA} + s + \frac{N}{2}iDb)(N + 1)$, where C_{MA} is the agent's code size and s is the size of the state. Based on an evaluation of these models, with estimated values for all parameters, the authors select a single design paradigm which is recommended for the implementation of this application.

Concerning an analysis of the drawbacks of mobile agents, the authors found that mobile agents always produce highest network traffic compared to all other design paradigms, because an agent carries all documents already found yet, whereas in all other paradigms documents are sent back to the client immediately. Thus, a mobile agent's data grows continuously with each hop, so that in sum it grows quadratically in the number of servers.

The authors assume in their model a network where transmission costs only depend on the number of bytes to transmit – and not on bandwidth and latency values. According to the authors themselves, this is unrealistic, but necessary to keep their model simple. Using such an uniform network, it is rather impossible that the mobile agent approach produces less network traffic than the remote-evaluation approach, because code in the remote-evaluation approach is smaller and mobile agents have to migrate $N + 1$ times, whereas in the remote-evaluation approach only N migrations are necessary. In our opinion it must be considered that networks for real-world

application are heterogeneous, meaning that, for example, a migration between two remote servers is faster than sending two requests from the client. Additionally, the authors can only estimate values for the parameters of their model, for example the size of the mobile agent's code, *before* having implemented this agent and do not explain how to obtain reliable values. The authors do not consider that parameters might change in the future, which might reverse their recommendation.

3.1.2. Mixture between Agent Migrations And Remote Procedure Calls

The second approach is proposed by Straßer and Schwehm [1997]. Their thesis is that only a mixture between agent migrations and remote procedure calls leads to minimal network traffic.

The authors develop a simple mathematical model for network load and execution time of client-server and mobile agent based approaches for a given application scenario. Several parameters are known in advance, for example the amount of communication necessary between client and server(s) as well as bandwidth and latency for all network connections. They model the advantage of mobile agents to filter or compress server results before sending it back to the client by a so-called compression factor σ. Agent migration is modeled as implemented in their Mole [Baumann et al., 1998] mobile agent system, where agent code is not always transmitted along with the agent's state, but is usually dynamically loaded from the agent's home server if necessary. Class downloading can be avoided, if the necessary class is already available at the destination agent server. Therefore, a parameter P models the probability to download any class from the agent's home server. Nevertheless, the authors do not evaluate their model with regard to class downloading probability P.

First, they evaluate a single client-server like interaction with regard to different values for the server result size and the compression factor. The result is as expected and shows that, for example, with low compression factors mobile agents produce higher network load because sending code to the server causes a fixed overhead, whereas with high compression factors mobile agents produce smaller network load. After that, they consider a scenario where a sequence of interactions between a single client and several servers is processed.

The authors' main idea to solve the problem of deciding between the two paradigms is that only a mixed sequence of agent migrations and remote procedure calls produces minimal network load. The agent only migrates to a subset of all servers to be visited, whereas the other servers are accessed using remote procedure calls. The optimal sequence depends on the size of requests and results, and on the network quality between each pair of nodes. All these parameters are assumed to be known in advance.

To assess this technique, the authors compute the network load for all possible combinations of migrations and remote procedure calls using a mathematical model.

Under the assumption that network bandwidth and latency are known in advance, they are able to show that it is actually a mixture that produces minimal network load. At last, the authors proof their theoretical findings by experimental validation. Here, an agent is able to compute its optimal communication pattern by itself using the developed mathematical model. Values for bandwidth and latency are measured by the underlying mobile agent system Mole. The authors compare three *mobility strategies*, where the agent migrates always, or never, or according to the results of the mathematical model. The measured execution times show that the optimized mobility strategy has always least execution time. Iqbal et al. [1998] continue on this work and present several algorithms to compute the optimal migration sequence of a single agent. The approach is based on an algorithm to determine the shortest path in a directed and weighed graph.

Both papers show that only a mixed sequence of remote procedure calls and agent migrations lead to an optimal network load. Thus, to determine the optimal communication sequence, knowledge of several parameters, for example about network bandwidth, latency, request, and result size are assumed. However, it is not clear how these values can be obtained in general and how robust their approach is against variations of these values.

3.2. Performance Analysis of Simple Mobile Agents vs. Client-Server

In this section we will develop ourselves a simple mathematical model in order to compare network load for both client-server and mobile agent based paradigms. The aim is to show under which circumstances the use of mobile agents causes lower network load and which characteristics of mobile agents are responsible for sometimes higher network load. Therefore, we do not consider the remote-evaluation and the code-on-demand paradigm here, although for example the remote-evaluation paradigm might produce less network load than the mobile agent paradigm, compare for example Picco [1998]. We want to keep the model as simple as possible. Therefore, we focus our analysis of network traffic in terms of transmitted bytes and do not consider transmission time, except of one scenario where we consider a heterogeneous network.

The application scenario we will use consists of a set of computers, where one system takes the client role and all other systems are servers. The client sends *client requests* to servers to obtain data items that are sent back as *server result*. The size of a single client request is B_{req} and the size of a server result equals B_{res}. If no data matches the request, the server has to send some kind of error notification about this case, which has size B_{rep}. A mobile agent that is sent from a client to a server has code of size B_{c} and state information of size B_{s}. Additionally, an agent carries data items, e.g. the client request or the results found at previously visited servers.

Parameter	Unit	Description
B_{CS}	byte	network load in the client-server approach
B_{MA}	byte	network load in the mobile agent approach
T_{CS}	sec	response time in the client-server approach
T_{MA}	sec	response time in the mobile agent approach
B_{req}	byte	size of a client request
B_{rep}	byte	size of a server error reply
B_{res}	byte	size of a server result
B_{c}	byte	size of a mobile agent's code
B_{s}	byte	size of a mobile agent's state
σ	$0 \leq \sigma \leq 1$	compression factor
m	number	number of servers to be visited
m^*	number	number of servers at which client-server and mobile agents produce the same network load
n	number	number of communication steps
p_i	$0 \leq p_i \leq 1$	probability that data is found at server i
$\delta(L_{\mathrm{i}}, L_{\mathrm{j}})$	sec	delay (latency) between network nodes L_{i} and L_{j}
$\tau(L_{\mathrm{i}}, L_{\mathrm{j}})$	byte/sec	throughput between network nodes L_{i} and L_{j}

Table 3.1.: Overview of the symbols used for the mathematical model in Section 3.1.

As we have already identified in Chapter 1 there are two individual advantages of mobile agents, both able to reduce network load:

1. Reduction of network load by avoiding network protocol overhead, e.g. avoiding many communication steps in a network protocol.

2. Reduction of network load by filtering and compressing data at the server side.

In our model, we describe data filtering and compression by a single parameter σ, $0 \leq \sigma \leq 1$, which stands for a *compression factor*. The compression factor is applied to the server result B_{res}, so that only $(1-\sigma)B_{\mathrm{res}}$ must be sent back to the client resp. carried by a mobile agent. Table 3.1 gives an overview of all used symbols.

For the sake of simplicity, we make the following additional assumption: When a mobile agent migrates to another computer it carries all its code, all state information, and all data with it. Only if an agent migrates to its home server, code transmission is omitted because code can be assumed to be already there. This corresponds to a migration technique that is implemented in most mobile agent systems. For the moment, we do not consider the impact of other techniques for agent migration, for example the one implemented in Mole (see Section 3.1.2). Our model of network

Scenario	$\mathbf{B}_{\text{req}}$	$\mathbf{B}_{\text{rep}}$	$\mathbf{B}_{\text{res}}$	$\mathbf{B}_{\text{c}}$	$\mathbf{B}_{\text{s}}$	σ	$\mathbf{p}_{\text{i}}$	Network	Figure
1.1	50	20	var.	2000	100	.7	n/a	hom.	3.1(a)
1.2	50	20	3000	2000	100	var.	n/a	hom.	3.1(b)
1.3	50	20	100	2000	100	n/a	n/a	hom.	3.2
2.1	100	50	10 000	3000	200	.8	$\frac{1}{m}$	hom.	3.3(a)
2.2	100	50	10 000	3000	200	.8	$\frac{1}{m}$	hom.	3.3(b)
3.1	100	n/a	10 000	3000	200	.8	n/a	hom.	3.4
3.2	100	n/a	var.	3000	200	var.	n/a	hom.	3.5
3.3	100	n/a	10 000	3000	200	.8	n/a	hom.	3.6
3.4	100	n/a	10 000	3000	200	.8	n/a	het.	3.7

Table 3.2.: Typical values of model parameters, which we use for all scenarios in Section 3.1. Here, "n/a" indicates that this parameter is not needed in this scenario and "var." indicates that this parameter is varied in this scenario.

load is placed on top of the TCP/IP stack, so that we do neither model TCP or IP headers, nor network load that is caused by data retransmission, etc.

We will now discuss the behavior of mobile agents in the following three scenarios:

1. Network of one client and one server, where the client accesses the server one or many times.
2. Network of one client and m servers, where the client is searching for a single data item that might be stored at any server.
3. Network of one client and m servers, where the client is searching for data items at all servers.

3.2.1. Scenario 1: Network of One Client And One Server

We first consider the case of reducing network load by data filtering and compression. In the client-server approach, the client sends a request of size B_{req} to the server, which answers with a result of size B_{res}. Here, no data filtering or compression can be applied. Thus, the amount of bytes that is sent over the network is:

$$B_{\text{CS}} = B_{\text{req}} + B_{\text{res}} \tag{3.1}$$

In the mobile agent approach, the client sends an agent to the server. The agent consists of code of size B_{c} and state information of size B_{s}. The agent carries the request object of size B_{req} as data item. At the server, the agent communicates locally with the server, which does not produce any network load. The agent has the code

to filter or compress the server result, so that only $(1-\sigma)B_{\text{res}}$ must be carried to the agent's home server. Please note, that the agent does not carry code and the request object during the home migration, because the code is already available at the home server and the request object is no longer needed. However, state information must be sent back to the agent's home. Thus, the amount of bytes for the mobile agent approach is:

$$B_{\text{MA}} = B_{\text{c}} + 2B_{\text{s}} + B_{\text{req}} + (1-\sigma)B_{\text{res}} \tag{3.2}$$

From Equation 3.1 and Equation 3.2 we can derive a verification of the thesis when mobile agents produce less network load than client-server approaches.

$$\begin{aligned} B_{\text{MA}} &\leq B_{\text{CS}} \\ B_{\text{c}} + 2B_{\text{s}} + B_{\text{req}} + (1-\sigma)B_{\text{res}} &\leq B_{\text{req}} + B_{\text{res}} \\ B_{\text{c}} + 2B_{\text{s}} &\leq \sigma B_{\text{res}} \end{aligned} \tag{3.3}$$

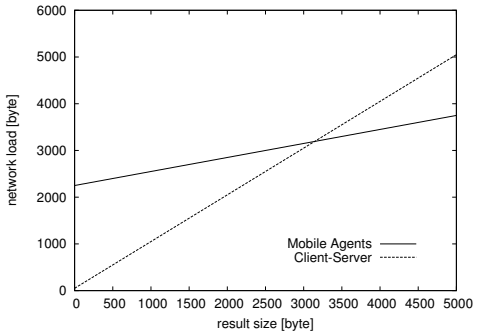
We see that a mobile agent produces lower network load, if and only if its code including double the state is lower than the amount of bytes of the server result the agent could save by compression and/or filtering.

We evaluate this scenario with the parameters found in Table 3.2 (Scenario 1.1 and 1.2). The result is shown in Figure 3.1. Figure 3.1(a) compares the network load of the client-server approach with the mobile agent approach for a fixed compression factor $\sigma = .7$ while varying the server result size between 0 and 5000 byte. The diagram shows that the usage of mobile agents produces less network load only if the server result size is large. The reason for this is the fixed network load overhead for transmitting mobile agent's code and state to the server, which is in this scenario equal to 2100 byte. Figure 3.1(b) compares the network load of the client-server approach with the mobile agent approach for a fixed server result size of 3000 byte while varying the compression factor σ between 0 and 1. This diagram shows that the usage of mobile agents produces less network load only if the compression factor is high.

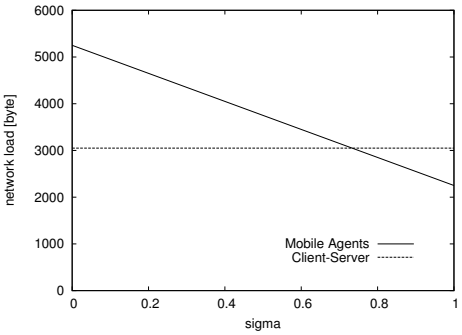
We will now have a look at the advantage of avoiding several network protocol steps. A typical scenario is when a client has to check a server periodically to get informed about changes, e.g. when a stock rate goes below a given limit. Therefore, the client sends requests of size B_{req} to the server, which answers with a server reply of size B_{rep} in the case of no changes and with a server result of size B_{res} when the change happened. Let us assume that the change happened after n requests were sent. For the client-server approach the network load amounts to

$$B_{\text{CS}} = nB_{\text{req}} + (n-1)B_{\text{rep}} + B_{\text{res}} \tag{3.4}$$

The agent has to migrate to the remote server, which costs $B_{\text{c}} + B_{\text{s}}$ and it carries the request of size B_{req}. After processing, the agent migrates back, which costs



(a) Server result size vs. network load for fixed compression factor.



(b) Compression factor σ vs. network load for a fixed server result size.

Figure 3.1.: Evaluation of Scenario 1.1 and 1.2: mobile agents produce less network load only if the server result is large or the compression factor is high.

$B_\mathrm{s} + B_\mathrm{res}$. Please note, that we do neither consider filtering, nor data compression in this scenario. For the mobile agent approach the network load amounts to

$$B_\mathrm{MA} = B_\mathrm{c} + 2B_\mathrm{s} + B_\mathrm{req} + B_\mathrm{res} \tag{3.5}$$

We evaluate this scenario with the parameters found in Table 3.2 (Scenario 1.3). The result is shown in Figure 3.2 and it compares the network load of the client-server approach with the mobile agent approach for fixed request, reply, and result size, while varying the number of request n necessary until the event occurs. It can be seen that in the client-server approach network load increases in proportion to the number of requests while mobile agents produce constant network load. Thus, mobile agents only produce less network load if the number of requests is beyond a threshold. The reason for this is again the fixed network load overhead for transmitting code to the server.

3.2.2. Scenario 2: Network of m Servers, Searching for a Single Data Item

In this scenario the client searches for a single data item that is currently only available at one out of a set of m servers. Thus, in the client-server approach, the client has to access each server sequentially until the information is found. We denote the set of all servers with $\mathcal{L} = \{L_1, \ldots, L_\mathrm{m}\}$. The probability that the information is

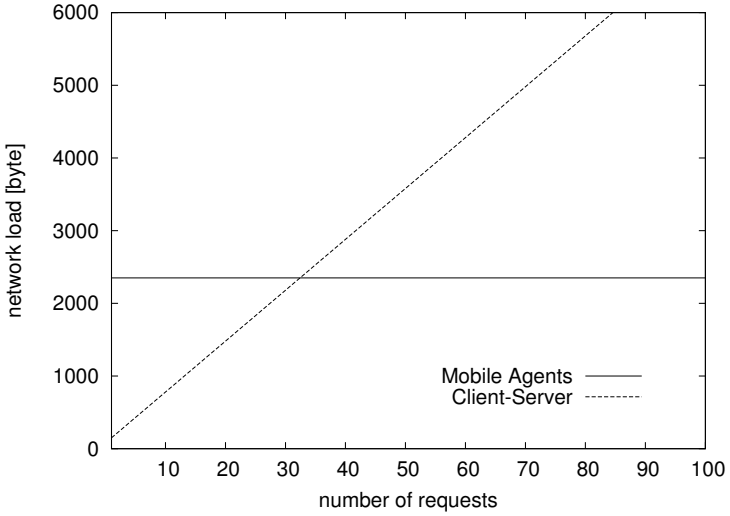


Figure 3.2.: Evaluation of Scenario 1.3: Number of request vs. network load. Mobile agents produce less network load only if the number of requests is high.

found at server L_i equals p_i, where $0 \leq p_i \leq 1$. Therefore, it is important in which order the servers are accessed and we define that the same order is used in the client-server approach as well as in the mobile agent approach. After the information is found at server L_i, servers $L_{\mathrm{i}+1}, \ldots, L_\mathrm{m}$ are not visited anymore.

Let us first consider the client-server approach. If the information item is found at the first server, then only a single client request of size B_req and a single server result of size B_res are sent. If the information is found at the second server, then two requests of size B_req, a single error reply of size B_rep (from the first server), and one server result of size B_res is sent. We weigh each single case with its probability and obtain the following network load:

$$\begin{aligned} B_\mathrm{CS} &= p_1(B_\mathrm{req} + B_\mathrm{res}) + p_2(2B_\mathrm{req} + B_\mathrm{rep} + B_\mathrm{res}) + \ldots \\ &\quad p_m(mB_\mathrm{req} + (m-1)B_\mathrm{rep} + B_\mathrm{res}) \\ &= \sum_{i=1}^{m} p_i(iB_\mathrm{req} + (i-1)B_\mathrm{rep} + B_\mathrm{res}) \end{aligned} \tag{3.6}$$

We now look at the mobile agent approach. If the information is found at the first server, the agent migrates only to the first server and comes back with the compressed result. Please remember that the agent does not carry its code when migrating home. If the information is found at the second server, the agent has to migrate three times,

it needs not to carry any reply message from the first server, but one compressed server result from the second server. Again, each case is weighed with its probability. In sum, this amounts to:

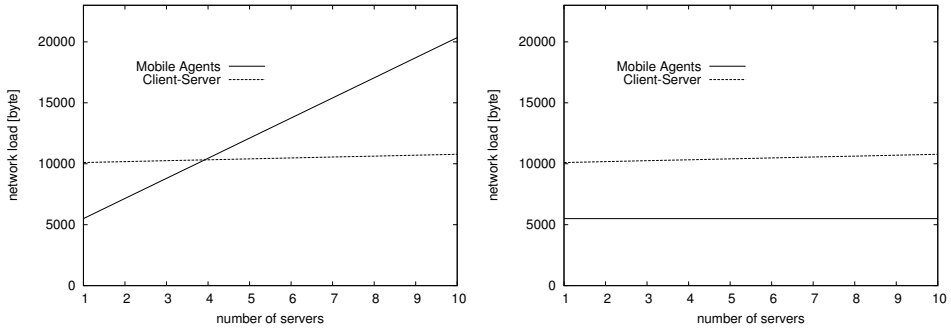
$$\begin{aligned} B_{\mathrm{MA}} &= p_1(B_{\mathrm{c}} + 2B_{\mathrm{s}} + B_{\mathrm{req}} + (1-\sigma)B_{\mathrm{res}}) + \\ &\quad p_2(2B_{\mathrm{c}} + 3B_{\mathrm{s}} + 2B_{\mathrm{req}} + (1-\sigma)B_{\mathrm{res}} + \ldots \\ &\quad p_m(mB_{\mathrm{c}} + (m+1)B_{\mathrm{s}} + mB_{\mathrm{req}} + (1-\sigma)B_{\mathrm{res}} \\ &= \sum_{i=1}^{m} p_i(iB_{\mathrm{c}} + (i+1)B_{\mathrm{s}} + iB_{\mathrm{req}} + (1-\sigma)B_{\mathrm{res}}) \end{aligned} \tag{3.7}$$

We evaluate this scenario with the parameters found in Table 3.2 (Scenario 2.1). The result can be found in Figure 3.3(a). The diagram compares network load for the client-server approach and the mobile agent approach for fixed request, reply, and result size while varying the number of servers m. It can be seen that only for a small number of servers, mobile agents produce less network traffic, because here data filtering and compression have a positive effect on the overall network load. Beyond a specific number of servers, mobile agents produce higher network load, because of the overhead of sending code and state information to each server. Thus, network load increases in proportion to the number of server in the mobile agent approach, whereas network load only increases slightly in the client-server approach.

We will now evaluate the same scenario in a network where only the costs at the client network interface are considered. For example, if the client is a mobile phone that has a GPRS connection to the Internet, costs depend on the number of bytes sent from the mobile phone to the Internet service provider – we denote this network connection as *uplink*. Of course, the network load of the client-server approach is identical to the one above (Equation 3.6). For the mobile agent approach we ignore all costs related to migrations between servers in the network in this scenario. Thus, the network load amounts to:

$$\begin{aligned} B_{\mathrm{MA}} &= p_1(B_{\mathrm{c}} + 2B_{\mathrm{s}} + B_{\mathrm{req}} + (1-\sigma)B_{\mathrm{res}}) + \ldots \\ &\quad p_m(B_{\mathrm{c}} + 2B_{\mathrm{s}} + B_{\mathrm{req}} + (1-\sigma)B_{\mathrm{res}}) \\ &= \sum_{i=1}^{m} p_i(B_{\mathrm{c}} + 2B_{\mathrm{s}} + B_{\mathrm{req}} + (1-\sigma)B_{\mathrm{res}}) \qquad (3.8) \\ &= B_{\mathrm{c}} + 2B_{\mathrm{s}} + B_{\mathrm{req}} + (1-\sigma)B_{\mathrm{res}} \qquad (3.9) \end{aligned}$$

Figure 3.3(b) shows that network load is much smaller now in the mobile agent approach because data filtering and compression has a positive effect. Network load is constant in the mobile agent approach, because only uplink costs are considered



(a) Network load between all nodes is considered. Mobile agents produce smaller network load, only if the number of server to be visited is small.

(b) Network load at the uplink is considered. Mobile agents always produce smaller network load.

Figure 3.3.: Evaluation of Scenario 2.1 and 2.2: Network load vs. number of servers.

and, therefore, it is irrelevant on which server the information is found, whereas it increases slightly in client-server approach. Please note that a scenario where only uplink costs are considered is the only case where for high number of servers mobile agents produce lower network load than client-server techniques – in all other scenarios, it is vice-versa: the higher the number of servers the higher the network load for mobile agents. Thus, it is very profitable to use mobile agents, if only uplink costs must be considered.

3.2.3. Scenario 3: Network of m Servers, Select Information at All Servers

In this scenario the client has to collect data items from all servers in the network, so that in any case all m servers are visited. In the client server approach, the client sequentially accesses each server, sending a request and receiving a server result. The total network load amounts to:

$$B_{\mathrm{CS}} = m(B_{\mathrm{req}} + B_{\mathrm{res}}) \tag{3.10}$$

In the mobile agent approach, the agent migrates from its home server to server L_1 with produces costs for code, state, and request transmission. On server L_1 the agent selects and filters data, so that the cost for the next migration to server L_2 increases by $(1-\sigma)B_{\mathrm{res}}$. At each succeeding server new data items of cost $(1-\sigma)B_{\mathrm{res}}$

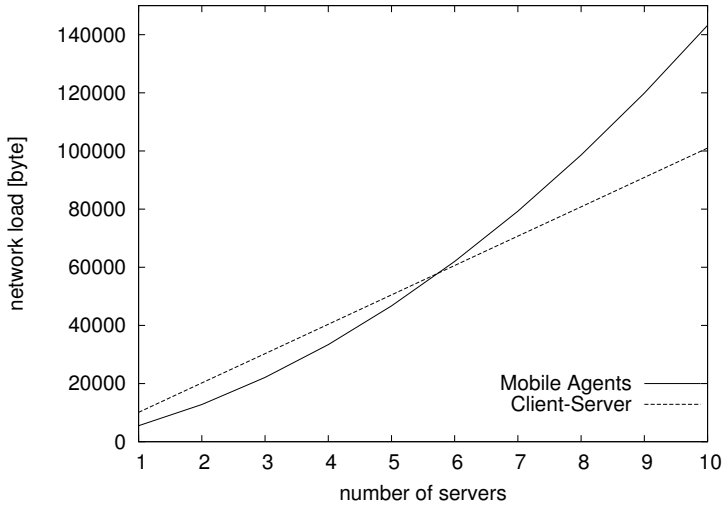


Figure 3.4.: Evaluation of Scenario 3.1: Network load vs. number of servers. Network load for mobile agents increases quadratically in the number of servers.

must be added, so that for the migration from server L_i to server $L_{\mathrm{i}+1}$ data items of size $i(1-\sigma)B_\mathrm{res}$ must be take along. In sum, network load for the mobile agent approach equals:

$$\begin{aligned} B_\mathrm{MA} &= B_\mathrm{c} + B_\mathrm{s} + B_\mathrm{req} + \\ &\quad + B_\mathrm{c} + B_\mathrm{s} + B_\mathrm{req} + (1-\sigma)B_\mathrm{res} + \\ &\quad + B_\mathrm{c} + B_\mathrm{s} + B_\mathrm{req} + 2(1-\sigma)B_\mathrm{res} + \ldots \\ &\quad + B_\mathrm{s} + m(1-\sigma)B_\mathrm{res} \\ &= mB_\mathrm{c} + (m+1)B_\mathrm{s} + mB_\mathrm{req} + \frac{m(m+1)}{2}(1-\sigma)B_\mathrm{res} \end{aligned} \tag{3.11}$$

We evaluate this scenario using the parameters given in Table 3.2 (Scenario 3.1). The result is shown in Figure 3.4. The diagram compares network load of the client-server approach with the mobile agent approach for fixed size of client requests and server results while varying the number of servers to be visited. Network load increases in proportion to the number of servers in the client server approach, whereas it grows quadratically in the mobile agent approach. The reason for this is that a mobile agent collects data items from *each* server and must carry all results. Mobile agents only produce lower network load when the number of servers to be visited is small.

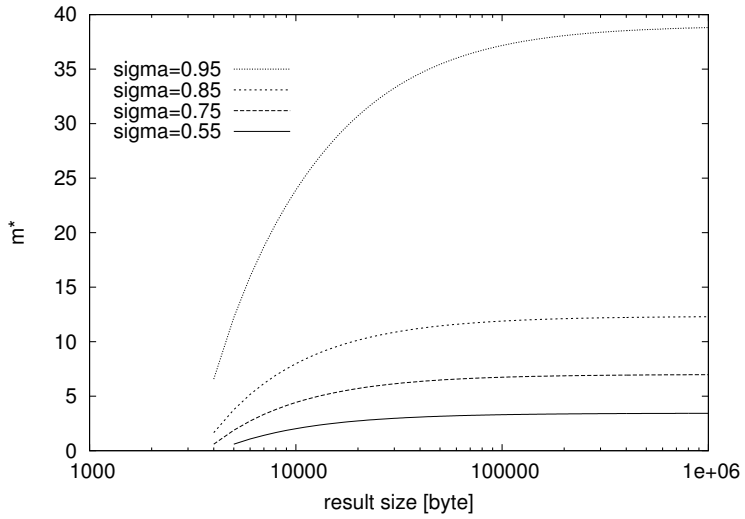


Figure 3.5.: Evaluation of Scenario 3.2: Result size vs. m^* for various compression factors. There is an upper bound for m^* for each value of σ.

As can be seen in Figure 3.4, there exists a number of servers for which both paradigms produce the same network load. We denote this number of server with m^*. We are now interested in how m^* changes while varying the server result size B_{res} and the compression factor σ. We evaluate this scenario using the parameters found in Table 3.2 (Scenario 3.2). Figure 3.5 shows the relation between m^* and the server result size for four different compression factors. It can be seen that the number of servers up to which mobile agents produce lower network load has an upper bound for each value of σ, which depends on the size of the server result. This upper bound is higher when the compression factor is higher.

Now, we will evaluate the same scenario as Scenario 3.1, but consider only costs at the client network interface. Network load for the client-server paradigm is the same as in Equation 3.10. The network load for the mobile agent approach equals:

$$\begin{aligned} B_{\text{MA}} &= B_{\text{c}} + B_{\text{s}} + B_{\text{req}} + \\ &\quad B_{\text{s}} + m(1-\sigma)B_{\text{res}} \\ &= B_{\text{c}} + B_{\text{req}} + 2B_{\text{s}} + m(1-\sigma)B_{\text{res}} \end{aligned} \tag{3.12}$$

We evaluate this scenario using the parameters found in Table 3.2 (Scenario 3.3). The result can be seen in Figure 3.6. As in Scenario 2.2 we can see that mobile agents produce much lower network load than client-server techniques due to data filtering and compression.

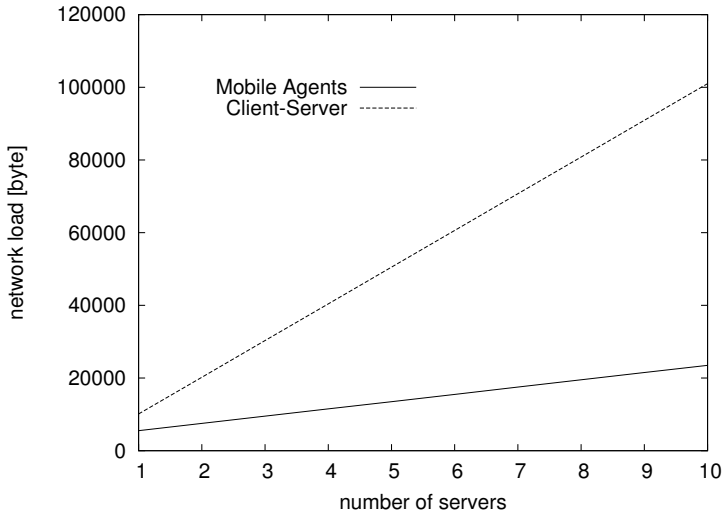


Figure 3.6.: Evaluation of Scenario 3.3: Number of computers vs. network load, only considering network load between client and any server.

At last, we will discuss the influence of network parameters on the response time in both approaches. We want to show that in a heterogeneous network it is not valuable to assess a paradigm solely on basis of network traffic, especially when the network connection between client and any server has lower bandwidth than inter-server connections. Therefore, we introduce $\delta : \mathcal{L} \times \mathcal{L} \rightarrow \mathbb{R}$, where $\delta(L_\mathrm{i}, L_\mathrm{j})$ describes the delay (latency) of a network connection between node L_i and node L_j and $\tau(L_:, L_\mathcal{L}) \times \mathcal{L} \rightarrow \mathbb{R}$, where $\tau(L_\mathrm{i}, L_\mathrm{j})$ describes the throughput between node L_i and node L_j. For the moment, we omit processing time. From Equation 3.10 we obtain the following equation for the response time:

$$T_\mathrm{CS} = \sum_{i=1}^{m} 2\delta(L_0, L_\mathrm{i}) + \frac{B_\mathrm{req} + B_\mathrm{res}}{\tau(L_0, L_\mathrm{i})} \tag{3.13}$$

We assume the client as node L_0. The execution time for a simple client-server call consist of the time for transferring request and result plus the delay for this network connection.

In the mobile agent approach, we can derive from Equation 3.11 the following equation:

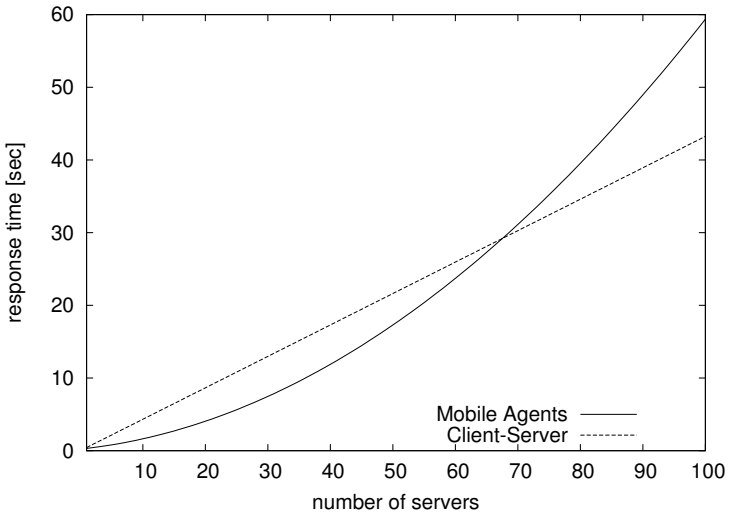


Figure 3.7.: Evaluation of Scenario 3.4: Response time vs. number of servers. The number of servers where network load of client-server and mobile agents are equal is higher than in Figure 3.4.

$$T_{\mathrm{MA}} = \left(\sum_{i=1}^{m} \delta(L_{\mathrm{i}-1}, L_{\mathrm{i}}) + \frac{B_{\mathrm{c}} + B_{\mathrm{s}} + B_{\mathrm{req}} + (i-1)(1-\sigma)B_{\mathrm{res}}}{\tau(L_{\mathrm{i}-1}, L_{\mathrm{i}})} \right) + \\ + \delta(L_0, L_{\mathrm{m}}) + \frac{B_{\mathrm{s}} + m(1-\sigma)B_{\mathrm{res}}}{\tau(L_0, L_{\mathrm{m}})} \tag{3.14}$$

We will now evaluate this scenario using the parameters given in Table 3.2 (Scenario 3.4). The result is shown in Figure 3.7. We assume delay time to be 90 ms between the client and each server, and 30 ms between a pair of servers. Throughput is assumed to be 40 000 byte/sec for the client-server link, and 200 000 byte/sec for inter-server connections.

The only difference in this evaluation compared to the evaluation of Scenario 3.1 is the heterogeneous network and we can see its effect inasmuch as the break-even point between client-server and mobile agents increased from about 5 to nearly 70 servers. Due to the fact that the slow link between the client and all servers is used very often in the client-server approach, response times are very high. However, beyond a certain number of servers, mobile agents become worse because of their characteristic to carry all data.

3.3. Discussion of Our Own Results And a Further Literature Review

In the last section we developed a mathematical model for network traffic and response time of client-server based approaches and mobile agents for three general application scenarios. Our evaluation of this model confirms the thesis that mobile agents produce less network load than client-server techniques only, if their code together with the state is smaller than the amount of network load that can be saved by their use.

The general result of our mathematical model is that within the same application scenario we can find both parameter constellations where smallest network traffic is achieved by either client-server or mobile agents. As a consequence, we must conclude that it depends on the values of several factors, as for example code size, server result size, number of servers to be visited, etc., which paradigm produces smallest network traffic resp. response time. In this difficult situation, software designers would surely benefit from any rule of thumb to decide, which paradigm should be used in a given situation. However, in our opinion, this does not make sense, as we abstract from several factors in our model for the sake of simplicity – and these parameters undoubtedly will also influence the decision and would invalidate these rules or make them at least inaccurate outside our model.

Limitations of Our Model and Other Approaches

Some factors that also influence network traffic and response time in real applications are for example those considering network quality. One important aspect is the error probability of a given network connection, especially in the case of wireless connections. A mathematical model including this parameter is for example presented by Jain et al. [2000]. Another factor we did not model is server processing time. As we ignored this factor, our model assumes response time to be equal to agent migration resp. data transmission time, which is of course inaccurate for all processing intensive tasks. However, to extend our model with processing time would make it indispensable to model server scalability too – and this would have been clearly out of the scope of our intention for the moment. Notwithstanding, we are aware of this factor, as Gray et al. [2001] pointed out recently that scalability of a mobile agent server software is a severe penalty for the overall performance of mobile agent based applications.

In contrast to our general mathematical model, where we tried to figure out the main parameters influencing the performance in both paradigms, some authors focus on parameters which are very specific for the examined application. For example, Puliafito et al. [1999, 2001] consider an example from the information retrieval domain where a fixed number of servers must be accessed. After the first request is

processed and the server result is sent back to the client, the client decides whether another request must be sent to the server. The authors analyze the influence of the probability p_r of reusing the same server for a subsequent request. Such a subsequent request produces network load and, therefore, transmission time in the client-server approach, whereas in the mobile agent approach only processing time must be added. The evaluation shows that with a low value for p_r, the client-server approach performs better, because of the overhead of mobile agent migration. However, with increasing probability p_r the client-server approach results in higher processing time than the mobile agents approach.

There are several other papers that also discuss trade-offs between client-server and mobile agent approaches, which we will not mention in detail, because they would contribute any new results: Spyrou et al. [2000], Outtagarts et al. [1999], Papastavrou et al. [1999], Knudsen [1995], Spalink et al. [2000], Theilmann and Rothermel [1999], Samaras et al. [1999].

Advantages of Mobile Agents

We evaluated our model using several scenarios and we identified the following limited advantages of mobile agents with a simple migration capability only. Mobile agents produce lower network traffic,

1. if the number of requests during a communication is high so that many data transmissions of the client-server paradigm can be avoided by using mobile agents, or

2. if the size of the server result is high combined with a high compression or filter factor, so that a much smaller number of bytes must be sent back to the client.

As already stated the concrete value of the threshold beyond which mobile agents produce lower network load depends on several factors. Therefore, we are unfortunately not able to be more precise at this point than to say that the number of request or the compression factor must be *high*.

It is obviously easier to give concrete numbers, if instead of a mathematical model real-world experiments are conducted. For example, Ismail and Hagimont [1999] present results of experiments with the Aglets mobile agent system and a client-server implementation based on Java RMI. The authors consider an example from the information retrieval domain consisting of a single client and two servers. The first server offers information about hotels and the second server is a telephone directory. The task is to get a list of hotels in a given town together with their telephone numbers. So, the client first requests from the hotel database a list of hotels, then selects those hotels in the given town by itself, because it is assumed that the interface of the first server does not offer any filter function. For each selected hotel, the second

server is asked for the hotel's telephone number. Measurements were down in a network of three computers residing in different European cities. The results show that the overall execution time depends on the number of hotel records returned by the first server. Below 30 records, the RMI based client-server approach has a lower execution time, whereas above this record number, the mobile agents approach performs better. The experiment shows that mobile agents are a good choice only if *many* data must be processed. Otherwise, mobile agents may produce more network load than client-server techniques.

The other two scenarios deal with cases where more than a single server must be accessed. In these cases, mobile agents work completely different as compared to client-server based techniques. Whereas in the latter approach the client accesses each server resulting in a star-shaped communication flow, mobile agents hop from server to server not visiting the client meanwhile. In Chapter 2.2.2 we identified especially this behavior as a major difference of mobile agents as compared to mobile objects. The main advantage of this behavior comes from the fact that the network connection between the client and any server is not used frequently by mobile agents, so that mobile agents produce

1. smaller network load, if only network traffic at the client-interface is considered, and

2. smaller response time, if the uplink has a smaller bandwidth (or higher latency) than all inter-server connections.

We investigated a scenario where only data transmission at the client interface, the so-called uplink, is considered. The result was that mobile agents have advantages, if the user is interested in minimizing uplink costs, as it is conceivable with having a GPRS mobile phone. In this case we do not consider all the network traffic that is produced between servers so that all these transmission costs for code and data transmission are not taken into account. Of course, this advantage can only exist in combination with one of the advantages mentioned above. For example, if an agent were not able to reduce the server result, its network load would only be smaller for a very high number of servers.

The second advantage deals with the case of heterogeneous networks, i.e. networks in which network connections do not have the same quality. We concentrated on those networks, where the uplink, i.e. the connections between the client and any server, has lower bandwidth or higher latency than connections between servers. A typical example for such a network consists of a mobile client that must use a wireless LAN connection with a bandwidth of 2-11 Mb/sec rather than Ethernet with 100 Mb/sec. In such a heterogeneous network, we showed that mobile agents have a smaller response time than client-server techniques. The reason for this is that mobile agents do need this small bandwidth connection only twice for the migration from

the client and, finally, back to the client, whereas in the client-server paradigm, each server is accessed using this bottleneck connection. In our evaluation we neglected to analyze the impact of network bandwidth and network latency on the performance of mobile agents in detail, because this has been done in the literature before:

Rubinstein and Duarte [1999] evaluate trade-offs of mobile agents in network management tasks. Performance of mobile agents is compared to an approach using SNMP. The network topology used in these simulations consist of a LAN of Managed Network Elements (MNE) connected to a management station by a bottleneck link. Simulations are made using a network simulation software. The application scenario the authors look at are simple network management task, like for example retrieving SNMP variables from all MNEs. The authors consider the network load at the bottleneck link and response time of a single task as performance parameters. The authors conduct several experiments, varying network latency and bandwidth, the initial mobile agent's size, and the number of bytes to select at each MNE. The result is that the performance of mobile agents does not change with the latency, because mobile agents only use the bottleneck link twice, whereas all SNMP messages must traverse the bottleneck link. Varying bandwidth of the bottleneck link has the result that for a small bandwidth both mobile agents and SNMP messages present larger response times. Here the authors found that for a lower number of MNEs, SNMP messages perform better, but with increasing number of MNEs, mobile agents have lower response times. Increasing mobile agents' size has the expected effect of higher response times, whereas response times for SNMP messages remain constant.

Drawbacks of Mobile Agents Using a Simple Migration Technique

We will now analyze the reasons why mobile agents sometimes produce higher network traffic. The first obvious reason is the size of the code that has to be transmitted to each server the agent visits. Agent code is usually larger than a simple client-server request, because agents do not only carry the code for data filtering and/or compression, but they also need additional logic implemented to decide which servers should be visited. Only in few applications, this order might be a fixed itinerary, implemented as a primitive array of URLs. In most applications the decision for the next server to visit is done dynamically during runtime. On the other hand, large code size is not in all cases a drawback. There is often a simple relation between the quality of filtering and/or compressing data at the server site, and agents' code size. Simply speaking, the more sophisticated the data filtering task is (which results in a higher compression factor), the larger the code is that is necessary to achieve this compression factor. Higher network traffic of mobile agents as compared to client-server techniques can be caused by situations where the achieved compression factor is low, although much code was sent to the remote server – and those situations are

not always avoidable.

A second drawback of the mobile agents paradigm is that sending code to each server causes a fixed overhead in network traffic for each agent migration. Therefore, before mobile agents can beat client-server based techniques, there can be several client-server interactions as long as they are in sum smaller than this fixed overhead. If the advantage of mobile agents comes from data compression or filtering than this data reduction must save at least network traffic in the size of agent's code.

A third drawback deals with the migration technique. One simplification of our model is that we do not consider different techniques for agent migration. We assume the widely used technique that sends code as one unit to each server. In some cases, this might include pieces of code with low execution probability on specific servers, i.e. it is improbable that these pieces of code will be executed on a specific server. As an example, just think of a task divided in several sub-tasks. On a specific server only one of these sub-tasks is executed, and, therefore, code for all other sub-tasks was transferred superfluously. Kotz and Gray [1999] describe this as following: "Thus, ..., mobile agents (especially those that need to perform only a few operations against each resource) often take longer to accomplish a task than more traditional implementations, since the time savings from avoiding intermediate network traffic is currently less than the time penalties from slower execution and the migration overhead.". We will come back to this drawback of mobile agents in Chapter 5, where we will extend our model by several migration techniques.

The last three drawbacks for mobile agents' bad performance were deduced from the first scenario (pp. 28) of our model evaluation, where only a simple interaction between a single client and a single server was examined.

If a mobile agent has to migrate to many servers instead of only a single one, we can find further drawbacks. First of all, it is clear that a higher number of servers to be visited does not have any positive impact on code migration. The agent's code must be transferred to each server that the agent visits, and, therefore, we could repeat all the drawbacks of mobile agents presented above even in the multi-server case. But also another drawback of mobile agents becomes obvious. If more than a single server must be visited, the result collected at server L_{i-1} is transferred as part of the mobile agent's data to server L_i. If the result of server L_{i-1} is not needed at any server $L_i, \ldots, L_m$, then it was superfluous to transfer it. The same is true for the other case, where data originally created at the client must be transferred to all servers $L_1, \ldots, L_{i-1}$, although server L_i is the first one which needs them, e.g. to create an appropriate request.

Our evaluation showed that mobile agents are only useful for a *small number of servers to be visited* – a fact that intuitively contradicts the general idea of mobile agents as multi-hop entities. Besides, we have shown mathematically that there exists an upper bound for the number of servers, beyond this bound mobile agents are unable to *ever* produce lower network traffic. For example, even in the case of a

very high compression factor ($\sigma = .95$), mobile agents are better than client-server, only if less than 40 servers are to be visited.

3.4. Summary of Part I

In the first part of this thesis we introduced the concept of mobile agents and compared it to more traditional design paradigms for distributed applications. We gave a brief description of the history of mobile agents, presented several application domain where mobile agents seem to be useful, and discussed the technical advantages of mobile agents as compared to the client-server model.

In a first mathematical evaluation we examined a performance analysis of simple mobile agents as compared to the client-server approach. The evaluation showed that mobile agents have the possibility to reduce the network load and processing time as compared to client-server based applications, because code is shipped to the data, instead of shipping data to the code. However, we also showed some severe drawbacks of mobile agents, features which are in sum responsible for higher network load and longer processing time in certain situations.

All these drawbacks are caused by the simple migration technique used in our mathematical model, which, however, reflect the current state-of-the-art in agent migration. We learned that code migration is a very expensive task that must be optimized and has to become more flexible. We also learned that data migration has an important impact on the performance of mobile agents. It is our thesis that *the migration process of mobile agents must be optimized to let them migrate in a more flexible and fine-grained way.* What we mean is that a mobile agent should not always migrate as one unit that consist of all code, state, and data information, but that it is sometimes useful to let the agent decide which code and data item should be transferred to the next server. In the next part we will, therefore, focus on the migration process of mobile agents and reason about possible optimizations.

Part II.

Analysis and Synthesis of Mobile Agent Migration Techniques

The second part of this thesis focuses on the migration aspect of mobile agents. We start in Chapter 4 with an introduction into the migration process, describe how the Java language supports building mobile agents, and discuss performance aspects of agent migration. In Chapter 5 we will propose a classification scheme for agent migration. We will show that the process of transferring an agent from one computer to another can be implemented in very different ways, so that the software designer of a mobile agent system has to make several design decisions or choose between several design alternatives. In Chapter 6 we will reason about performance issues of mobile agents and introduce the new Kalong migration technique.

4. The Mobile Agent Migration Process

After looking at mobile agents from the application point of view, we will now focus on the migration process, which we identified to be one of the most crucial aspects for high-performance mobile agents. So far we explained that agent migration simply as the process of transferring a mobile agent from one computer system to another, without going into technical details. In this chapter we will catch up on this issue.

We will start with a generic framework for an agent migration process that matches almost all current implementations. This framework highlights some of the most important technical requirements for a migration component. After that, we will introduce the Java programming language in Section 4.2 with its features to support the migration process and describe the migration process of the Tracy mobile agent system.

4.1. Generic Framework for Agent Migration

The process of agent migration, although implemented in each mobile agent system differently, can be described by a general framework. Introducing this framework also helps us to unequivocally define some terms, which we will use in the next chapters and the rest of this thesis.

A mobile agent is a software program that is in most systems executed as part of a so-called mobile agent server software. This server software controls execution of agents and provides some basic functionality for agent communication, agent control, security, and migration.[1] This mobile agent server is called *agency* in the following. On each computer system that wants to host mobile agents, an agency of the same type must be installed.[2] All agencies that are able to exchange mobile agents form a logical network that we call the *mobile agent system*. Each computer system can host several agencies in parallel and each agency is reachable by at least one URL

[1] Of course, it is possible to build a system of (mobile) agents by just letting an agent be a *process* in the meaning of operating systems. Processes can communicate with each other by primitives offered by the operating system and even migration can be achieved with special *distributed* operating systems or can be provided as the only service of the underlying mobile agent server software. Tacoma [Johansen et al., 1995] is an example for such a system.

[2] Recently, some research groups started to develop methods to make mobile agents interoperable, i.e. that two different agencies are able to exchange agents. See for example Pinsdorf and Roth [2002] for more information.

address to which migration is directed to. The URL also serves as a name of the agency. For this moment it is not interesting, how the agency is structured, e.g. some mobile agent systems subdivide a single agency into several *places*. Each place is a closed area and agents in different places neither know or see each other, nor can communicate with each other.

As any software program, mobile agents are written in some programming language. As already described in Section 2.2.3, any programming language can be used in principle for implementing mobile agents. In most systems there is a restriction inasmuch as the same programming language must be used for mobile agents as was used for the whole mobile agent system. Only few systems, e.g. Tacoma and D'Agents, allow agents on the same system to be implemented using different programming languages. The first mobile agent systems had mobile agents implemented in script language, as for example Telescript, TCL, or Perl. Current mobile agent systems use the Java programming language, because of its many features that lessen the effort for building mobile agent systems.

When an agent is started on an agency, this one becomes the agent's *home agency*. The principal who starts the agent is called the *agent's owner* and the owner also defines the *agent's name*. The owner information is important to decide in foreign agencies how trustworthy the agent is. The agent's name is important to identify an agent unequivocally on all agencies of the mobile agent system. All this information about an agent's home agency, agent's owner, and agent's name become attributes of the agent. Usually, an agent returns to its home agency after it has fulfilled the given task. The other important agency is the one that holds the code of the agent, we call this one a *code server*. Usually, the home agency equals the code server, but this is not a must.

Agencies are typically multi-agent systems, i.e. a single agency can host many agents in parallel. To provide quasi-parallel execution, some kind of scheduling is offered. In most systems this process of scheduling is not programmed within the server software, but is delegated to the programming language resp. the operating system. For example, a very common case is that each agent owns a thread[3]. During execution, the agent is allowed to start new child threads, of course.

Mobile agents consist of three components: *code*, *data*, and *execution state*. The code contains the logic of the agent and all agents of the same type use the same code[4]. The code must be separated from the code of the agency, so that it can be transferred without the code for the agency to another one, and the code must be identifiable and readable for the agency, e.g. in form of a file from the local file system

[3] Being a thread less than a process, see Tanenbaum [2001] for more information.

[4] Here, we have a rather pragmatic and narrow notion of *type* for agents: Two agents are of the same type, if they use the same code. More programming language like definitions would refer to the interface or the communication protocol the agent offers, see Zapf and Geihs [2000] for a detailed discussion on other approaches for defining the notion of a type for agents.

or byte stream from the network. Usually, as in other programs too, an agent's code consists of more than one file, e.g. many class files in the Java programming language.

The second component of an agent is *data*. This term corresponds to the values of the agent's instance variables, if we assume an agent to be an instance of a class in object-oriented languages. The data are sometimes also called the *object state*. It is important to note that not all data items an agent can access are part of its object state. Some variables reference objects that are shared with other agents or the agency software itself, for example file handlers, threads, the graphical user interface, or other resources and devices that are not movable to other servers. Thus, we have to restrict the agent's immediate data to those data items the agent owns and that are movable. Problems arising from the fact of non-movable resources are discussed in the next chapter.

The third component is the *execution state*, which comprises of the program counter, frame stack, and all other information that is necessary to resume execution at the remote agency. What this means, depends very much on the decision of the mobile agent designer and the underlying execution environment (processor, operating system, virtual machine), as we will see in the next chapter. For the moment, we can state that the execution state contains the current value of the instruction pointer and the stack of the underlying processor. The difference between object and execution state information is that the elements of the object state are directly controlled by the agent itself, whereas execution state information is usually controlled by the processor resp. operating system.

The typical behavior of a *mobile* agent is to *migrate* from one agency to another from time to time. During the process of migration, the *current agency*, i.e. the one the agent currently resides on, is called the *sender agency* and the other agency, to which the agent wants to migrate to, is called the *receiver agency*. During the migration process the sender and the receiver must communicate over the network and exchange data about the agent that wants to migrate. Thus, we can say, that some kind of communication protocol is driven, and we call this the *migration protocol*. Some systems simplify this task to an asynchronous communication, comparable to sending an electronic mail, whereas other systems develop rather complicated network protocols on top of TCP/IP.

The whole migration process contains six steps, which are executed in sequence, except of S3 and R1 which are executed in parallel. Please refer to Figure 4.1.

The first three steps (S1-S3) are executed on the sender agency:

S1 *Initialize the migration process and suspend thread.* The process of migration typically starts with a special command, the *migration command*, by which the agent announces its intention to migrate to another agency, whose name is given as parameter of the migration command. The first task for the agency is now to suspend the execution thread of the agent and to guarantee that no

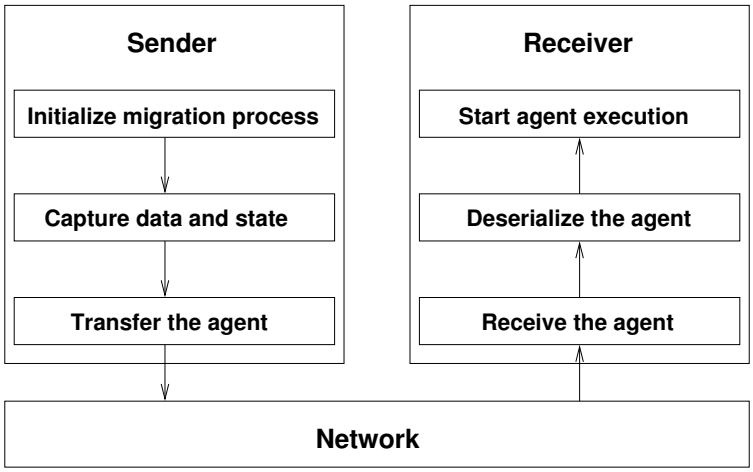


Figure 4.1.: The mobile agent migration process.

other child thread is still alive. This requirement is important for the next step, where it is imperative that data and state are frozen and cannot be modified later on.

S2 *Capture agent's data and execution state.* The current state of all variables (the data) of the agent is *serialized*, i.e. their current values are written to an external persistent representation, e.g. a memory block or a file. The agent's state is also stored there, so that the point of suspension is known. Result of the serialization process is the *serialized agent* which is a flat byte stream that consists of the agent's data and state information.

S3 *Transfer the agent.* The serialized agent is transferred to the receiver agency using a migration protocol. Whether any code is sent to the receiver agency depends on different parameters and will be discussed later.

The last three steps (R1-R3) are executed on the receiver agency.

R1 *Receive the agent.* The serialized agent is received using the migration protocol. The receiver agency checks whether the agent can be accepted based on information about the agent's owner and the sender agency. The receiver agency may filter out agents that come from agencies that are unknown or not trusted.

R2 *Deserialize the agent.* The serialized agent is deserialized, i.e. the variables and the execution state are restored from the serialized agent. The result of this

step should be an exact copy of the agent that existed on the sender agency just before reaching the migration command.

R3 *Start agent execution in new thread.* The receiver agency resumes agent execution by starting a new thread of control. At least when resuming execution, the agent's code is needed. In this general framework we make no assumption about how the code is transferred to the receiver agency. One possible technique is for example that the receiver agency loads the code from the agent's home agency or its code server. We will discuss those techniques in the next chapter.

In the next section we will now look at an implementation of a migration process. As an example we chose our Tracy mobile agent system, which is implemented in the Java programming language.

4.2. Migration in the Tracy Mobile Agent System

In this section we will describe the migration process of an existing mobile agent system, which was developed using the Java programming language. We will follow the generic framework introduced in the last section and explain the advantages of Java for programming mobile agent systems and mobile agents. We chose the Tracy system as an example here, not only because Tracy is the result of our own research, but also because the complexity of the Tracy migration process lies in the middle between a very simple one, e.g. used in Semoa [Roth and Jalali, 2001] and a difficult one, e.g. used in Aglets [Lange and Ishima, 1998]. A detailed introduction into the Tracy mobile agent system would be out of the scope of this thesis and is, therefore, postponed to the appendix, see pp. 267.

The Java programming language [Arnold and Gosling, 2000] has become the defacto standard programming language for mobile agents[5]. All systems, developed in the last four years, are using this programming language for the mobile agent system as well as for the mobile agents. Even both projects mentioned in the last section that do not solely support Java, cannot be named as opponents of this language at all, because in these projects one of the main research issues is multi-language support and both do support Java.

The advantage of Java comes from several built-in features that lessen the effort for building mobile agent systems. In this section we will especially focus on features that support the migration process, e.g. object serialization, dynamic class loading,

[5]For some time, the question which languages are suitable for mobile agents was topic of intensive research in the community. For a detailed discussion about the language requirements for mobile agents, we refer to the dissertation thesis of Knabe [1995], and the following papers [Cugola et al., 1997a,b; Knabe, 1997b; Thorn, 1997].

and reflection. We will also briefly mention foundations of the security architecture of Java. Despite of these advantages, some aspects of Java are also imperfect with regard to the requirements of mobile agent systems – we will also briefly discuss these drawbacks.

4.2.1. Foundations of Java

Java is an object-oriented language developed by Sun, Inc. Although the original project goal was simply to develop a new programming language (Oak) for a new kind of remote control with LC display and touchscreen (named "*7"), Java became *the* Internet language since 1995. For some time, the most famous application domain for Java were applets that are shipped from a Web server to a Web browser. Today, due to major performance improvements achieved in the last years, Java has become a widely used programming language for server-based applications, too.

The most important feature that made Java an Internet programming language is its *portability*. Java programs are compiled into a architecture independent byte code format [Lindholm and Yellin, 1999] which is executed using a Java virtual machine. As virtual machines exist for almost all current hardware platforms and operating systems, Java programs have the enormous advantage of being executable on almost all existing computer systems. Portability is a very important requirement for mobile agent systems, because mobile agents must be able to migrate in a network of heterogeneous computer systems.

A consequence of the architecture independent byte code format is that Java is an interpreted language. The byte code is executed by the virtual machine, which completely protects the underlying operating system from direct access out of Java programs. This simplifies security control, because an intermediate code format allows easier code inspections for security violations than compiled native code. As all interpreted languages, Java has a lower execution performance than compiled code. However, very sophisticated techniques were developed for Java to translate intermediate code into optimized native code during execution (just-in-time compilation and hot-spot optimization).

The language itself supports development of *safe* applications, because, in contrast to C for example, Java has a pointer model that does not support pointer arithmetic and illegal type casting. The *byte code verifier*, a component of the virtual machine, filters out code before execution that violates basic semantics of Java. Even during runtime, a security manager controls all potentially unsafe operations, as for example file access, network connections, or access to the graphical user interface. It is dynamically determined, whether the given program is permitted to perform these operations.

Java comes with many libraries, e.g. for data structures, network programming, graphical user interfaces, etc. Especially network programming is supported using

sockets as well as using remote method invocation (RMI), which is the object-oriented version of the remote procedure call concept. Java RMI is so powerful that implementing a very simple mobile agent system can be done in less than 100 lines of code, compare for example Avvenuti and Vecchio [2000].

Unfortunately, Java also has some drawbacks with regard to mobile agent systems. The main disadvantage is perhaps the fact that it is impossible to obtain the current execution state of a thread in form of the current instruction pointer and calling stack, so that it is practically impossible to preserve and later resume execution of a mobile agent in detail. Therefore, as we will see later, Java based mobile agents can only offer a weak form of mobility, where the agent is restarted at the receiver agency by invoking a method (instead of jumping into it). Another drawback is the lack of resource control, e.g. for memory or processor cycles. Therefore, it is not possible to avoid denial of service attacks, which is a specific type of security attack, where the attacker tries to consume so much resources that the system is not able to handle incoming requests anymore.

4.2.2. Representing Agents in Tracy

In Tracy, an agent is an object of a specific class, named *Agent*. Tracy distinguishes between three types of agents, namely stationary system agents, stationary gateway agents, and mobile agents. The corresponding classes are *SystemAgent*, *GatewayAgent*, and *MobileAgent*, which are all direct subclasses of class *Agent* and are all member of package *de.unijena.tracy.agent*.

Class *Agent* is the main class within the TracyAPI. It is an abstract class that serves as base for all agents and must not be instantiated by the programmer. Class *Agent* defines methods and variables to control an agent's life-cycle, get and set internal data structures, and receive messages. Some of theses methods are useful for the programmer, e.g. methods to inform about the current agency, whereas some methods are only useful for an agency to control the agent. Class *Agent* also defines some methods that are supposed to be overridden by subclasses.

To give a practical example, we subclass *MobileAgent*, although we have not introduced class *MobileAgent*, yet. In the following example we will only refer to agent functionality that is common to all agent classes. To define an agent in Tracy, package *de.unijena.tracy.agent* must be imported, which includes all basic definitions.

```
1 import de.unijena.tracy.agent.*;
2
3 public class MyFirstTracyAgent extends MobileAgent
4 {
5   SomeOtherClass other = new SomeOtherClass();
6
7   public MyFirstTracyAgent()
```

```
8   {
9     // do some initialization
10  }
11
12  public void startAgent()
13  {
14    // do something
15  }
16 }
```

Method *startAgent* is defined abstract in class *Agent* and even no direct subclass does provide an implementation for it. This method is the entry point that is called at the agent's home agency to start the agent and, therefore, every user-defined agent must implement this method.

Usually, an agent consists of more than one class. In the example above, we see that this agent has a variable named *other* of type *SomeOtherClass*.

4.2.3. The Migration Process

In the following we will explain the migration process as implemented in Tracy according to the framework introduced in the last section. We do not follow the sequence introduced in the framework, but combine tasks that belong together, for example S1 and R3, etc. We omit to introduce the network transmission task here. Tracy uses its own migration protocol, called SATP, which is an asynchronous network protocol that bases on the TCP/IP protocol. We introduce this protocol later in Chapter 7.

Starting the Migration Process And Resuming Execution

In this section we will show how a mobile agent can start a migration to a receiver agency and how execution is resumed at the receiver agency. Actually, there are two ways to move a mobile agent. The first way is to use the *go* command to initiate migration with a default migration behavior to a single remote agency, and the other way is to use so-called *migration properties* to configure the migration process in detail, e.g. to define a complete itinerary. Then, the next *go* command automatically chooses the next destination in the given itinerary. In this section, we will only concentrate on the standard migration technique, using migration properties is explained in the appendix.

An agent migration is initiated by calling a method named *go* with the name of the receiver agency as the first parameter and the name of the method to invoke after migration as the second parameter. Method *go* is defined in class *MobileAgent* and cannot be overridden (defined **final** there) and should not be overloaded by the agent itself.

final protected void *go*(*URL destination*, *String methodName*)
Migrates an agent to the receiver agency `destination` and restart it by invoking method `methodName`.

The name of the receiver agency is just a URL, where the protocol should be *tracy* and the host name is the name of the receiver agency. Usually, a port number is also required, so that for example a complete destination address is *tracy://tatjana.cs.uni-jena.de:4040*. Calling method *go* stops agent execution immediately and statements following the *go* invocation will never be executed, neither in the case of a successful migration, nor in the case of a migration error. A *go* command might be included in a **try** ... **catch** clause. In this case, neither the Tracy-defined runtime exception *AgentExecutionException*, nor any super classes of this exception, must be caught.

```
1  try
2  {
3    // some code that might throw an IOException
4
5    System.out.println("Running on server \"tatjana.cs.uni-jena.de\"");
6    go( "tracy://domino.cs.uni-jena.de:4040", "runAtRemote" );
7
8    // statements below will never be executed
9    System.out.println("This message will never be seen.");
10 }
11 catch( IOException e )
12 {
13   System.err.println( e.getMessage() );
14   e.printStackTrace();
15 }
```

There are two other methods that are shortcuts for the above mentioned *go* method:

final protected void *go_home*(*String methodName*)
Migrates an agent to its home agency and restarts it by invoking the method with name `methodName`.

final protected void *go_back*(*String methodName*)
Migrates an agent to the agency it came from and restarts it by invoking the method with name `methodName`.

In the case that migration is not successful, e.g. because the receiver agency does not accept agents from the current agency or both agencies use different versions of the migration protocol, the agent must be reactivated at the current agency. Therefore, method *migrationFailed* is called, which is defined empty in class *MobileAgent*.

protected void *migrationFailed*()
Is called in case of any migration error.

The default behavior of this method is to do nothing, which lets the agent wait for new messages to become active again. Usually, this method should be overridden and could for example try to migrate again.

Resuming agent execution at the receiver agency is done using the Java *reflection* technique. After the agent was deserialized and at least the agent's main class is at the receiver agency so that the agent object can be successfully instantiated, agent execution is resumed. In Tracy the agent is resumed by starting a method whose name was given as second parameter in the *go*-statement. The name of this method was transmitted as part of the state of the agent.

Java reflection is a powerful technique to determine information about classes, their variables and methods during runtime. Additionally, it is possible to invoke a method of an arbitrary object only by having its name in a *String* variable. In the following example, we show an extract from the Tracy source code, where for a given mobile agent object a method with name *methodName* is invoked.

```
1 import java.lang.reflect.Method;
2
3 protected void startAgent( MobileAgent mobileAgent,
4                            String methodName ) throws Exception
5 {
6     Method method = mobileAgent.getClass().getMethod( methodName, new Class
          [] {} );
7     method.invoke( mobileAgent, new Object[] {} );
8 }
```

In line 6, we first determine the class name of the given mobile agent object, then ask this class for a method of name *methodName*. The second parameter of method *getMethod* contains an array of types that the wanted method must accept as parameter. An empty array as in the example indicates that the method should accept no parameters. If the agent's class has such a method, it is stored in variable *method*. In the other case, an exception is thrown. In line 7 this method is invoked with the mobile agent as parameter and an empty array of objects, which means that this method has no parameter at all. Method *startAgent* is called within a new thread that is assigned to the agent.

Object Serialization And Deserialization

After a mobile agent has indicated to migrate to another agency, serialization of the agent takes place. Serialization means that all variables of the agent, together with

all recursively referenced objects and their variables are traversed and put into a flat byte array. The set of all objects to be serialized is called *object closure*.

In the current version of Tracy we use the standard object serialization technique that is already implemented in Java [Sun, 1999]. To use this technique, each class whose objects should be serialized during their life-time must implement interface *java.io.Serializable*. Class *MobileAgent* already implements this interface, so that all mobile agents in Tracy can be serialized. Additionally, all variables that a mobile agent class defines, must be either marked serializable too, or marked as *transient*, which means that this variable is not an element of the object closure. If a not serializable object is found during the serialization process an exception will be thrown. Note, that class variables, i.e. those marked as **static**, are not part of the serialized object. Thus, the object will probably retrieve different values of class variables at the destination agency.

Java object serialization does only determine the object state of an agent and not its execution state. In Tracy only the name of the method that should be invoked at the receiver agency is part of the agent's state.

The following extract from the Tracy source code gives an impression of how simple the task of object serialization is in Java. Method *serializeAgent* gets a reference to the mobile agent as parameter and returns the serialized agent as byte array. In case of an error, the method returns **null**.

```
1  private byte[] serializeAgent( MobileAgent mobileAgent )
2  {
3    ByteArrayOutputStream baos = new ByteArrayOutputStream();
4    try
5    {
6      ObjectOutput oos = new ObjectOutputStream( baos );
7      oos.writeObject( mobileAgent );
8      oos.flush();
9    }
10   catch( IOException e )
11   {
12     return null;
13   }
14
15   return baos.toByteArray();
16 }
```

The most important statement is line 7, where the agent is serialized to an output stream. In line 15 this output stream is converted into a flat byte array.

The result of the serialization process is the so-called *serialized agent*, which is now transferred to the receiver agency. To instantiate a new mobile agent from a byte

array, is only a little bit more complicated. The standard Java object serialization technique allows to instantiate a new object simply from a byte array containing the serialized object. This procedure is correct, because the serialized object contains all information about used classes that are necessary to define and initialize the object correctly.

```
1  private MobileAgent deserializeAgent( byte[] bytestream )
2  {
3    try
4    {
5      ByteArrayInputStream bais = new ByteArrayInputStream( bytestream );
6      ObjectInputStream ois = new TracyObjectInputStream(
7                                          new TracyClassLoader( this ),
8                                          bais );
9
10     mobileAgent = (MobileAgent)ois.readObject();
11     return mobileAgent;
12   }
13   catch( Exception e )
14   {
15     return null;
16   }
17 }
```

This method uses two Tracy-specific new classes to instantiate mobile agents. Class *TracyObjectInputStream* subclasses the standard Java class *ObjectInputStream* which is responsible to conduct the complete deserialization process. If a class must be loaded during deserialization, then method *resolveClass* is called. In class *TracyInputStream* this method is overridden, so that our new class *TracyClassLoader* is used for this task. If the process of deserialization fails, value **null** is returned.

Finding Classes and Dynamic Class Loading

The default migration technique of Tracy transfers the serialized agent together with the state information, which only comprises of the method name to invoke at the destination, and all classes the agent might ever use to the receiver agency.

Therefore, the first problem before an agent can migrate is to determine which classes must be transferred to the next agency. Comparable to the definition of *object closure*, we can also define *code closure* as following: The code closure consists of the agent's main class, e.g. *MyFirstTracyAgent* in the example above and all classes that are used for variables, method parameters, method return values, and local variables of any class of the code closure. Unfortunately, Java does not provide an

easy way to determine this set of class names automatically. Class *Class* of the Java API provides method *getDeclaredClasses* which only returns an array of classes that are used for member variables. Using method *getDeclaredMethods* we can obtain information about all methods and using this information we can determine classes that are used for parameters and return values. But, Java does not provide a method to determine information about local variables defined within methods. Therefore, we implemented our own technique that looks at the byte code of the class. There, we find the *constant pool* [Lindholm and Yellin, 1999] which is a table that contains all class names that are ever used in this class. To read this table we use a tool named ByCal (byte code analyzer) which was developed within our project. This tool offers several services to analyze Java byte code, transform it, and even perform sophisticated control and data flow analyzes on it.

Obviously, the code closure should not contain classes that are not to be transferred to the receiver agency, because they can be assumed to be already there, e.g. classes of the Java API or classes that are part of Tracy. Before collecting the byte code for each class of the code closure, those classes are deleted from the code closure. To read the byte code for each class that is element of the code closure, Java does not provide a simple technique, too. Therefore, we search for the class in all directories and Java archive files that are defined in the environment variable *CLASSPATH*.

After the agent is received at the destination agency, its code must be linked to the code of the already running agency. This is another advantage of the Java programming language that allows dynamic class loading and linking. This mechanism allows the virtual machine to load and define classes at runtime.

In Java, an instance of class *ClassLoader* is responsible to load and define classes. Each class loader defines an own name space, so that different classes with the same name can be loaded into a single virtual machine without conflicts. The default class loader that is used unless the user specifies to use an own one, loads classes from the local file system, i.e. from directories or from Java archives that are listed in the *CLASSPATH*. Each object knows the class loader it was created by and a single class loader has usually created many objects, e.g., if an object creates new objects, the same class loader is used.

When a class loader has to load a class, it looks for the corresponding class code according to the following rules:

1. First, the class loader checks, whether the class has been already loaded before and gets the byte code from the cache in this case.

2. Then, the system class loader is asked to load the class from the system Java JAR file.

3. Then, the class loader looks at the *CLASSPATH* variable and searches all directories and Java archives.

4. Finally, it delegates the task to a user-defined class loader by invoking method *findClass*.

When the class code is found, it is defined by calling a special method *defineClass* of the class loader. User-defined class loaders must override method *findClass* and can load the byte code for the given class name for example, by either using a HTTP or FTP server, or using any other technique. Due to the default Tracy migration technique, all class files are already at the destination agency, so that the *TracyClassLoader* has only to look for the byte code in a local repository, where all incoming classes are stored.

5. Design Issues of Agent Migration

After we introduced technical details of the migration process and showed how the migration process is implemented in the Tracy mobile agent system, we will now discuss other approaches to implement mobile agent migration. The main goal of this chapter is to collect the design issues and to reason about design alternatives of agent migration. We introduce the term *mobility model* to describe the migration technique of a specific mobile agent system and we propose a language to describe mobility models. As an example, we describe the mobility models of two existing mobile agent systems, Aglets and Grasshopper, using our new language. The chapter concludes with a brief review of other approaches to classify mobile agent migration techniques.

In this chapter, we will not consider other design issues than those that are related to agent mobility. Of course, there are many other issues a designer must take into consideration when implementing a mobile agent system, as for example agent naming, agent communication, security, monitoring, fault-tolerance, etc. We know that there are several interdependences between all these issues and we agree that it would be very useful to propose a complete list of design issues for mobile agent systems. Some approaches were done in this direction, for example by Picco [1998], Hammer and Aerts [1998], and Karnik and Tripathi [1998]. However, in this thesis we will focus only on mobility issues. The very important relation between agent migration and agent security is discussed in a later chapter.

5.1. Mobility Models

The migration technique of Tracy that we have introduced in the last chapter was only one option how migration could be implemented in mobile agent systems. Actually, all mobile agent systems have implemented their own migration technique and differences can be found in all phases of our generic framework for agent migration. We already mentioned some of them in the last chapter, as for example other migration initiation commands than *go*, different understandings of the elements of an agent's state, other techniques to relocate classes, and different migration protocols.

The goal of this section is to gather information about other migration techniques in order to

1. discuss the design issues and design alternatives for agent migration, show pros

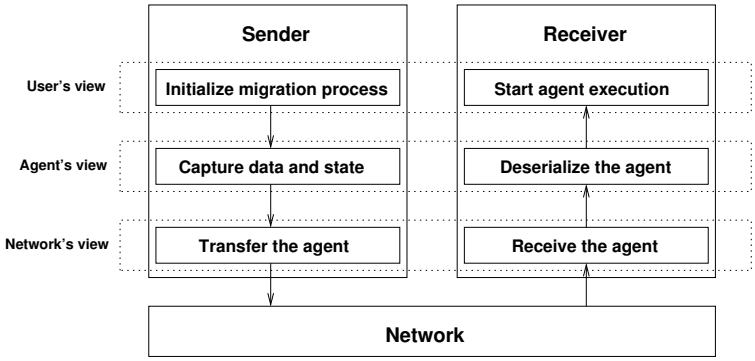


Figure 5.1.: The migration process and the three levels of our mobility model.

and cons, and discover dependences between different design issues,

2. develop a language to describe the migration technique of an existing mobile agent system.

The first point is important for a designer of a new mobile agent system, because he has to decide which migration technique the new system should provide. Using the results of the discussion in this chapter, he will be able to approximate the effort for each design alternative and then weigh between them. The second aspect is to describe the migration technique of an existing mobile agent system in a unified way, which makes it easy to compare different approaches.

To structure our discussion we introduce the term *mobility model*. The mobility model of a mobile agent system describes almost all important features concerning agent migration. A mobility model defines three views on migration issues and each view corresponds to two phases of our six-phase generic framework presented in the last chapter, see Figure 5.1.

1. User's view: How to initiate agent migration?
2. Agent's view: How is data and code relocated in the network?
3. Network's view: How is data and code transferred over the network?

The first view focuses on all issues that are related to the user interface of agent migration (phases S1 and R3). Design issues are here for example the migration command, the technique to resume agent execution, and how the receiver agency can be addressed. The second view focuses on the technique for data and code relocation (phases S2 and R2). Here, we introduce the term *migration strategy*, which describes

which pieces of code migrate to which other agency and how data is handled in each case. The last view focuses on the technique to transmit data over the network (phases S3 and R1). We already introduced the term *migration protocol* which is an important part of this view.

To describe a mobility model, we propose a language, named *Mobility Language* (MoL), for which we will give a definition using *Extended Backus-Naur Form* (EBNF). A description of a mobility model consists of several lines, where each line comprises of a key-value pair. The key is the name of a design issue and the value is either a single design alternative or a collection of design alternatives, separated by a semi-colon, that were chosen for the mobile agent system to be described.

For example, the following excerpt of a description in MoL defines that an *AgencyName* consists of a *SymbolicName* together with a *HostName*. The second line defines that this agency can be addressed using either a *Protocol* and the *SymbolicName*, or a *Protocol*, the *SymoblicName*, and a *PortNumber*.

```
1 AgencyName = SymbolicName + HostName .
2 AgencyAddress = Protocol + SymbolicName ;
3                 Protocol + SymbolicName + PortNumber .
```

To describe the grammar of MoL, we use EBNF. We will write <MobilityModel> to denote a design issue (non terminal symbol) and "weak-migration" for a design alternative (terminal symbol). We use the following meta symbols: A sequence of symbols is separated by +, alternatives are printed using brackets [... | ...], repetitions with at least one element are printed using braces { ... }, and finally an optional symbol is marked with parenthesis (...). So, to start our definition of MoL, we define that a mobility model consists of three views, where each view must be defined.

1. <MobilityModel> ::= <User> + <Agent> + <Network>

In the following three sections, we will discuss each view of a mobility model in detail.

5.1.1. Programmers' View

The first view is considered with all aspects of migration that are related to the agent programmer.

2. <User> ::= <Naming> + <Creating> + <Code> + <Data> + <Migration>

Naming And Addressing

The first design issue <Naming> considers the aspects of the agency name and the types of addresses that can be used for agent migration.

3. <Naming> ::= <AgencyName> + <AgencyAddress>

The aspect of *agent naming* is not considered in our description, because it is part of the design issues for the whole mobile agent system. Each agent has a name to be identified during its life-time. The name is used for example by other agents to set up a communication channel and is used by the agency itself to control the agent. In some mobile agent systems, the name is also used for agent tracking and for remote communication between agents currently residing at different agencies. Therefore, the agent name must be unique and immutable for the agent's whole lifetime. The structure or scheme for an agent name can be very different. Besides a symbolic name, e.g. *Blofeld*, it usually contains also the name of the home agency and the name of the underlying host system to make the name globally unique. The symbolic name can be explicitly given by the agent's owner in form of an easy to read name, or implicitly computed by the agency, e.g. as a digest of the agent's code together with some random to make the name unique. In the latter case, an human-readable alias name might be provided by the agent's owner. Concerning migration it is not important how an agent's name is structured or obtained as long, as it is guaranteed that each name is globally unique. For successful migration the agent name is very important, as the constraint that no two agents with the same name must exist can be best validated during the migration process. Therefore, the agent's name is an element of the migration protocol and the receiver agency checks whether there is an agent with the given name already registered with the agency.

More important with regard to agent migration is the structure of the agency's name. Each agency must have a name to be identified whenever the agency itself or any resources of it must be addressed. If the mobile agent system only allows a single agency on each host, then it is sufficient to have the "Hostname" as the agency name, e.g. *tatjana.cs.uni-jena.de*. For some application areas it might be more convenient to have more than a single agency on each host, so that each agency must have a "SymbolicName", e.g. *fortknox*. If an agency is further structured into several smaller units, sometimes called places, then also each place must have a name, which becomes part of the symbolic name, e.g. *whyte/penthouse*. The symbolic name without the host name is sufficient for agency addressing, if there exists an technique for name resolving comparable to domain name resolving of Internet names. If such a technique is not available, then the agency name must consist of the symbolic name and the host name.

4. <AgencyName> ::= "AgencyName" + "=" + <AddressNameScheme> + "."

5. <AddressNameScheme> ::= ["SymbolicName" | "HostName" | "SymbolicName" + "+" + "HostName"]

For migration it is necessary to address a single agency. Addressing requires more information than naming, because a network connection usually requires the name

of a network protocol and a port number to which communication is directed to. To require a protocol makes of course only sense, if the mobile agent system provides more than a single transmission protocol. If a port number is required too, then the mobile agent system should commit that this port number is never changed and that all agencies use the same port number, as a change forces changes at all agent's source codes (assumed that addresses are hard-coded in the agent's sources and not given by the user). As this usually cannot be guaranteed, for example because another software system already uses this port number, it is wise to allow a dynamic resolution of port numbers according to the required protocol. For example in Tracy there is a *port resolution service* from which information about used port numbers and network protocols can be obtained, so that the user never has to deal with port numbers. Thus, in addition to the source code presented on p. 55, we could also write *go("tracy://tatjana.cs.uni-jena.de").*

In Grasshopper the *region registry* is responsible to resolve port numbers. In a system where such service is not available, protocol and port number is mandatory if the agency offers several transmission protocols in parallel. A mobile agent system might allow more than a single addressing scheme.

6. <AgencyAddress> ::= "AgencyAddress" + "=" + <AddressSchemes> + "."

7. <AddressSchemes> ::= <AddressScheme> + ({";" + <AddressScheme>})

8. <AddressScheme> ::= ("Protocol" + "+") + <AddressNameScheme> + ("+" + "PortNumber")

Creating Agents

When an agent is created, it must be decided where the agent is started first. In the usual case, the agent is started at the current agency, i.e. the one on which the agent's owner placed the creation command. In this case, the current agency becomes the agent's home agency. Sometimes it might be useful to start the agent immediately at a remote agency, e.g. if the current host does not have enough resource to execute a full equipped agency software, but only some kind of loader. To allow this feature, other agencies must support remote starting by offering a communication interface for this.

9. <Creating> ::= "CreateAt" +
"=" + "CurrentAgency" + (";" + "RemoteAgency") + "."

Agent Code

The next important issue that must be decided is the place from which code can be loaded. We call such a place a *code source*. The code source is used to load classes

when the agent is created at its home agency as well as whenever classes are not available at any remote agency and must be loaded dynamically on demand.

The most natural way is to add the directory, where agent class files can be found, to the *CLASSPATH* variable, so that the Java virtual machine can find it. This solution is not comfortable, because it makes it necessary that *CLASSPATH* already contains all directories for all agents that might ever be started at this agency *before* starting the whole agency software. The reason for this derives from a limitation of the Java virtual machine, which does not allow modifications of *CLASSPATH* during runtime. The consequence is that the whole agency must be shut down in order to make changes of the *CLASSPATH* variable visible. More flexible is to allow agent class files to be stored anywhere in the file system and to give their location as parameter during agent creation. In both cases it is not easy for remote agencies to load classes, because direct access to the file system of a remote agency is usually prohibited, as long as class loading is not part of the migration protocol. The most flexible way is to store class files somewhere in the file system, where they are reachable using standard network transmission protocols, e.g. HTTP or FTP.

Next, the file format for the agent's code must be chosen. The easiest way is of course to store each class file separately as this is the output of the Java compiler. Java also allows to bundle many class file into a single Java archive (JAR) file, which might be compressed and digitally signed.

10. <Code> ::= "Code" + "=" + <CodeSources> + "."

11. <CodeSources> ::= <CodeSource> + ({ ";" + <CodeSource>})

12. <CodeSource> ::= ["ClassPath" | "FileSystem" | "HTTP" | "FTP" | "MigrationProtcol"] + "+" + ["Class" | "Jar"]

Agent Data

Next, it must be decided, what types of data the mobile agent can access and use. Each type of data has its own behavior during agent migration. We distinguish four types of data that are useful for mobile agents:

Proxy non-mobile, remote access possible. Data items of this type only exist at the agent's home agency and are not movable to other agencies. However, mobile agents can access these data items transparently from any other remote agency and any modifications are transmitted to the agent's home agency. We name this data type *proxy* as on each agency there exists a proxy object[1] that is part of the serialized agent and that is responsible to transparently forward modifications to the home agency. Access to files or the graphical user interface might benefit from using this type of data.

[1] Compare the proxy design pattern described in Gamma et al. [1994].

Static non-mobile, remote access not possible. Data items of this type only exist at a single agency and cannot be taken along during migration as they are physically bound to it. This data type is common for files or the graphical user interface, whenever they are not of type proxy. In Java, these data items can be marked as **transient**, so that they are not part of the serialized agent.

Moving mobile, source removed. This data type is used for all local or member variables of the agent that are not shared with other agents or the agency. Data items of this type are part of the serialized agent. After migration, data items of this type do not exist at the sender agency any more.

Copying mobile, source not removed. This data type is used for all variables for which the agent has a reference and that are shared with other agents or the agency itself. A copy of the data item is part of the serialized agent, so that at the remote agency the agent has still access to it but modifications are not visible at the original data item at the last agency.

13. <Data> ::= "DataTypes" + "=" + <DataTypes> + "."

14. <DataTypes> ::= <DataType> + ({ ";" + <DataType>})

15. <DataType> ::= ["Proxy" | "Static" | "Moving" | "Copying"]

Migration

We subdivide design issues concerning migration into the following items:

16. <Migration> ::= <Initiator> + <Mobility> + <DestinationAddress> + <Effect> + <Error>

The first design issue that must be discussed is the initiator of an agent migration. Migration can be initiated by the agent itself, or by some other instance, e.g. another agent or the agency. Usually, it contradicts the autonomy of software agents that an external instance can decide to migrate an agent. Nevertheless, in some situations it does make sense to allow this, e.g. for load-balancing or in the case of severe errors, which makes it necessary to shut down a complete agency. In this case the agent should be forced to migrate to another system to prevent severe damage for the agent. The other scenario, where migration is initiated from outside the agent is when an agent is forced to migrate back to its home agency. If migration is initiated from outside the agent, is might be useful to allow the agent to vote against the migration.

17. <Initiator> ::= "MigrationInitiator" + "=" + "Agent" + <OtherInitiator> + "."

18. <OtherInitiator> ::= (";" + "OtherAgent" + ("withVeto")) + (";" + "Agency" + ("withVeto")) + (";" + "Owner" + ("withVeto"))

The next issue is the type of mobility. Existing mobile agent systems can be distinguished by the type of mobility they offer to the programmer and, actually, this is the most discussed design issue concerning agent migration in the literature. Each type of mobility can be characterized by the interpretation of the term *state*. The type of mobility can be *weak* or *strong*, and in both cases further issues must be decided.

19. <Mobility> ::= "Mobility" + "=" + ["Weak" + "." + <Weak> | "Strong" + "." + <Strong>]

The weakest form of mobility only transmits the instance variables (object state) and the code of the mobile agent to the destination platform. The mobile agent is initialized and started by invoking a designated method. This kind of mobility is used for example in Aglets, Grasshopper, Mole [Straßer et al., 1997], and Discovery [Lazar and Sidhu, 1998]. We call this type of mobility *weak mobility with fixed method method invocation*.

In a stronger form of mobility the mobile agent system allows the programmer to define the name of a starting method within the *go*-command. On the destination site the agent is initialized and started by invoking the given method. This kind of mobility is used e.g. in Voyager [ObjectSpace, 1997]. The drawback of these two forms of mobility is that the programmer has additional effort to implement state marshaling and unmarshaling of local variables. Consider the following example:

```
1  public class AMobileAgent extends MobileAgent
2  {
3    private int _copyOfLocal = 0;
4
5    protected void anyMethod()
6    {
7      int local = 10;
8
9      // some code
10
11     // before we can migrate, we have to save variable local
12     _copyOfLocal = local;
13
14     go( ''tracy://tatjana.cs.uni-jena.de'', ''runAtRemote'' );
15   }
16 }
```

This example shows how to save the value of local variables in object variables, so that it is part of the serialized agent. Method *runAtRemote* can use variable *_copyOfLocal* again.

In both mobility levels, the migration command can only be the last instruction within a method, as changing the platform induces invoking a new method. The difference between these two types of mobility is not really evident, as it is very easy to map the latter type of mobility to weak mobility, for example in the Java programming language. We show an example of an agent that simulates the latter form of mobility, although the mobile agent system only provides weak mobility.

```
1  public abstract class GoWithMethodName extends MobileAgent
2  {
3    private String nextMethod = null;
4
5    protected void go( URL destination, String methodName ) throws Exception
6    {
7      nextMethod = methodName;
8      go( destination );
9    }
10
11   public void run()
12   {
13     Method method = this.getClass().getMethod( nextMethod, new Class[] {} );
14     method.invoke( this, new Object[] {} );
15   }
16 }
```

The *go*-command stores the name of the method to invoke at the remote agency in a variable that is part of the serialized agent. At the remote agency, the designated method *run* is started as usual for weak mobility. This method uses the Java reflection mechanism to call the given method itself. We call this type of mobility *weak mobility with arbitrary method invocation.*

20. <Weak> ::= "WeakMobility" + "=" ["FixedMethod" | "ArbitraryMethod"] + "+" + ["Command" | "Ticket"] + "."

When only a weak form of mobility is offered, the command to initiate the migration can be a specific command or predefined method of the agent, or a *ticket*. The first way is to use a migration command, e.g. *go* or *move*, where parameters define, to which agency the agent should migrate to. The other way is to store necessary information about the destination is a data structure that is called a ticket. When agent execution terminates, the agency reads the ticket and migrates the agent. Using a migration command has the consequence that agent execution is terminated

at exactly the point where the migration command occurs. All statements following the migration command are never executed, except an migration error occurs, which we will discuss later. Using a ticket does not give direct control of migration to the programmer. A ticket can be redefined several times which makes it not obvious to the programmer what the agent really will do when execution terminates.

Mobile agent systems that offer the highest level of agent mobility can not only marshal all instance variables, but also all local variables of the current method, together with the program counter and the call stack of the Java virtual machine. On the destination platform the agent is initialized and started at the first instruction after the *go*-command. We call this type of mobility *strong*. First mobile agent systems, like Telescript [White, 1996] or AgentTCL [Gray, 1996] offered this kind of mobility, because it is the most natural one for the programmer. It is comparatively easy to add all features to support strong mobility to a mobile agent system, if full access to the underlying programming language, the compiler, and the runtime-system is available. A new command `go` can be supplemented that initiates the complete marshaling process, or open access to call stack and program counter can be conceded to the programmer of the mobile agent system.

To implement strong mobility in mobile agent systems written in the Java programming language, the source code of the Java virtual machine (JVM) must be modified, or the agent's source code has to be transformed to simulate this. Modifying the JVM is difficult to carry out, although it is said to be done in Ara [Peine and Stolpmann, 1997] and Sumatra [Acharya et al., 1997], compare also the Merpati project at University of Zurich, Swiss [Suri et al., 2000]. A modification of the JVM has also to be considered strategically imprudent. A customer can only use the resulting mobile agent system, if he uses the modified JVM, not to mention problems of licensing the JVM source code.

The other way to offer strong mobility to the programmer is agent source code transformation. Fünfrocken [1999] transforms the agent's source code by a pre-processor that inserts code to save and recover the execution state. Another comparable attempt is made by Sekiguchi et al. [1999]. The drawback of both methods is a longer source code and a not neglectable performance decrease. Other techniques to achieve strong mobility were developed by Illmann et al. [2001], Bettini and Nicola [2001], and Wang et al. [2001].

Strong mobility, in the way we described it above, does only mean that the agent can interrupt itself to start the migration. The reverse case, that the agency can interrupt an agent, e.g. to perform load-balancing or to start an emergency migration to a neighbor-platform because of a system failure, is not possible with any of these migration concepts. As it is at the time of writing impossible to achieve this *transparent* type of mobility, we omit to add a design alternative for this case. Compare Walsh et al. [2000] who describe how to achieve this type of mobility in principle in Java.

21. <Strong> ::= "StrongMobility" + "=" + ["SourceCodeTransformation" | "JVM-Modification"] + "."

Whether weak or strong mobility should be implemented in mobile agent systems, was a major issue controversy discussed in the literature in the last years. Baumann [1995] states that strong mobility is in most case useless, because "... a migration step is a major break in the life of an agent". Usually, a mobile agent works in phases, where each phase is completely executed at a single agency. It is at the transition between phases, where a migration takes places and not within a single phase. Cabri et al. [2000] argue along the same line while stressing that weak mobility leads to a "clean programming style, ..., resulting in a more clear and understandable program". Advocates of strong mobility argue with the more natural programming style and with advantages in agent engineering that are possible with this type of mobility, see for example Belle and D'Hondt [2000]. Walsh et al. [2000] argue that the advantage of strong mobility is "that long-running or long-lived threads can suddenly move or be moved from one host to another", which immediately leads to the question of agent autonomy.

Next, the target of a migration must be discussed. Each migration is directed to some target, whose address must be defined using a migration command or a ticket. Usually, migration is directed to a remote agency whose name is known. Migration can also be directed to another agent or a resource that the agent wants to use. Then, it must be discussed, whether only the next destination is specified or a complete itinerary or route can be defined. In some application scenarios it might be very useful to have a mechanism to define a route, because the agent has to repeat a single task on several agencies, e.g. to collect data. The route can be defined by the programmer of the agent and can be fixed, i.e. not modifiable by the agent during runtime, or fully flexible, so that the agent can define the route by itself.

22. <DestinationAddress> ::= "DestinationAddress" + "=" + <DestinationType> + ({ ";" + <DestinationType>}) + "."

23. <DestinationType> ::= <Resource> + "+" + <Cardinality>

24. <Resource> ::= ["Agency" | "Agent"]

25. <Cardinality> ::= ["Single" | "Fixed Route" | "Definable Route"]

The next issue is the effect of an agent migration. Usually, a migration has the effect that the agent is moved completely to the remote agency and there still exists only a single instance of this agent. An alternative is to make a fresh copy of the agent, which is sent to another agency and started as if it has not existed before. The other case is to clone the agent. In this case a copy is sent to a remote agency,

but this copy already has the same data as the original agent. The latter technique is for example used in AgentTCL and is called *forking* there.

26. <MigrationEffect> ::= "MigrationEffect" + "=" + <Effects> + "."

27. <Effects> ::= <Effect> + ({ ";" + <Effect>})

28. <Effect> ::= ["Move" | "Copy" | "Cloning"]

The final design issue is related to an agency's behavior in case of a migration error. The technique to use here depends on the type of mobility chosen. The first technique is to restart the agent and a local variable indicates that an error has occurred. If a weak form of mobility was chosen, then this kind of error notification is used. If the system allows to invoke an arbitrary method after successful migration, then invoking an error method is a good alternative. The reason of the migration failure can be given as parameter for example. If a system provides strong mobility, then throwing an exception is the best technique. Grasshopper offers this technique too, although it only offers a weak form of mobility.

29. <Error> ::= "MigrationError" + "=" + ["Restart" | "ErrorMethod" | "Exception"] + "."

5.1.2. Agent's View

In this section we will have a look at the way code can be relocated within the network. Data transmission is not an issue here, because the types of data supported by the mobile agent system were already defined in the last section and all systems solely offer a technique were the serialized agent is sent as a single unit from the current agency to the next remote agency.

Migration Strategies

The type of code relocation that is used for agent migration is named *migration strategy*. In the following, we will describe four common migration strategies with regard to transmission type, site location and code granularity, compare Figure 5.2.

Some systems offer a migration strategy that we call *push-all-to-next strategy*. The code of the agent (together with the code of all referenced objects) and the serialized agent, are transmitted at once. Some systems do not transmit the whole code but filter out those pieces of code available on each platform already, e.g. ubiquitous classes, like the standard classes of Java and code of the mobile agent system. This migration strategy is used for example in Voyager 2.0 [ObjectSpace, 1998], Ara [Peine, 1997], and Extended Facile [Knabe, 1997a]. It corresponds to one of the main characteristics of mobile agents, that is autonomy. The agent needs no connection to the home

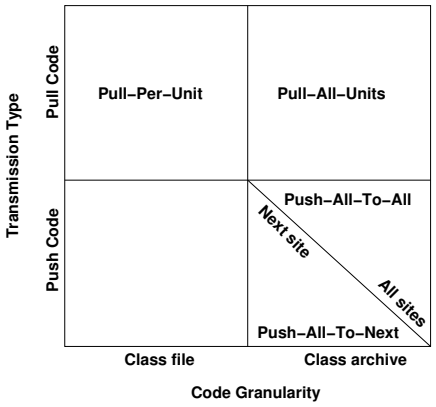


Figure 5.2.: Overview of migration strategies.

agency, from which it was started. At a first look we could consider this strategy to be fast, because only one transmission is necessary for the complete agent. However, a major drawback is that code is transmitted to the destination site that is probably never used.

The second approach does not transmit any code along with the data transmission. We call this the *pull strategy.*After receiving and unmarshaling the agent's data, the mobile agent server on the destination site tries to invoke the given method and then starts loading the corresponding class files dynamically. The pull strategy can be further divided in *pull-per-unit* and *pull-all-units*. The first strategy dynamically loads code on a per-class policy, whereas the second strategy loads all class files as one package immediately if one class file must be loaded. The pull strategy is used for example in Mole [Straßer et al., 1997] and in Grasshopper. This strategy can be slower than the push-all-to-next one, because several network connections may be necessary to load all required class files. When delegating this task to the Java virtual machine, one network connection per class is needed (pull-per-unit), unless several classes have been combined into one Java archive (pull-all-units). The major drawback of both pull-oriented migration strategies is that there must be an open network connection, or at least a fast way to reconnect, either to the home platform or to the last platform the agent came from. If it is impossible to connect to any of these platforms, the agent cannot be executed.

The fourth migration strategy is *push-all-to-all* strategy. As in the push-all-to-next strategy the complete code of an agent is transmitted, but not only to the next destination, but to all destination platforms the agent is going to visit. Of course, this requires that the agent knows all its destinations in advance, e.g. by a given itinerary. When an agent arrives on a destination platform, the execution can start

immediately without any further code downloading.

Besides these four different strategies, combinations of push and pull techniques are used in some systems. As we will describe in detail later, for example Aglets offers a technique where classes with a high probability to be used at the next agency are pushed and missing classes are pulled from a code server on demand later. The MASIF standardization approach suggests to push only the agent's main class and let all other classes be loaded on demand. However, the decision which classes are pushed and which are pulled is done by the system. In contrast, for example in Sumatra [Acharya et al., 1997], the programmer can combine push and pull strategies manually.

Almost all mobile agent systems offer only one of these strategies and an interesting question is, whether it makes sense to adapt the migration strategy according to an application scenario. We already mentioned some qualitative arguments in favor of each migration strategy, but we did not provide exact quantitative arguments. For the moment, we will only mention that our own research [Braun et al., 2000b, 2001b] has come to the result that there are non-neglectable performance differences between all strategies – we come back to this question in the next chapter of this thesis.

Code Transfer

Concerning code relocation we will now discuss the following issues:

30. <Agent> ::= <CodeTransfer> + (<CodeCache>) + (<UbiquitousClasses>)

As described in the last section, code relocation strategies can be distinguished in push and pull strategies. Some systems offer both approaches, which has the consequence that some (not all) classes are pushed to the next destination and other classes are pulled on demand.

31. <CodeTransfer> ::= "CodeRelocation" + "=" + <CodeStrategies> + "."

32. <CodeStrategies> ::= <CodeStrategy> + ({ ";" + <CodeStrategy>})

33. <CodeStrategy> ::= [<Push> | <Pull>]

When the system offers a push strategy it must be decided, which classes must be sent to the agency. We call this issue <ClassClosure>. The first technique is to determine all classes the agent might ever use during its life-time from the agent's main class file. How to determine an *agent class closure* in Java was already described in the last chapter. The next technique only collects those classes for which an object exists in the serialized agent ("SerializedAgentClosure"). As these classes are necessary to deserialize the agent successfully at the remote agency, it is a good compromise between sending all classes at once and no classes at all. However, the

fact that some classes are *in use* does not say anything about the probability for other classes to become in use just at the next agency. Some systems allow the user to bundle classes in a Java archive (JAR) file and whenever a class from a JAR file must be transmitted, the whole JAR file will be transmitted ("JarClosure"). The last technique allows the user to make the decision which classes to transmit dynamically during runtime.

The next design issue that must be decided, is whether code can be pushed only to the next agency or to all agencies that were defined in a route or that are available within the logical network.

34. <Push> ::= "Push" + <ClassClosure> + "To" + <PushTarget>

35. <ClassClosure> ::= ["AgentClassClosure" | "SerializedAgentClosure" | "JarClosure" | "UserDefinedClosure"]

36. <PushTarget> ::= ["NextServer" | "ManyServers"]

When a system offers a pull technique it first must be decided what transmission unit is used. Usually, only single class files will be transmitted. Although not implemented in any mobile agent system, also downloading of a complete JAR file is possible. The next issue is the place where the remote agency looks for the class to be loaded. Here, a mobile agent system must define a strategy, which instances are to be asked for the code and in which order. Possible components are for example the class loader which is responsible to load all agent's classes. A class loader should have a local cache of all classes already loaded. Next, the local *CLASSPATH* variable could be checked for a class with the given name. Then, an agency wide class cache, which is shared by all class loaders can be asked, and at last, several other agencies, for example the last agency visited or the agent's home agency can be asked.

37. <Pull> ::= "Pull" + ["Jar" | "Class"] + "From" + <PullTargets>

38. <PullTargets> ::= <PullTarget> + ({ "+" + <PullTarget>})

39. <PullTarget> ::= [<AgencyType> | "ClassLoader" | "ClassPath" | "Cache" | "CodeSource"]

40. <AgencyType> ::= ["Home" | "Remote" | "LastServer"]

An important design issue is to decide which classes are never transmitted, even if they are member of a class closure, because they are assumed to exist at every agency already. Those classes were named *ubiquitous* in the last chapter. A mobile agent system might define that this is defined by the system, e.g. all class from Java packages (which have specific prefixes) and all classes of the mobile agent system are

never transmitted. Sometimes it might be useful to allow the user to add package prefixes or class names to this list ("UserDefined"), because mobile agents are part of some application which can also be assumed to exist on every remote agency. The most flexible way is to allow the agent to define its own filter list.

41. <UbiquitousClasses> ::= "UbiquitousClasses" + "=" + ["SystemDefined" | "UserDefined" | "AgentDefined"] + "."

Last, it is important to decide, whether class code is cached by the agency after it was loaded for the first time. Although caching is a good technique to save time when loading the same classes very often, it has the negative effect that class code changes might not become visible to the agency as the cache is not informed about changes. Caching can be found most often when an agent's code is reachable using the *CLASSPATH* variable, because all classes of the *CLASSPATH* are loaded by the system class loader of the Java virtual machine. For other classes it must be decided whether the agent class loader or a component of the agency manages the class cache. In the first case, agents of the same type are not able to share code, which has the consequence that the same class might be loaded more than once. In the latter case, agents might share classes which might lead to problems with different class versions. In this case it must also decided whether any kind of version management is implemented. Another design issue concerning code caches is whether the class is only asked before loading classes or also before class transmission. The first approach can only prevent class downloading, whereas the second approach could also prevent classes to be pushed to the agency. This technique must be supported by the migration protocol, because class names must be sent before pushing any agent code. The remote agency could check for which classes the code is already available.

42. <CodeCache> ::= "CodeCache" + "=" <Instance> + "+" + ["BeforeTransfer" | "BeforeLoad"] + "."

43. <Instance> ::= ["ClassLoader" | "Agency"] + "+" + ("VersionManagement")

5.1.3. Network's View

The last view considers all aspects that are related to data transmission. The *transmission strategy* defines the way an agent is actually transmitted to the destination platform. Some Java based mobile agent systems, e.g. Mole, use a proprietary and very simple migration protocol that is based on *remote method invocation* (RMI) [Sun, 2002] for this task. The destination agent server a RMI server that provides a method to accept a mobile agent. The penalty of using RMI is its poor performance. Most systems use migration protocols on top of TCP/IP or HTTP.

Concerning the migration protocol, it must be decided whether an asynchronous or synchronous protocol is used. The advantage of the first one is performance, whereas

it can also be more unreliable, as the remote agency does not acknowledge reception. The second issue to decide is whether migration is failure atomic, i.e., whether the migration protocol guarantees that the mobile agent is transmitted completely or not at all. The next issue is the network protocol. Here, several protocols are listed, the most common are perhaps TCP/IP and RMI. Grasshopper defines some kind of meta-protocol, which can be used to determine which protocols the receiver agency supports before sending the agent.

44. <Network> ::= <MigrationProtocol> + <TransmissionProtocols>

45. <MigrationProtocol> ::= "MigrationProtocol" + "=" + ["Synchronous" | "Asynchronous"] + ("FailureAtomic") + "."

46. <TransmissionProtocols> ::= "TransmissionProtocol" + "=" <NetworkProtocols> + "."

47. <NetworkProtocols> ::= <NetworkProtocol> + ({ ";" + <NetworkProtocol>})

48. <NetworkProtocol> ::= ["TCP/IP" | "CORBA" | "SSL" | "RMI" | "RMISSL" | "SOAP" | "META" | "HTTP" | "Other"]

5.2. Examples for Mobility Models

In the following section we will describe the mobility model of the mobile agent systems Aglets and Grasshopper. We chose these two systems, because they are considered as the most famous system and used for real-world application development. We will not describe the Tracy mobility model here, as a detailed introduction into the migration technique of Tracy will be part of the next chapter.

5.2.1. Aglets

Aglets is a mobile agent system that was developed by IBM since 1995. The first announcement was given at the JavaOne conference in 1996 and the first version of Aglets was published in 1998. Since the year 2000, Aglets is an open source project at Sourceforge and no longer supported by IBM. The Aglets system supports the MASIF standard.

Aglets provides a weak form of mobility, where a method named *run* is called whenever an agent is started or restarted at an agency. The migration initiation command is named *dispatch* and gets as single parameter the URL of the destination agency. To address agencies, a protocol and a host name are mandatory, a symbolic agency name can be added, if more than a single agency exists at the destination. The port number is predefined and cannot be changed. Route management is not

supported by the *dispatch* method but can be implemented by the user with a specific design pattern.

When an agent is created only a single code source can be defined, but the type of code source is very flexible. It usually contains a URL to a resource that can be accessed using the HTTP protocol or the Aglets migration protocol ATP. Classes must be stored in a directory that is listed in the *AGLET_EXPORT_PATH* environment variable to be fetched from any agency using the ATP protocol. If the code source contains a file system path, classes cannot be downloaded from remote agencies. If no code source was defined at all, then the agent's code is loaded using the environment variable *AGLET_PATH*, which contains a list of directories of the local file system. These classes cannot be transferred to other agencies.

The migration strategy of Aglets is not just a simple push or pull strategy but a sophisticated combination of both. When an agent should migrate all classes for which an object exists in the serialized agent (Aglets calls these *classes in use*) are pushed to the next agency. If the code source is a JAR file, then the whole JAR file is pushed to the next agency, without regard whether a class is in use or not. At the destination agency, missing classes are loaded on demand (pull). To load a class, the following strategy is used. First it is checked whether the current class loader has already loaded this class before, next the class is searched in the local *CLASSPATH*. This has the effect that a local class might be loaded although the agent has pushed its own class file. Third, the local cache manager is checked, which is in the current version of Aglets only able to cache JAR files correctly. At last, the agent's code source is checked for the code. Code downloading always happens on the base of single class files and never on the base of complete JAR files. Classes that are not supposed to migrate must be defined using the two environment variables *CLASSPATH* or *AGLETS_PATH*.

The user can influence the migration strategy used for the next migration only by modifying the object state of the agent. To prevent a class to be transferred to the next agency, no object of this class must exist in the serialized agent. The most flexible way to achieve this is to set all variables of this class to **null**. Then, the Java serialization process will not consider this class. The other way is to define a variable as **transient**, which has the same effect. The reverse case, to transmit a class although no object of this type currently exists, can be achieved as easy. It is only necessary to add a variable of type *Class* which must be initialized with the name of the class to transmit, for example *Class forceTransmission = MyClass*.**class**.

Other important issues of the Aglets mobility model are for example, that an agent's owner is able to retract an agent from any agency, if he knows the current location of it. Aglets uses the standard Java serialization technique, so that data types static, copy, and move are supported. The class cache is only asked before class downloading and the migration protocol is defined on top of TCP/IP and can be also be tunneled within a HTTP protocol to get through firewalls.

Here is the complete description of the Aglets's mobility model:

```
1  // Programmer's view
2  //
3  AgencyName = SymbolicName + HostName .
4  AgencyAddress = Protocol + HostName ; Protocol + HostName + SymbolicName .
5  CreateAt = CurrentAgency .
6  Code = ClassPath + Class ; ClassPath + Jar ; FileSystem + Class ;
7         FileSystem + Jar ; HTTP + Class ; HTTP + Jar ;
8         MigrationProtocol + Class; MigrationProtocol + Jar .
9  DataTypes = Static ; Copy ; Move .
10 MigrationInitiator = Agent ; OtherAgent ; Owner .
11 Mobility = Weak .
12 WeakMobility = FixedMethod + Command .
13 // Agent's view
14 //
15 DestinationAddress = Agency + Single .
16 MigrationEffect = Move .
17 MigrationError = Restart .
18 CodeRelocation = Push SerializedAgentClosure To NextServer ;
19                  Push JarClosure To NextServer with base Jar ;
20                  Pull Class From ClassLoader + ClassPath + Cache + CodeSource .
21 UbiquitousClasses = SystemDefined .
22 CodeCache = ClassLoader + BeforeLoad .
23 // Network's view
24 //
25 MigrationProtocol = Synchronous .
26 TransmissionProtocol = TCP/IP ; HTTP .
```

5.2.2. Grasshopper

The Grasshopper mobile agent system is developed by IKV++, Berlin, Germany. The first version was developed at GMD FOKUS in 1995. Since 1998 the product is maintained by IKV, which is a GMD spin-off company. Currently, Grasshopper is passed over to a be part of a new product called enago. Grasshopper supports both MASIF and FIPA standards.

Grasshopper provides a weak form of mobility, where a method named *live* is invoked to start the agent. The migration initiation command is *move* which gets as parameter the URL of the next destination. Interesting is Grasshopper's technique for catching migration errors as in this case an exception is thrown. Grasshopper does not provide techniques for route management.

When an agent is created, several code sources can be defined from which code can be loaded on demand. A user definable code source can be either on the local

file system, or can be accessed via the HTTP protocol. In both cases it seems not to be possible to define a JAR file as code source. When an agent migrates, *no* classes are transmitted along object state migration, so that Grasshopper does not support any kind of push strategy. When the agent is deserialized, classes must be pulled according to the following strategy. First, the agency's *CLASSPATH* is checked by the system class loader. If code is reachable using this class loader, no two different classes with the same name can exist. Next, the last agency from which the agent has come from is asked for the class. If it is still not found, all agent's code sources are checked sequentially, and at last the agent's home agency is asked for the code. In all these cases, class files are only cached by the agent's class loader so that transmission of the same class for different agents is not avoided. As Grasshopper only provides a single pull migration strategy, there is no chance for the user to adapt migration behavior of its agent. It is even impossible to modify the class downloading strategy, for example to bypass the last agency to be asked for the code.

Other important issues of the Grasshopper mobility model are that an agent can be forced to migrate by other system components, but the agent is able to vote against a migration. Very interesting are Grasshopper's transmission strategies. Grasshopper not only supports several network protocols, but also has a meta protocol by which two agencies can communicate which network protocols are supported by both systems. A service called *region registry* is responsible to maintain a directory of all agencies active within the local subnetwork. Using this region registry makes it for example possible to omit port numbers in agency address.

```
1  // Programmer's view
2  //
3  AgencyName = HostName + SymbolicName .
4  AgencyAddress = HostName + SymbolicName ; Protocol + HostName +
       SymbolicName ;
5                  Protocol + HostName + SymbolicName + PortNumber .
6  CreateAt = CurrentAgency .
7  Code = ClassPath + Class ; HTTP + Class ; .
8  DataTypes = Static ; Copy ; Move .
9  MigrationInitiator = Agent ; System WithVeto .
10 Mobility = Weak .
11 WeakMobility = FixedMethod + Command .
12 // Agent's view
13 //
14 DestinationAddress = Agency + Single .
15 MigrationEffect = Move .
16 MigrationError = Exception .
17 CodeRelocation = Pull Class From ClassLoader + ClassPath + LastServer +
18                  CodeSource + Home.
19 UbiquitousClasses = SystemDefined .
```

```
20 CodeCache = ClassLoader + BeforeLoad .
21 // Network's view
22 //
23 MigrationProtocol = Synchronous .
24 TransmissionProtocol = TCP/IP ; CORBA ; SSL ; RMISSL ; META .
```

5.3. Related Work – Other Classification Approaches

In this chapter we proposed a classification scheme for the migration issues of mobile agents. Although almost each mobile agent system has implemented a different migration technique, as far as we know, no sophisticated approach was developed to classify these different techniques so far. Some authors describe design issues of mobile agent systems in general, e.g. related to agent communication, agent naming, security, etc. Concerning agent mobility these approaches keep superficial and in most case do only mention the difference between weak and strong mobility.

For example, Hammer and Aerts [1998] collect several design issues concerning mobile agent systems, but only three issues concerning migration were detected. First, according to Hammer and Aerts, it must be decided, whether migration is *state preserving* or not. The next issue is whether migration is *failure atomic* or not, and the last issue is whether the agent can define an itinerary or not. Karnik and Tripathi [1998] only distinguish between strong and weak mobility, whether the agent is moved, cloned, or forked, and whether code is pushed to the next agency or pulled from the home agency. Cabri et al. [1998] only decide between strong and weak mobility and whether migration is initiated explicitly by the agent itself or implicitly by the underlying agency software.

Fuggetta et al. [1998] have proposed the so far best approach to classify different techniques for mobile code, unfortunately their approach is not suited to classify mobile agent migration techniques as it mainly tries to cover all types of mobile code approaches, as code-on-demand, remote-evaluation, and mobile agents. The authors distinguish strong and weak mobility, where they use the term weak mobility in case of remote-evaluation, where except of some initialization data no *state* information is shipped to the remote server. In our definition, weak mobility contains the object state of the agent. The authors define that strong mobility is supported in two forms, migration and remote cloning. The first form is comparable to our migration effect of *moving* an agent, whereas the latter one is comparable to our migration effect of *cloning* an agent. Weak mobility is divided into code shipping (remote-evaluation) and code fetching (code-on-demand) – do not confound this with our distinction between push and pull migration strategies. Finally, the authors distinguish between asynchronous and synchronous techniques, where the sending execution unit is either

non-blocking or blocking and waiting for the result. The authors also classify data space management, which results in a comparable enumeration of alternatives as in our approach for the *data* design issue.

6. Reasoning about Improved Mobility Models – The Kalong Model

In the previous two chapters we have collected information about current implementation techniques for mobile agent migration. We can now start reasoning about new mobility models resp. migration techniques that surmount the drawbacks of simple mobile agents, as discussed in Section 3.3.

In Section 6.1 we will discuss these drawbacks against the background of current implementations and evaluate whether today's mobility models are able to solve any of these drawbacks. Then, in Section 6.2, we will collect factors influencing mobile agents' performance and discuss, how this performance can be improved. One important aspect of Section 6.2 is to investigate whether the migration strategy has an effect on the overall performance of mobile agents. At last, in Section 6.3, we will describe our idea of a new mobility model, named Kalong. Using the Kalong mobility model gives the programmer of mobile agents resp. the mobile agent itself the possibility to influence the migration strategy in a very fine-grained way and offers other very important new features to increase migration performance.

6.1. Drawbacks of Simple Migration Techniques and Current Implementations

In Section 3.3 we discussed the results of our mathematical model for network load and transmission time for mobile agents using a single, very simple migration technique (push-all-to-next). We detected several inherent drawbacks of mobile agents that are responsible for a higher network load and higher processing time as compared to client-server approaches.

The inherent drawbacks of mobile agents are in summary:

- An agent's code is typically larger than a simple client-server request and causes a fixed overhead for each migration. It is only meaningful to send large code to remote agencies, if server results can be decreased decisively by data filtering or compression.
- An agent's code is transmitted to a remote agency, even if it is never used over there. In the case of the push-all-to-next migration strategy, transmission

of never used classes cannot be avoided. However, even in the case of a pull migration strategy, code might be loaded superfluously because of the object state serializing technique used by all Java based mobile agent systems. We will present an example for this later.

- An agent's data is transmitted as a single unit, which has the effect that a mobile agent carries data items to all servers of the given itinerary, even if they are never used before reaching the home agency again. In the other case, the agent carries data items to several agencies, although they are never used at the first agencies. This was the reason for poor performance in case of a high number of agencies.

However, we have also already detected a major advantage of mobile agents. In case of small-bandwidth network connections a mobile agent must use this bottleneck only rarely for migration, whereas client-server approaches have to use them several times.

It is fair to assume that using other migration strategies than the simple push-all-to-next strategy could be a solution for the problem of superfluously transmitted class code. For example, in the pull-per-unit strategy, code is never transmitted along an agent's state transmission and always loaded dynamically on demand. This has the consequence that classes are only transmitted if they are imperative at the remote agency. It is important to understand what it means for a class to be imperative at a specific agency. Usually, we would expect to load a class, only if the corresponding object is accessed, i.e. used or defined. In Java, at least as long as the Java object serialization technique is used, code must also be downloaded if an object of this type is part of the serialized agent. Look at the following example for a mobile agent:

```
1  import de.unijena.tracy.agent.*;
2
3  public class SomeTracyAgent extends MobileAgent
4  {
5    protected SomeClass first = new SomeClass();
6    transient SomeOtherClass second = new SomeOtherClass();
7
8    public MyFirstTracyAgent()
9    {
10     // do some initialization
11   }
12
13   public void startAgent()
14   {
15     // do something
16     go( "tracy://tatjana.cs.uni-jena.de", "runAtRemote" );
```

```
17    }
18
19    public void runAtRemote()
20    {
21      // do something
22      if( /* ... */ )
23      {
24        SomeOtherClass third = new SomeOtherClass();
25        // do something
26      }
27      go( ``tracy://domino.cs.uni-jena.de'', ``runAtNext'' );
28    }
29  }
```

After initialization, the agent immediately migrates to *tatjana.cs.uni-jena.de* and execution is resumed by invoking method *runAtRemote*. Variable *first* is part of the serialized agent, whereas variable *second* is not because it is marked as **transient**. We assume that the agent migrates using strategy pull-per-unit, so that no classes are sent along the object state. When the agent is deserialized at the remote agency, besides the agent's main class *SomeTracyAgent*, also class *SomeClass* is loaded, because it is needed to reconstruct the agent correctly. Thus, this class *is* loaded, although method *runAtRemote* does not use variable *first* at all. An example, where code downloading really depends on a use, is variable *third*, which is a local one defined within method *runAtRemote*. If we assume that no other object variable uses *SomeOtherClass*, then only if the expression in line 22 evaluates to **true**, code for class *SomeOtherClass* is loaded. What we have learned from this example is that when using the standard Java serialization technique, the agent resp. programmer does not have precise control which classes are downloaded at the destination agency. Even if strategy pull-per-unit is used, classes might be downloaded although they are not really necessary at the remote agency.

We now want to discuss, whether the disadvantages of the two simple migration techniques enlisted above can be resolved by any mobility model presented in the last chapter. The first point – mobile agents' code being larger than a simple client-server request – can of course not be solved by any mobility model automatically. The problem is that an agent's code usually contains other methods that are not needed at the next destination which lengthen the code. An agent's code size would surely benefit from a code split, where each piece of code only contains methods with high coherence. Code transmission would then work on the basis of code pieces. As long as such a *code splitting technique* is not available, the programmer can only attach importance to this problem when designing his agents.[1]

[1]Compare Section 10.3 for more information about a class-splitting technique that was developed

The second point – superfluously transmitted classes – can not be solved neither in the Aglets system, nor in the Grasshopper system. For example, in Grasshopper classes are never pushed to next agencies, but always loaded dynamically on demand. As Grasshopper uses the standard Java serialization technique, some classes might be loaded although code is not needed at the destination. Besides, classes are loaded superfluously in the case when two agents of the same type reside at the same agency. Code is loaded twice in this case, because Grasshopper does not provide code caching on the level of agencies. In Aglets, classes in use are pushed and other classes are loaded on demand. The user is only able to influence which classes are pushed to the next agency, e.g. by storing class code in different directories on the local file system and defining environment variables appropriately. As the Aglets system also bases on the Java serialization technique, classes will be downloaded.

The third point – fine-grained data transmission – cannot be solved by any system, because both use the standard Java serialization technique, where the object closure always contains all data items in use and not marked **transient**. The user has no chance to download data items from the agent's home agency during the agent's tour or to send data items back to the home agency, when it is known that they are not used at the next agencies.

The result of this brief analysis of current mobility models and systems is that none of them is even close to solve the problems we identified to be inherent drawbacks of simple migration techniques. Although both inspected mobile agent systems use other than the simple push-all-to-next migration strategy, they are not able to solve the problem of superfluously transmitted code and they are not able to solve the problem of data transmission. In the next section we will discuss approaches to improve the performance of mobile agents in general.

6.2. Improving the Performance of Mobile Agents

Before we will reason about specific techniques to improve the migration process using new sophisticated mobility models, we will think about performance optimizations for mobile agents in general. In the next section we will concisely discuss several approaches to improve the performance of mobile agents proposed in literature so far. Then, we will discuss the influence of different migration strategies on mobile agents' performance.

6.2.1. Overview of Mobile Agents' Performance Aspects

The performance of mobile agents is influenced by several factors and within the life-cycle of a typical mobile agent, we find several occasions where performance can

as part of the Tracy project.

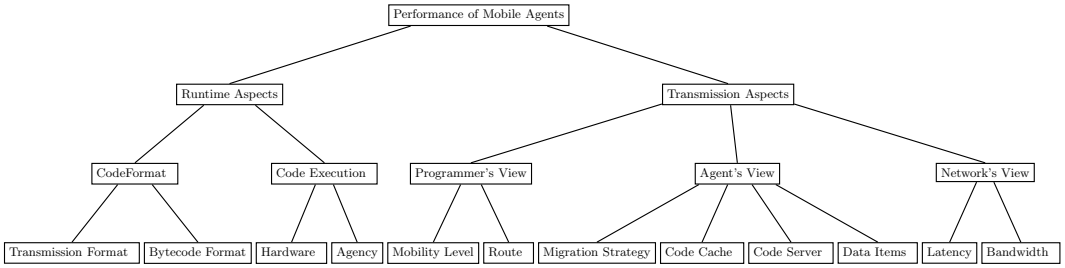


Figure 6.1.: Classification of mobile agents' performance issues.

be improved, as for example: its task given by the user, the route or itinerary, its code size, the size of collected data, network parameters like bandwidth and latency, etc.

To structure our discussion, we developed the following classification schema, compare Figure 6.1. We first divide *runtime aspects* and *transmission aspects*.

Runtime Aspects

In the first class we place techniques by which an agent's execution time can be improved. The first important aspect is the format, used to send an agent's code to destination agencies.

Code Format The code format influences code size and, therefore, transmission time and, with a higher extent, also code execution time. We can distinguish here between source code, intermediate byte code, and machine code. *Machine code* is specific to a processor architecture family and cannot be executed on processors that do not belong to this family. In heterogeneous systems like mobile agent systems, it is very important to have code in a format that can be understood and executed at all or almost all nodes. Therefore, machine code is in general not a good format to transmit mobile agents' code, unless some kind of simulator or translator is used that is able to translate code between different architectures. However, as a huge part of computer systems base on either Intel Pentium, Compaq Alpha, or Sun SPARC processors, techniques could be used which send an agent's code in multiple representations to destination agencies. In the distributed operating system community this was for example implemented in systems described by Dubach et al. [1989] and Shub [1990]. In the mobile object system Emerald [Steensgaard and Jul, 1995], not all code representations are sent but only one is selected that will be used at the next destination. This technique is not suitable for mobile agents, as a later migration would not be possible in this case. Knabe [1997a] describes several drawbacks of

these approaches, as for example that addition of a new processor architecture type requires a new compilation of all agents. Additionally, he points out that the number of different processor architectures is too high to allow code transmission for *all* processor types, which is undoubtedly true.

A better solution is to transmit an agent's code in a format that is independent of the underlying processor architecture. This might be *source code* or some *intermediate code* representation, if such is available for the used language. In first mobile agent systems, as for example Telescript and AgentTCL, which used script languages for programming mobile agents, code was transmitted in source code format. Source code must be compiled at each agency again to an executable format, which increases execution time and becomes very uneconomical if a huge amount of source code must be compiled only to execute a few lines of code at this agency. An advantage of this code format is high execution performance due to the imperative compilation process.

Intermediate code is the result of a compilation process that is performed at the agent's home agency. Intermediate code is a low-level representation, which consists of commands for a virtual machine. Java byte code is the most popular example for a intermediated code format, especially for mobile agent systems. Intermediate code is usually more compact than source code and could already contain several architecture independent code optimizations, e.g. dead code elimination or loop unrolling [Aho et al., 2000], which will increase execution performance. Intermediate code is either interpreted by some kind of virtual machine or immediately translated into machine code at the destination agency. Java uses a mixture of both techniques, where code is interpreted when executed for the first time and then translated into machine code during runtime (just-in-time compilation). Later, this code is further analyzed to detect performance bottlenecks and the respective code areas are especially optimized (hot-spot-optimization).

An approach that combines transmission of source code, intermediate code, and machine code is for example proposed by Knabe [1997a]. In his mobile agent system, which uses the programing language Extended Facile, the agent programmer can influence in which transmission format the agent should migrate the next time. The decision might be based on the agent's task as can be seen in the following example which we take from the paper cited above: In the case of a so-called batch agent, which has to execute a long-running process, the destination agency will benefit from a highly optimized machine code which can only be produced, if high-level source code is available for compilation. Another example is related to code size vs. network quality. If the network connection has low bandwidth, for example a dial-up connection, transmission time is more important than execution time and, therefore, the smallest code representation should be sent, independent of how good execution time will be.

If we now look at Java as programming language, another issue that influences performance of mobile agents is its byte code format. Although it is very easy to

produce and interpret, Java byte code has some major drawbacks with regard to code optimization and security. For example, the stack based architecture of the Java virtual machine makes it difficult to optimize code for RISC processors and the built-in byte code verifier does only provide some simple code checks whether basic semantics of Java are violated. In the programming language community, alternatives to Java byte code are discussed that allow easy code annotations for code optimization and provide sophisticated security checks. See for example Amme et al. [2001] for a new safe intermediate code format for Java based on static single assignment that can be translated to highly-optimized machine code very fast.

Code Execution Another aspect that influences execution performance is the underlying hardware architecture, i.e. CPU, memory, system load, etc. Hardware parameters are especially important in the case of Java, where the virtual machine itself needs a huge amount of memory. Unfortunately, this aspect cannot be influenced by the agent or agent programmer. A second aspect is the agency software and its optimizations on the level of Java code. Some drawbacks of the Java virtual machine can be resolved by skillful programming within the agency software. Most of these techniques are not restricted to programming of mobile agent systems, but are applicable in all Java programs. We only want to mention here two techniques, i.e. improved object serialization techniques and thread pools. Some authors propose new object serialization techniques for Java, which either speed up the whole serialization process or produce a smaller serialized object. See for example Philippsen and Zenger [1997] for an example for such a technique.

As mobile agent systems are multi-threaded systems, every software agent usually owns at least a single thread of control. To achieve parallel execution of agents, thread scheduling is provided by the Java virtual machine resp. by the underlying operating system. Unfortunately, thread creation is very expensive in Java and should, therefore, be minimized to save execution time. The problem is aggravated by the fact that mobile agents usually have only a very short visiting time at each agency, so that a thread's life-time is usually very short. To solve this problem, it is a well-known technique in Java to use so-called thread pools, which is more or less a data structure that manages a stock of sleeping threads. Whenever a new thread is needed, no new thread is created but an already existing thread from the thread-pool is resumed and associated with the new task. When the task is finished the thread is put back to the pool again. The technique of thread pools is for example described by Soares and Silva [1999] and its influence on the overall execution performance is measured using several experiments.

Transmission Aspects

In the second class, all techniques are summarized that influence network load and transmission time during agent migration. According to the three views of mobility models, we distinguish here three optimization issues.

Programmer's View From the programmer's point of view, the most influential factor is the level of mobility that is supported. Whereas weak mobility is very easy to achieve and works very fast in Java based systems, it is very complicated to achieve strong mobility in Java, as already mentioned in Section 5.1.1 on page 63. Strong mobility not only increases code size and lengthens transmission time, but also increases execution time as can be seen from the experiments made by Fünfrocken [1999] for example. Another aspect of the programmer's point of view is the itinerary an agent has to execute. The optimization goal within this level is to find suitable agencies and services, and to optimize the route to all of these agencies. The order in which agencies are visited is most important of course and there are several trade-offs to be considered. For example is it more useful to go to neighboring agencies first, because they are easy to reach, and even to risk that information is not found there, or go to far away agencies first where the probability to find right information is high, but migration costs are also high and transmission may be untrusted. In an earlier paper we named this level of migration optimization the *macro* level, see Erfurth et al. [2001a,b].

The only work that proposed an optimization on the macro level is by Barbeau [1999]. He uses the term *migration strategy* to describe the way of relocating an agents' state and code within the network. Barbeau uses a more coarse-grained approach, because he still views an agent's code as one transmission unit, whereas we consider code to consist of multiple pieces that can migrate (almost) independently. He compares three strategies: First, an agent visits all nodes sequentially. Second, all nodes are visited sequentially but the agent's state is uploaded to its home server periodically. Third, the agent is sent in parallel to all nodes that it has to visit using a multi-cast protocol. The author evaluates these migration strategies using mathematical models and is able to show that it makes sense to upload the state of an agent to its home agency under certain circumstances from time to time. Further, he can show that the parallel strategy performs better than any sequential strategy.

Agent's View From the agent's point of view, a mobile agents' performance is influenced by code and data relocation techniques, we call this the *micro* level of optimization. The issue of optimization here is the amount of network load that is produced by a single agent migration between two agencies. Migration time may not depend directly on the amount of bytes that are to transmitted, because in networks with varying values for bandwidth it might be even faster to send a larger number

of bytes through a high-bandwidth network connection than sending a small number of bytes through a one with low-bandwidth.

Techniques to optimize network load and transmission time can be easily deduced from the problems of the simple migration technique, summarized at the beginning of the last section. The first problem is related to the actual size of code, which can be reduced by using sophisticated compression techniques that are developed for Java byte code, see for example Bradley et al. [1998] and Pugh [1999]. The second problem is raised by code that is superfluously transmitted to a destination agency, either because it is not needed there or because it does already exist there due to a prior code transmission of the same or another agent of the same type. However, besides the *qualitative* arguments against this technique (see p. 72), we can also mention *quantitative* arguments. An extension of our model of network load and transmission time to consider different migration strategies is not in the scope of this section, so we postpone it to the following one. There we will compare all migration strategies presented in Section 5.1.2 in two application scenarios. The result of this evaluation is that no migration strategy works best in every situation, i.e., no migration strategy is able to solely produce lowest network load resp. transmission time in all application scenarios.

The consequence is that it is not sufficient to make a simple exchange of strategies, but it is necessary to provide a dynamic choice, which migration strategy should be used for the next migration. As we have seen in the last section, current mobility models are not able to provide techniques which allow a dynamic decision between different migration strategies, e.g. in Aglets it is not possible to decide which class should be pushed or pulled and in Grasshopper it is not even possible to push classes at all.

Another technique to avoid class transmission is a class cache. The limitations of current class caches used in Aglets and Grasshopper have already been described. Those class cache techniques are only able to avoid class downloading from the agent's home agency, but code transmission as done in the push-all-to-next strategy. An evaluation of class caches was done for example by Soares and Silva [1999]. In the case of push strategies current class caches are completely useless. A solution would be a class cache technique that already becomes active when an agent migrates. For example, in a first step the names of classes that should be transmitted are sent to the destination agency as part of the migration protocol. The destination agency answers with information about what classes are already available and which not. In practice we would have to deal with the problem of equal class names for different classes and different class versions, of course.

If class downloading cannot be avoided, then it should be at least as fast as possible. Here it is important to load classes from a near-distance server as a lower distance could improve time for downloading. Current mobile agent systems only allow to define a single code server (Aglets), which in most cases equals the agent's home

agency, or to define multiple code servers, to which code must be transmitted using techniques outside the mobile agent system, e.g. a simple FTP file transfer. Hohl et al. [1997] describe techniques to improve the migration performance by using more than a single code server in the Mole system. This code server need not to be located on the same host as the agent server. If an destination agency must download classes during execution of a mobile agent, it first looks for them at neighboring code servers, before loading them from the home agency. Code servers communicate to exchange information about existing classes. The authors assume that classes are registered at a code server by the programmer manually. They do not provide any performance measurements to prove their concept.

The last drawback of simple migration techniques used so far, is data handling. We saw that it costs a huge amount of network load to carry data items to servers, although these data items are never used at this agency. We already have seen that using the standard Java object serialization technique, that is used in all current mobile agent systems, this drawback cannot be avoided. What is necessary is a technique to transmit data items independently of the object state of the agent, so that an agent can dynamically download data items from its home agency in case they are really needed. Later, the agent can upload these data items again to avoid carrying them to next agencies.

A last aspect we would like to mention here, is the way how code is transmitted from the sender agency or a code server to the destination agency. Usually, code transmission is completed before code execution is started. An optimized code transfer starts code execution *before* code transmission is completed, so that both phases are overlapped for some time. This can be easily achieved for example in the case of Java applets, if the JAR file is reordered so that classes that are needed first at the destination are placed at the beginning of the file. Krintz et al. [1999] and Stoops et al. [2002] describe approaches to implement this technique, not for mobile agent systems but only for mobile object systems.

Network's View From the network's point of view, we see two factors that influence mobile agents' performance: network bandwidth and network latency resp. the overall architecture of the network in which the agent has to operate in. As we will see in the next section in detail, the type of network, e.g., whether all network connections have the same quality or not, has a great impact on performance. Of course, the agent or agent programmer has no mean to influence these values but it is very important to be able to react on them.

6.2.2. Performance and Migration Strategies

In this section we will evaluate the relation between performance of mobile agents and migration strategies used. In the last chapter we introduced the two main classes

of migration strategies, i.e. push and pull strategies. We also learned that in most cases they are used stand alone, but in rare cases also in some kind of combination.

An interesting question is now how the migration strategy influences the migration performance of mobile agents. One severe simplification of our mathematical model for network load and transmission time was especially that only push-all-to-next was supported. To evaluate the influence of the migration strategy to the performance of mobile agents, we will now extend our model to allow dynamical class downloading. Prior version of this model were published as Braun et al. [2000b] and Braun et al. [2001b].

We extend our model, where the agent has to visit m servers $\{L_1, \ldots, L_\mathrm{m}\}$ to collect data from each server. In contrast to our first model, we now assume that an agent consists of several class files, which can be dynamically loaded during execution from the agent's home agency. The decision which class files must be loaded is influenced by the communication of the agent to the local agent server.

To model network load, we assume that an agent consists of u units (classes) of code, each of length $B_\mathrm{c}^k, k = 1, \ldots, u$, some data of length B_d (which at least contains the request of length B_req), and state information of length B_s. A request to load a specific code unit has length B_r for all units B_c^k. The probability of dynamically loading code unit k on server L_i is $P_{L_\mathrm{i}}^k$. By this we can model two aspects. First, it expresses the probability that a specific code sequence is to be executed. Second, we can also model the fact that code is already in a local code cache. On server L_i the agent's data increases by $B_\mathrm{res} \geq 0$ byte.

The migration process consists of marshaling data and state, transmitting data, state, and code to the destination agency, and unmarshaling of data and state information. To model round-trip time, we make the following simplifications. Marshalling and unmarshaling of data is linear in time to the number of bytes and modeled by $\mu : \mathbb{N} \to \mathbb{R}$. For each pair of servers we know throughput $\tau : \mathcal{L} \times \mathcal{L} \to \mathbb{R}$ and delay $\delta : \mathcal{L} \times \mathcal{L} \to \mathbb{R}$ in advance. We assume both τ and δ to be symmetric, i.e. for all $i, j \in \mathcal{L} : \tau(L_\mathrm{i}, L_\mathrm{j}) = \tau(L_\mathrm{j}, L_\mathrm{i}) \wedge \delta(L_\mathrm{i}, L_\mathrm{j}) = \delta(L_\mathrm{j}, L_\mathrm{i})$.

We divide the migration process into three steps. First, the agent migrates from its home agency L_0 to the first server L_1 of the given itinerary. Second, the agent migrates from server L_i to server $L_{\mathrm{i}+1}$, where $i = 1, \ldots, m-1$. Last, the agent migrates back to its home agency. For the following we assume that $L_0 \neq L_\mathrm{i}, i = 1, \ldots, m$. We define $B_\mathrm{c} = \sum_{k=1,\ldots,u} B_\mathrm{c}^k$. $S \in \{\mathrm{pushnext}, \mathrm{pushall}, \mathrm{pullunit}, \mathrm{pullall}\}$ stands for a migration strategy. The network load for a migration from an agent's home agency is calculated by

$$B_{\text{leave}}(\mathcal{L}, S) = \begin{cases} B_{\text{d}} + B_{\text{s}} + B_{\text{c}} & \text{if } S = \text{pushnext} \\ B_{\text{d}} + B_{\text{s}} + |\mathcal{L}|B_{\text{c}} & \text{if } S = \text{pushall} \\ B_{\text{d}} + B_{\text{s}} + \sum\limits_{k=1,\ldots,u} P^k_{L_1}(B_{\text{r}} + B^k_{\text{c}}) & \text{if } S = \text{pullunit} \\ B_{\text{d}} + B_{\text{s}} + B_{\text{r}} + B_{\text{c}} & \text{if } S = \text{pullall} \end{cases} \tag{6.1}$$

A migration from L_{a} to $L_{\text{a}+1}$, with $a \in \{1, \ldots, m-1\}$ has network load of

$$B_{\text{mig}}(\mathcal{L}, a, S) = \begin{cases} B_{\text{d}} + aB_{\text{res}} + B_{\text{s}} + B_{\text{c}} & \text{if } S = \text{pushnext} \\ B_{\text{d}} + aB_{\text{res}} + B_{\text{s}} & \text{if } S = \text{pushall} \\ B_{\text{d}} + aB_{\text{res}} + B_{\text{s}} + \sum\limits_{k=1,\ldots,u} P^k_{L_{\text{a}+1}}(B_{\text{r}} + B^k_{\text{c}}) & \text{if } S = \text{pullunit} \\ B_{\text{d}} + aB_{\text{res}} + B_{\text{s}} + B_{\text{r}} + B_{\text{c}} & \text{if } S = \text{pullall} \end{cases} \tag{6.2}$$

When an agent migrates to its home agency, network load amounts to

$$B_{\text{home}}(\mathcal{L}, S) = B_{\text{d}} + \mid \mathcal{L} \mid B_{\text{res}} + B_{\text{s}}. \tag{6.3}$$

Finally, the whole network load equals

$$B_{\text{MA}}(\mathcal{L}, S) = B_{\text{leave}}(\mathcal{L}, S) + \sum_{l=1,\ldots,m-1} B_{\text{mig}}(\mathcal{L}, l, S) + B_{\text{home}}(\mathcal{L}, S). \tag{6.4}$$

To derive transmission time from network load, it is necessary to consider time for marshaling and unmarshaling of data and state and network latency. All network load must be divided by network throughput. To make the following formulas more lucid, we define the following abbreviations.

The time to load all necessary code units dynamically on server L_{s} from the agent's home agency is

$$\phi_s = \sum_{k=1,\ldots,u} P^k_{L_{\text{s}}} \left(\delta(L_{\text{s}}, L_0) + \frac{B_{\text{r}} + B^k_{\text{c}}}{\tau(L_{\text{s}}, L_0)} \right) \quad .$$

In the case that not only some, but all code units must be downloaded at server L_{s}, we can write

$$\Phi_s = \delta(L_{\text{s}}, L_0) + \frac{B_{\text{r}} + B_{\text{c}}}{\tau(L_{\text{s}}, L_0)}.$$

The time to push code to all agencies equals

$$\varphi = \sum_{l=1,\ldots,m} \left(\delta(L_0, L_{\text{l}}) + \frac{B_{\text{c}}}{\tau(L_0, L_{\text{l}})} \right).$$

The corresponding time for migrating an agent from its home agency is

$$T_{\text{leave}}(\mathcal{L}, S) = \begin{cases} 2\mu(B_{\text{d}} + B_{\text{s}}) + \delta(L_0, L_1) + \frac{B_{\text{leave}}(\mathcal{L},S)}{\tau(L_0,L_1)} & \text{if } S = \text{pushnext} \\ 2\mu(B_{\text{d}} + B_{\text{s}}) + \varphi + \frac{B_{\text{d}}+B_{\text{s}}}{\tau(L_0,L_1)} & \text{if } S = \text{pushall} \\ 2\mu(B_{\text{d}} + B_{\text{s}}) + \delta(L_0, L_1) + \frac{B_{\text{d}}+B_{\text{s}}}{\tau(L_0,L_1)} + \phi_1 & \text{if } S = \text{pullunit} \\ 2\mu(B_{\text{d}} + B_{\text{s}}) + 2\delta(L_0, L_1) + \frac{B_{\text{d}}+B_{\text{s}}+B_{\text{r}}+B_{\text{c}}}{\tau(L_0,L_1)} & \text{if } S = \text{pullall} \end{cases} \tag{6.5}$$

Note, that marshaling and unmarshaling of date and state information each takes $\mu(B_{\text{d}} + B_{\text{s}})$ of time. For example, in the case of a *pullunit* strategy, time comprises of the time for marshaling and unmarshaling of the agent's state, the time to open a network connection to the home agency, the time to transmit all agent's state information to the first agency and to load missing classes from the home agency (ϕ_1).

We define $B^a_{\text{d,s}} = B_{\text{d}} + aB_{\text{res}} + B_{\text{s}}$, which is the amount of accumulated data and state information at server L_{a}. The time to migrate from L_{a} to $L_{\text{a+1}}$, with $a \in \{1, \ldots, m-1\}$ is

$$T_{\text{mig}}(\mathcal{L}, a, S) = \begin{cases} 2\mu(B^a_{\text{d,s}}) + \delta(L_{\text{a}}, L_{\text{a+1}}) + \frac{B_{\text{mig}}(\mathcal{L},a,S)}{\tau(L_{\text{a}},L_{\text{a+1}})} & \text{if } S = \text{pushnext} \\ 2\mu(B^a_{\text{d,s}}) + \delta(L_{\text{a}}, L_{\text{a+1}}) + \frac{B_{\text{mig}}(\mathcal{L},a,S)}{\tau(L_{\text{a}},L_{\text{a+1}})} & \text{if } S = \text{pushall} \\ 2\mu(B^a_{\text{d,s}}) + \delta(L_{\text{a}}, L_{\text{a+1}}) + \frac{B^a_{\text{d,s}}}{\tau(L_{\text{a}},L_{\text{a+1}})} + \phi_{a+1} & \text{if } S = \text{pullunit} \\ 2\mu(B^a_{\text{d,s}}) + \delta(L_{\text{a}}, L_{\text{a+1}}) + \frac{B^a_{\text{d,s}}}{\tau(L_{\text{a}},L_{\text{a+1}})} + \Phi_{a+1} & \text{if } S = \text{pullall} \end{cases} \tag{6.6}$$

Note, that $B_{\text{mig}}(\mathcal{L}, a, S)$ refers to the amount of network load produced by a usual migration using strategy S, see Equation 6.2.

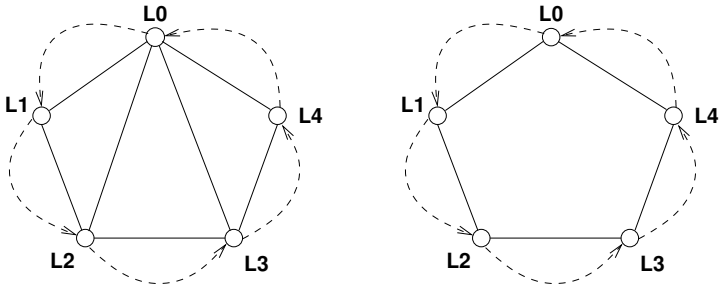
The time to migrate to the home agency is

$$T_{\text{home}}(\mathcal{L}, S) = 2\mu(B_{\text{home}}(\mathcal{L}, S)) + \delta(L_{\text{m}}, L_0) + \frac{B_{\text{home}}(\mathcal{L}, S)}{\tau(L_{\text{m}}, L_0)}. \tag{6.7}$$

Finally, the whole transmission time amounts to

$$T_{\text{MA}}(\mathcal{L}, S) = T_{\text{leave}}(\mathcal{L}, S) + \sum_{l=1,\ldots,m-1} T_{\text{mig}}(\mathcal{L}, l, S) + T_{\text{home}}(\mathcal{L}, S). \tag{6.8}$$

To evaluate an agent's round-trip based on this model, we consider the following two network scenarios, compare Figure 6.2 on the next page. In the first scenario we assume a homogeneous network, where all network connections have the same quality. We assume network bandwidth as $\tau = 800$ Kb/sec and delay $\delta = 5$ ms. In the second scenario we assume a heterogeneous network (ring topology), where



(a) Homogeneous network.

(b) Heterogeneous network.

Figure 6.2.: Examples for the network model used for the evaluation. Agencies are drawn as circles, network connections as solid lines, and agent migrations as dashed lines.

Class	Class size [byte]	Class download probability			
		Sc. 1	Sc. 2	Sc. 3	Sc. 4
1	10 000	1	1	1	1
2	15 000	0	.5	1	1
3	15 000	0	.2	.8	1
4	15 000	0	0	.5	1
5	15 000	0	0	.2	1

Table 6.1.: Code size and download probabilities in four different scenarios.

connections between neighboring servers are as fast as in the homogeneous case, but all other network connections only have a small bandwidth of $\tau = 250$ Kb/sec and delay $\delta = 10$ ms.

The agent consists of five classes, where the first class is the agent's main class and the four other classes are to process specific subtasks and are only necessary on a few servers. The code size of each class can be seen in Table 6.1. The initial data size (B_d) of the agent is 1000 byte and the initial state size (B_s) is 100 byte. The agent has to migrate to four servers, on each server it has to communicate to the local agent server. As a result, the agent's data grows by the value of the server result, which is 3000 byte on each server. A class request has length $B_\mathrm{r} = 20$ byte.

Figures 6.3 and 6.4 on page 98 compare transmission times of four different migration strategies, while varying class download probabilities in four scenarios. The

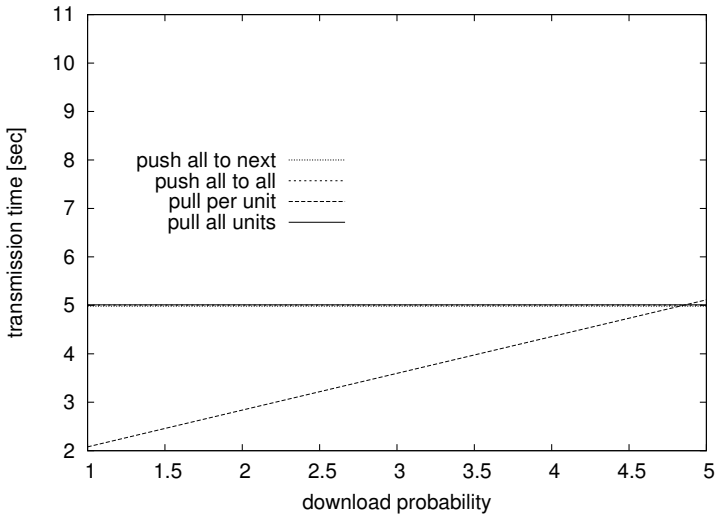


Figure 6.3.: Transmission time vs. class download probability in a homogeneous network for four different migration strategies.

probabilities for class downloading of these four scenarios can be seen in Table 6.1. In the first scenario only one class (the agent's main class) is downloaded, whereas in the last scenario all classes must be downloaded. The second and third scenario model that only a subset of classes must be downloaded.

In a homogeneous network the mobile agent's transmission time using strategies push-all-to-next, push-all-to-all, and pull-all-units are almost identical, because all code is transmitted independent of whether it is needed or not. The additional time to open a network connection and to transmit a code request takes only a few milliseconds when using strategy pull-all-units to the home agency. In contrast, strategy pull-per-unit grows linear in the download probability. It is faster than all other strategies in a homogeneous network, even if more than only one class must be downloaded. Only in the case of downloading all class files, this strategy leads to a higher transmission time, because of several code requests that must be sent to the home agency.

In a heterogeneous network, strategies push-all-to-all and pull-all-units take the same time, again. However, transmission time is higher than using strategy push-all-to-next, because in these strategies all class files must be transmitted using slow network connections, whereas in strategy push-all-to-next, code and data is sent via neighboring network connections. The diagram shows that in a heterogeneous network strategy pull-per-unit is slower than push-all-to-next, even if not all classes must be downloaded. Again, this is because code must be download from the home

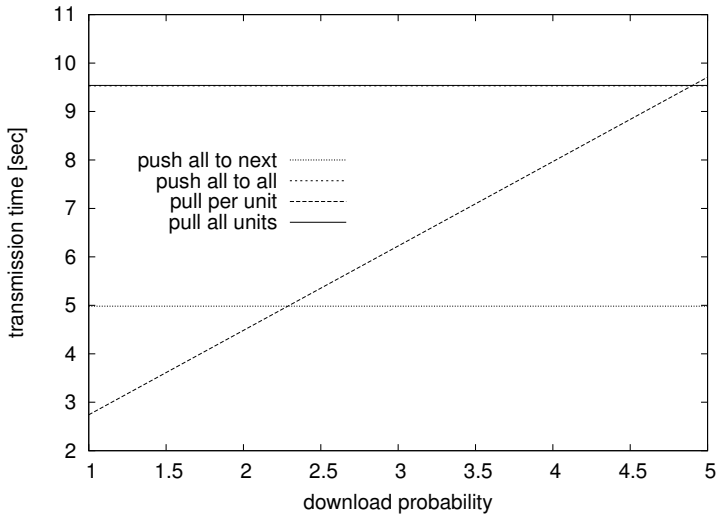


Figure 6.4.: Transmission time vs. class download probability in a heterogeneous network for four different migration strategies.

agency via slow network connections.

What we have learned from this evaluation is that no migration strategy produces least transmission time in every situation. In homogeneous networks, pull per unit is a good strategy, because code downloading is cheap, whereas in a heterogeneous network, code downloading from a far away agency is expensive and should be avoided. In such a network, it is useful to push code in most cases. Of course, the decision for a migration strategy is influenced by several factors, as for example code size, code download probability, etc.

6.3. The Kalong Mobility Model

In this section we will introduce our new mobility model, named Kalong, which is the synthesis of most ideas for improved mobility models proposed in the last two sections. The main feature of Kalong is its flexible and fine-grained migration technique, where an agent or its programmer can define new migration strategies for each single migration. In this section, we confine to the foundations of Kalong without going into technical details. In Part III of this thesis we will then formally describe the migration protocol and explain how to program migration strategies in detail.

Kalong differs from current mobility models in three main aspects:

1. Kalong defines a new agent representation and new transmission units. In our model, mobile agents not only consist of an object state and their code, but have also an *external state*, which comprises of data items that are not part of the object state. A mobile agent's code is no longer transmitted in form of classes or JAR files, but we introduce a new transmission format that we call *code unit*. A code unit comprises of at least one class or many classes, which are supposed to migrate together. A single class can be member of several units.

2. Kalong defines two new agency types, additionally to the already known *home* and *remote* agencies as in current mobility models. We introduce a *code server* agency, from which an agent can download code on demand, and we introduce a *mirror* agency, which is an exact copy of an agent's home agency. It is important to understand that agency types are only valid for a single agent, i.e., a single agency can be a mirror for one agent and a remote for another agent at the same time. A mobile agent can define an agency to be a code server or mirror and later release it again dynamically during runtime.

3. Kalong defines a new class cache mechanism, that not only prevents class downloading in the case of pull strategies, but also code transmission in the case of push strategies. Our class cache is able to avoid transmission of identical classes used in different agents and can distinguish between different versions of the same class.

All these new features are accompanied by new commands for agents to define their own migration behavior.

6.3.1. Agent Representation

We start our introduction with the new agent representation, compare Figure 6.5. As a consequence of the problems with Java serialization process, we allow agents

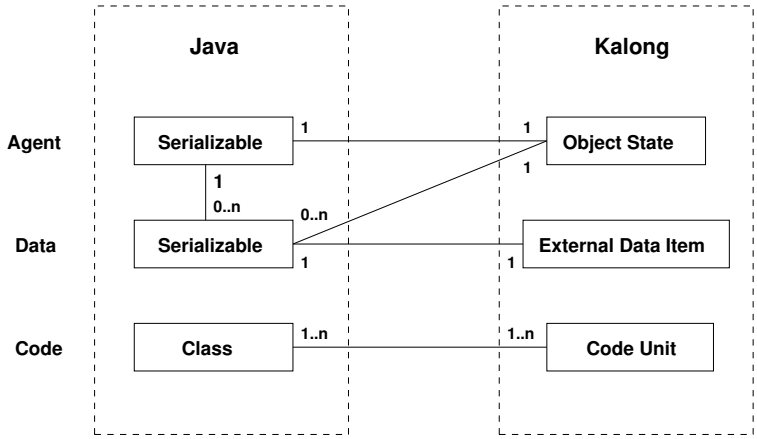


Figure 6.5.: Mapping of the Java agent representation to elements of the Kalong mobility model.

to have other data items besides their object state and we call these data items, the *external state*. Elements of the external state are plain Java objects, which must be serializable. Each data item of the external state must have a name to be stored and accessed by its owner. The external state is private for a single agent instance and it is not possible to share the external state with other agents in the meaning of blackboards for example, even if they were of the same agent type.

We introduce two new commands by which mobile agents can access their external state.

protected void *setData*(*String name*, *Serializable data*)
Store a data item under the given name in the external state.

protected *Serializable getData*(*String name*)
Receive a data item of the external name.

To delete a data item it must be set to **null**.

When an agent migrates, elements of its external state do not migrate automatically, in contrast to the elements of its object state. However, an agent can define one or many items of its external state to be part of the state that is sent to the destination agency. Such data items become invalid at the sender agency if migration was successful. Using this technique of external data items, an agent or its programmer can select data items for migration that will be used at next agencies with a high probability. Data items that most likely will not be used, are not transferred, which in sum can reduce network load.

Data items that were not sent as part of an agent's state, remain at the sender agency only, if it is the agent's home agency. If the sender agency is a remote agency, all data items of the external state will be transferred along the agent's state, as it is not allowed to leave data items at other than the agent's home agency. A data item that remains at an agent's home agency, is not accessible by its owner, if the owner is residing at another than the home agency. Thus, there is no possibility for a remote access. Instead of this, Kalong provides a technique to transmit data items from the agent's home agency to the current agency. First, a data item can be downloaded using method *loadData*.

protected void *loadData*(*String data*)
Load a data item of the external state from the agent's home agency.

When the data item is transferred from the home to the current agency, it becomes invalid at the home agency. No class code is sent along this data transmission. It is possible to transmit multiple data items in one shot, using either a list of names or wildcards.

Data items of the external state can be sent back to the agent's home agency using method *uploadData*.

protected void *uploadData*(*String data*)
Upload a data item from the current agency to the agent's home agency.

After this, the data item is not available at the current agency anymore and it becomes valid at the home agency again. This makes it possible to reduce network traffic by uploading data items that are not necessary at the agencies visited in the near future. It is of course possible that an agent later requests the same data item from its home agency again.

Using the concept of external data items, we not only solve the problem of superfluously transmitted data items, but also the problem of superfluously transmitted code due to the Java serialization technique. As we have seen in Section 6.1, for each element of the object state, code is necessary when deserializing the agent, even if the variable is not used at this agency. Using data items stored in the external state, code that is necessary to instantiate this object at the current agency is downloaded earliest when the data item is deserialized.

A further advantage of external data items outside our main line of argumentation is concerning security. One problem in the area of security problems of mobile agents is the fact that an agent's data must be protected against illicit reading and any manipulation by malicious agencies. Using our technique of an external state, it is possible to leave data items that might be the target of unauthorized reading attempts at the home agency as long as they are really needed. For example, a shopping agent needs its owner's credit card number not until it has found a shop from which the good shall be bought.

Kalong also introduces a new code representation where the basic code transmission unit is not a single class file or a JAR file. As we observed that often several classes migrate together, we introduce a new transmission unit, which we call *code unit*. A code unit consists of at least one or many Java classes, comparable to a JAR file. Classes, which are part of the same code unit should have a common criterion which makes them qualified for a transmission as single unit. A good reason to bundle classes is the same execution probability, for example, because all classes belong to the same subtask. Each code unit has at least one or many *code bases*, from which it can be loaded.

The decision, which class belongs to which code unit, is done by the agent itself, before its first migration. This distribution cannot be changed afterwards as this would contradict some fundamental other aspects of our mobility model. We come back to this issue later. It might seem to be difficult to define such a distribution, but we will show in a later chapter, how this can be done very easily. It is important to understand that such a distribution of classes into units is done by each agent instance itself and that two agents might have different code units, although they belong to the same type.

Code transmission always works on the basis of code units. In the case of push strategies, the agent can define which code units shall be sent to the next agency. If a code unit migrates, all classes of this code unit migrate. Code downloading which is necessary when using pull strategies, works as the following: If a class is needed, for example during the agent deserialization process, it first must be determined to which code unit the class belongs to. This might not be unequivocal, as it is possible to let a single class be part of more than one code unit. Second, it must be decided from which code base the code unit should be loaded. The technique to describe this choice will be explained later.

6.3.2. Migration Process

We will now describe how a migration is processed in Kalong. Kalong provides a very flexible and fine-grained technique to describe the migration strategy of a mobile agent. It is not only possible to define the migration strategy for a type of agents, or a single agent instance, but for each single migration that the agent has to perform. For the moment, we will only introduce the general concept of defining migration strategies in Kalong. A detailed introduction into programming migration strategies will be part of the next chapter of this thesis.

The parts, in which an agent is transmitted during a migration are:

1. State, which consists of the object state and, additionally, some other agent-defined data items of the external state.

2. Code units, which contain the code in form of Java class files.

As already said, it is not necessary that the agent carries all data items of the external state as well as all code units. It is possible and sometimes advisable to sent only code units and no state information at all, which will make the destination agency a code server – we will explain this later in detail. If only state information is sent, then the agent uses a pull strategy. In this case, at least a description of all code units that contains at least the names of all classes and code bases are sent to the destination agency, see Section 6.3.4.

After an agent has left its home agency, no information about this agent is deleted, except the object state and data items that were part of the state. At the destination agency, code units are received and stored so that the agent's class loader can access code using a class name. If state was sent during the migration, then the agent is deserialized. Classes not already available at the destination agency must be downloaded as described above.

When an agent migrates from an arbitrary remote agency to another agency, it can define a new migration strategy in terms of state and code units. The agent is free to define which data items shall be part of the agent's state and which code units shall migrate, with one exception concerning data items. As it is not allowed to leave data items at an arbitrary remote agency, the migration strategy must define for each data item whether it shall be part of the state or be uploaded to the agent's home agency. We define a rule, that all data items that are still valid at the current (remote) agency when an agent migrates, are mandatory part of the state. With code units, we do not have this problem, as it is without danger to delete them at the remote agency, after a successful migration. After an agent has left a remote agency, all information about the agent is deleted. Thus, this agency cannot be used for code unit downloading in future.

It should be obvious that using these two primitives of state and code unit transmission, together with the ability to define which elements of the external state should be part of the agent's state, it is possible to describe all migration strategies that we have introduced in the last chapter. For example, to describe the push-all-to-next strategy, we define all classes to form a single code unit, which is sent along with the agent's data to the next agency. To describe the pull-per-unit strategy, we define that each class forms a single code unit and none of them is transmitted along agent's code.

6.3.3. Types of Agencies

In the last section we have already mentioned that Kalong also defines some new types of agencies. So far, mobile agents can only migrate from their home agency to visit the so-called remote agencies. One very important rule we introduced in the last section was that all information about an agent is deleted at a remote agency after the agent has left it. Now, we will introduce two new types of agencies, which

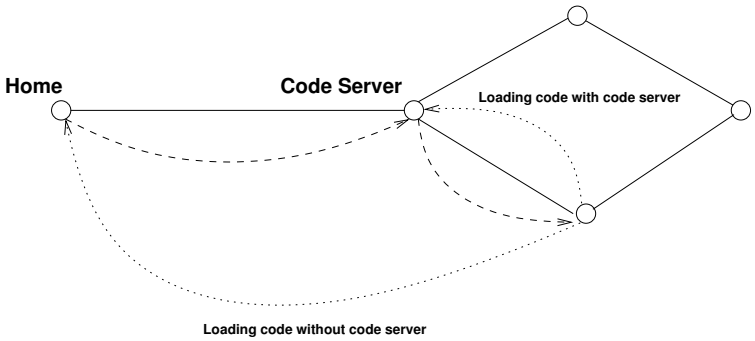


Figure 6.6.: Example to show the advantages of a code server. Agencies are drawn as circles, network connections as solid lines, migrations as dashed lines, and code requests as dotted lines.

both are able to keep or remember data and/or code for a single agent.

First, we introduce the *code server agency*. When an agent leaves a remote agency, it can define within the migration strategy that some code units shall be stored at the current agency. Code units must be already available to be stored, so that they must have been downloaded by the migration strategy before, if necessary. If at least one code unit is copied, this agency becomes a code server agency for this agent. The effect is twofold: First, the name of this agency is added to this list of *code bases*. Second, after a successful migration, these code units are not deleted and can in future be downloaded from this agency. As already mentioned, the agent must decide from which code base units should be downloaded and this will be explained later. The range of possible strategies goes from a simple one, that always uses the last code server agency defined, to very complex one, that consider network metrics and compares the cost of downloading code units from different agencies.

Using code server agencies, an agent has the chance to deposit code at several agencies that are near to the ones it will visit in the next future. For example, in Figure 6.6, we assume the cluster of agencies on the right side to be in the USA, whereas the left side shall be in Europe. When the agent now migrates in the USA and it is known that many servers should be visited over there and if additionally it is also worthwhile to use code downloading at all, then the agent can define a code server there.

This is not possible with any other mobility model currently available. In Aglets it is not possible to define something similar to a code server at all. All classes can only be downloaded from a single code base which must be defined during agent creation. In Grasshopper it is possible to define multiple code server, but also only when starting the agent and not during runtime.

One question remains open and that is about how to release code servers again. Therefore, we introduce the technique of sending commands between agencies. To send a command to another agency is a new primitive that we have not introduced so far. Using this command it is possible for the agent at any point to release a code server defined before. This has the effect that all code units are deleted and this agency is deleted from the list of code bases of all units.

Second, and as a direct consequence of the concept of a code server for *code* we introduce the *mirror agency*, which can keep information about *code and data*. The mirror agency is a copy of all data items of the external state and all code units. If a mirror agency exists it takes the role of the home agency as long as it is defined. The necessity of a mirror agency becomes obvious when looking at the example above, and assuming that the agent wants to use downloading and uploading of data items. It would be very expensive, if the agent would be forced to communicate to its home agency for exchange of data items. Therefore, an agent can define a mirror agency, which has the consequence that automatically all data items and all units of the home agency, which are not already at the current agency, are downloaded to the new mirror agency. All data items are set invalid at the home agency.

If the agent defines an agency to be a mirror and there already exists a mirror agency, then all data items and all units must have been loaded from the old mirror before. The last mirror agency must be released by the agent, which is done by sending a command to this agency as described above for code servers.

6.3.4. Code Cache

The third important aspect of our new mobility model is the code cache. Code caching is a technique to decrease network load by avoiding class transmission between two agencies in the case where code is already available at the destination agency. We already became familiar with the Java code cache technique, which is implemented as part of the class *ClassLoader* and which is able to avoid multiple downloading of code for the same agent instance. In contrast to this technique, our class cache shall not only work for a single agent instance, but shall be able to share classes between several agents.

The cache works on the basis of classes and not units, because it could happen that agents of the same type use different unit definition, so that for one agent instance a specific class is in one unit and for another instance it is in another unit. If we use a cache on the basis of units, it would only work for a single agent instance and therefore would be quite useless.

The goal of our code cache is to check during the migration protocol, whether code that belongs to the migrating agent is already available at the destination agency, without sending the whole code. We use the technique of *digests* or *hash values* to check whether two classes are equal or not. A digest is a sequence of bytes of fixed

length, for example 16 byte in the MD5 [Rivest, 1992] algorithm, which is produced from a stream of data of variable length. A digest algorithm must assure that it is computationally infeasible to find two data streams that produce the same digest[2]. As part of the SATP migration protocol, which we will introduce in the next chapter in detail, the sender agency transmits the so-called *Agent Definition Block* to the receiver agency. This block contains information about all units, all classes within these units, a digest for each class and information about code bases from which units can be downloaded. At the destination agency, each class is now checked against the local class cache. If it contains a class with an equal name and equal digest, then it can be assumed that code for this class is already available. The destination agency informs the sender about this fact by a specific reply message, which should the sender let neglect to send this class. In all other cases, code for this class is not available yet, and the sender is informed to send the code.

It then depends on the migration strategy, which units resp. classes are really send to the destination agency afterwards. If, for example, a unit is not pushed from the sender agency, and code is not yet available at the destination, then it is inevitable to pull code for these classes later. In the other case, if code is already available at the destination, we will discover this using our cache algorithm and then we can neglect to send code for these classes at all.

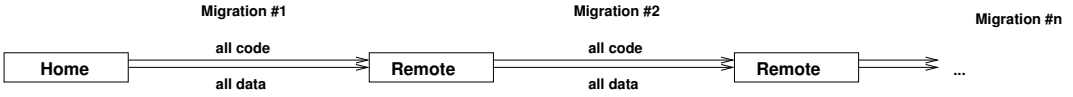
6.4. Summary of Part II

In this part of the thesis we first introduced a general framework for the migration process of mobile agents. We discussed design issues of the migration process and proposed the concept of a *mobility model* to describe the features of a mobile agent system with regard to agent mobility. After that, we discussed the drawbacks of the simple migration techniques used in today's mobile agent systems and discussed in detail possibilities to improve the performance of the migration process. As part of this discussion we extended our mathematical model for network load and migration time in order to compare different migration techniques. Finally, we proposed our new mobility model, named Kalong, which gives the programmer of mobile agents more possibilities to influence their migration process.

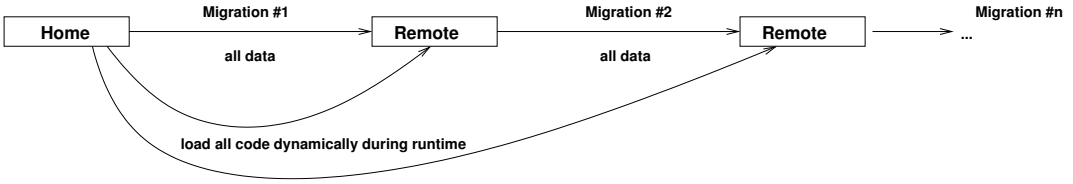
Until now, mobile agent systems only provide very simple migration strategies. The *push* strategy always transmits all code classes of the agent, together with the agent's state to the next destination, compare Figure 6.7(a). In contrast, the *pull* strategy never transmits any code class, but only the agent's state and imposes the task of downloading code on the receiving agency, compare Figure 6.7(b).

Using the new Kalong mobility model, the agent has the opportunity to select

[2]This does not mean that it is impossible, but the probability is very low that there exists a pair of x, y for which $H(x) = H(y)$, when H is the digest (hash) function.



(a) Push Migration Strategy



(b) Pull Migration Strategy

Figure 6.7.: Traditional migration strategies

classes that should be transmitted to the next destination agency, while other classes can be downloaded from the agent's home agency later. Also with regard to data items, Kalong provides new functions to select which parts of the agent's data state should be sent to the destination agency, while others remain at the home agency, compare Figure 6.8. If the agent needs a specific data item at an agency that is not yet available, it can be downloaded from the agent's home agency. The advantage of this technique is a reduction of network load, because the data items itself and the corresponding class code is only transmitted, if the data item is really used. We call this feature the *adaptive transmission of code and data*, which gives the agent programmer the chance to react to certain execution or network scenarios. No other mobility model currently allows the programmer to influence the migration process to such an extent. The necessity for adaptive transmission was already discussed: For example in case of low-bandwidth and unreliable network connection the agent should migrate with all its classes to avoid dynamic class loading later. However, if it is already known from the current execution state of the agent that specific classes or data items will not be used under any circumstances at the next destinations, their transmission is superfluous and should be avoided. With Kalong it is possible to implement migration strategies that take such rationals into account.

The second advantage of Kalong, as compared to all other mobility models, is its capability to dynamically define *code server* and *mirror* agencies. All other mobility models only distinguish between the agent's *home* server and the *remote* servers, which are all servers that the agent visits. The home server has the very important

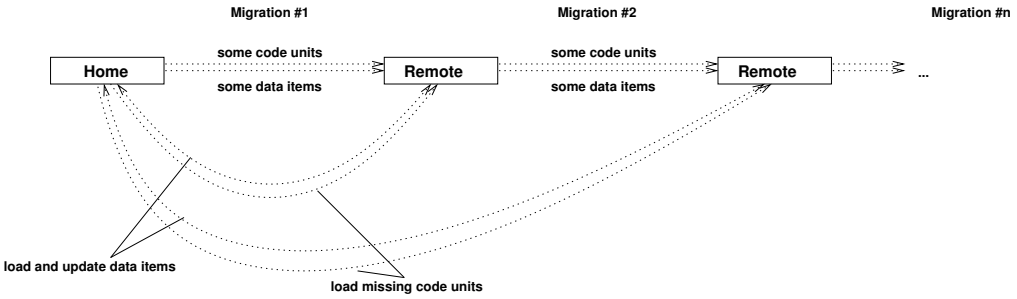


Figure 6.8.: Adaptive transmission of code and data in Kalong.

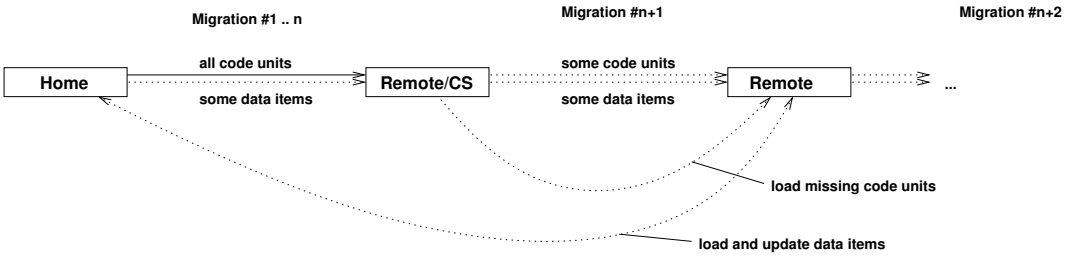
role to provide all of the agent's code so that it can be downloaded from this server later. In most mobile agent systems the home server is also the only server that provides the agent's code. In some systems the programmer can manually deploy agent code to other servers too.

In Kalong it is possible that the agent dynamically defines a server to become a code server, compare Figure 6.9. The effect is that all code of the agent is copied to this server and, therefore, can be also loaded from this server in the future. The advantage results from the fact that it is faster to load code from a near-located agency than from the far away home agency. The agent can decide in relation to its itinerary for example, which server should become a code server. When the agent terminates, it has to release all code servers to free resources.

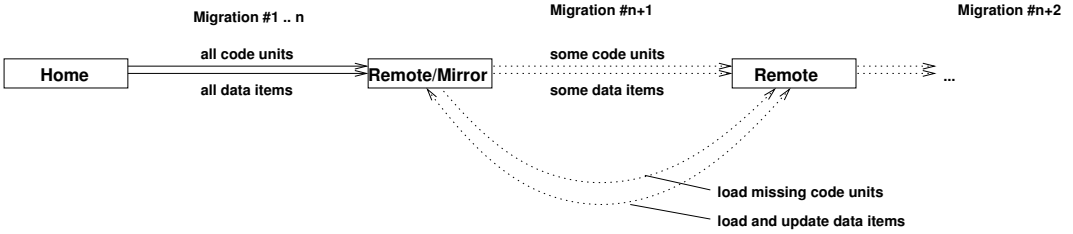
A mirror agency is an extension of a code server agency, where not only agent's code but also selected data items of the external state are moved to this agency. A mirror agency completely overrides the existence of a home agency, so that all data and code loading requests are directed to the mirror instead of the home agency. Like a code server, a mirror server should be used reduce the time for code and data transmission.

The third advantage of Kalong is a comprehensive technique for code caching. Before any class is transmitted to a destination agency, it is verified whether exactly this class is already available there. If the destination agency already has this class, the sender agency must not send it again. Using this technique, network load and transmission time can be decreased in the case that many agents of the same type (using the same classes) roam the network and visit the same agencies. The equality of classes is checked using a hash value, which guarantees that also different versions of the same class can be distinguished.

As far as we known, there is only a single paper available in literature that also discusses the concept of an adaptable migration process. Picco proposed a lightweight and flexible mobile code toolkit named μCode [Picco, 1999]. The main principle of



(a) Code Server.



(b) Mirror.

Figure 6.9.: Further advantages of the Kalong migration model.

μCode is the flexibility to control code relocation. The unit of transmission is called *group*, which can contain single classes, class closures, and objects. The programmer can choose which classes and objects shall be part of the next migration – a technique which is comparable to the possibilities a programmer has with Kalong. A migration is started by invoking method *ship* of class *Group*. Classes that are not already available at the destination server are downloaded from a single server that is given as parameter in method *ship*. μCode does not provide the possibility to load data items dynamically during runtime or to update data items at the agent's home server. It does not allow to define more than a single server from which code can be downloaded, and, therefore, does not allow to define code servers during runtime. Due to the lack of individual data transmission, μCode also does not provide a mirror concept. The source code of μCode is available as an open source project at Sourceforge[3].

The idea of introducing *code servers* to load code from near-distance agencies instead of the home agency, was presented by Hohl et al. [1997]. However, their

[3]Visit `http://mucode.sourceforge.com` for more information.

concept can be named static, as code server must be initialized manually be the agent programmer and can not by initialized by the agent during runtime. We already discussed this paper in Section 6.2.

Other papers available do not focus on the adaptation of the general migration process, but on optimizations for specific aspects of mobile applications. For example, Tanter et al. [2002] discuss the problem of determining the data items that a mobile application (which might be an agent) should take along during a migration or leave at the source environment to be accessed remotely. They explain techniques a programmer can use to specify the type of data migration for each instance. The authors work towards a technique, where the kind of migration can be exchanged dynamically during runtime.

Another paper discusses techniques to determine the itinerary of a mobile agent during runtime [Satoh, 2001, 2002] using the new concept of mobile agents being the provider for the migration service in a mobile agent system. The MobileSpaces system is a framework for building network protocols for migrating mobile agents over the Internet. It is characterized by two new concepts. First, mobile agents are organized in a hierarchy, which means that agents can contain other agents, resulting in a tree structure. Second, mobile agents can migrate to other mobile agents (inter-agent migration) as a whole, together with all their inner agents. A mobile agent migrates into another agent, which itself implements a network protocol for migration. The author only describes applications of this system on the level of route determination, and not the lower level of an optimized agent transmission.

Part III.

The Kalong Mobility Model And Its Implementation

In the last part we identified the migration process as a very resource-critical part of mobile agents and we described several techniques to improve the migration process. In Section 6.3 we gave a first informal description of our new Kalong mobility model, which is our suggestion to enhance the migration process of mobile agents. This part of the thesis is entirely devoted to a detailed specification of Kalong as mobility model and a description of its implementation as the Kalong software component.

7. Specification of the Kalong Mobility Model

This chapter specifies the Kalong mobility model and the SATP migration protocol. The reader may skip this technical chapter and continue with the description of the Kalong software component on Page 147 or with the evaluation of the Kalong mobility model in Part IV on page 201.

7.1. Introduction

This specification defines the Kalong[1] mobility model. Kalong provides an efficient technique for migration of mobile agents between computer platforms. It is designed to be portable between different machines, operating systems, network architectures, and network transport protocols. In the current version, Kalong bases on the Java programming language. It is planned to port Kalong to other programming languages later.

Kalong is supposed to be embedded in an agency software and to communicate with three other components, compare Figure 7.1.

1. The *agent manager* is responsible for agent execution and other basic functions of a mobile agent server. It communicates with Kalong to conduct a migration. In the other direction, Kalong notifies the agent manager about received agents.

2. The network is not directly accessed by Kalong but by using a *network adapter* component that abstracts from details of a network protocol and works as a dispatcher for a set of different protocols. Kalong communicates with the network adapter using a very small interface, which only provides functions to open and close a network connection and to send and receive byte sequences. For each network protocol, the network adapter launches a server that listens to a network port for incoming migrations. The network adapter informs Kalong about incoming requests.

[1] Kalong is the name of a fruit-eating flying fox, lat. *Pteropus vampyrus*, inhabiting Java island. The Kalong is remarkable for its span and its flying speed. The latter was the reason to choose this name for our mobility model, besides the relation to Java.

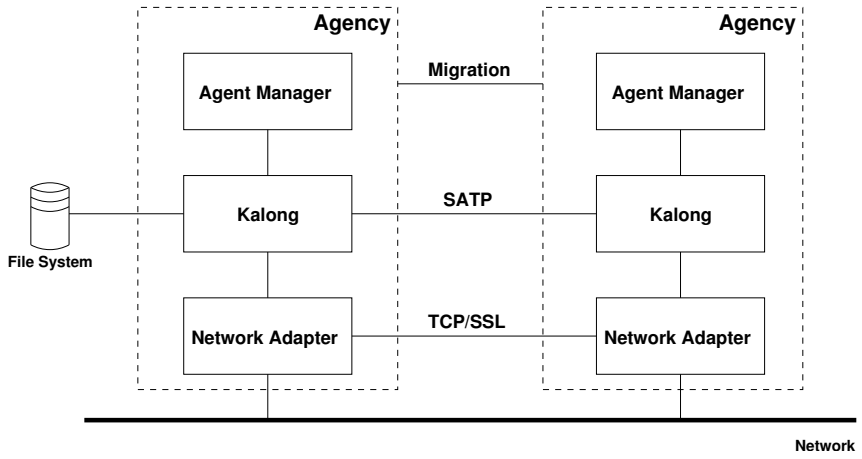


Figure 7.1.: Kalong and its environment.

3. Kalong must have access to a file system to load class files. Instead of a file system, classes can also be loaded from remote network resources, accessible using a URL address and a HTTP or FTP connection. The latter is not pictured in Figure 7.1.

Kalong defines a new migration protocol, named *Simple Agent Transport Protocol*[2] (SATP), which is an application level protocol to transfer agent information to a single destination agency. SATP works according to a simple request/reply scheme and must be used on top of a connection-based network protocol that provides a reliable flow of data, for example TCP. It can be embedded in any other application-level protocol that also uses TCP, for example SSL, HTTP, SOAP, etc. This specification defines SATP commands as well as the message format to be used, in contrast to the specification of SMTP for example, where the definition of the message format is moved to a companion protocol (RFC 822).

Kalong is responsible for the whole migration process and all tasks related to agent migration. Especially, Kalong must provide functions to serialize and deserialize agents, it must define an own class loader object for each agent, which directs requests to load classes back to Kalong. Kalong is *not* responsible for any kind of agent thread management. In the case of an agent migration, it is the agent manager that must control thread suspension and guarantee that no agent thread is able to still modify

[2]The name was chosen in tradition to other application-level protocols, like *Simple Mail Transfer Protocol* and *Simple Network Management Protocol*, and to distinguish it from Aglets' *Agent Transfer Protocol* (ATP). Unfortunately it happened that SATP became more complex than ATP.

the agent's state. After an agent has been received successfully, Kalong only notifies the agent manager to start the new agent.

In detail, Kalong defines the following models, interfaces, and protocols:

1. An agent model that introduces the concept of *external state* and a new level of granularity for class transmission, see Section 7.3.

2. An interface (*IKalong*, see Section 7.4 on page 122) for the agent manager to conduct a migration, an interface (*IAgentManager*, see Section 7.5 on page 134) for Kalong to access the agent manager.

3. An interface (*INetwork*, see Section 7.6 on page 134) for Kalong to use the network adapter, an interface (*IServer*, see Section 7.7 on page 135) for the network adapter to access Kalong.

4. A migration protocol, named SATP, which defines messages and their format sent over the network, see Section 7.8 on page 135.

As compared to our definition of mobility models, the focus of Kalong lies on issues of the agent's and network's view. Kalong's new agent model partially refers to the programmer's view too. However, Kalong does not define anything related to the mobility level (which is part of the programmer's view) for example, and it is the task of the agent manager to map requirements of the mobility level to the preferences of Kalong as described in the following sections.

7.2. Kalong Vocabulary

This specification uses the following terms:

Agent A mobile agent as used in this thesis. We will often use the term *agent instance* to denote a single agent object in contrast to the set of all agents of the same type, or the agent's type.

Agency Software that is necessary to execute and migrate mobile agents on a computer system. We distinguish between the *sender agency* which starts a *transfer* and the *receiver* agency, to which the *transfer* is directed.

Agent Manager Subcomponent within an *agency*, which conducts a migration process.

Connection A network connection is a virtual communication channel between two computers that is used to transmit SATP *messages*.

Context Kalong maintains for each agent a *context* data structure that comprises of all information necessary for Kalong, e.g. its name, its home agency, its data units, its state, etc.

Message The basic unit of SATP communication. A message is a sequence of bytes matching the syntax as described in the following sections.

Migration Is a special form of a *transfer*, which always has the consequence that agent execution is stopped at the sender agency and resumed at the destination agency. We define two new verbs to describe the direction of a migration. When an agent leaves an agency, then it *migrates out* and when an agent is received by an agency, then it *migrates in*.

Migration Strategy A migration strategy defines what agent information should be sent to which agency. A migration strategy can consist of one or many transfers. The agent manager defines the migration strategy by using the methods of interface *IKalong*.

Object State The *object state* or *agent object state* is equal to the serialized agent.

Request A request is a SATP *message* which is sent from a sender to a receiver agency.

Reply A reply is a SATP *message* which is sent as answer to a request from the receiver agency to the sender agency.

Transaction A *migration strategy* might consist of several *transfers*. As a migration strategy must be an atomic process, which is either completely executed or not at all, Kalong provides transaction management according to a *Two Phase Commit* (2PC) protocol.

Transfer The process of transferring agent information from a single sender agency to a single receiver agency. If the transfer includes an agent's state information, then it is a *migration*.

URL Each *agency* must have one or many addresses in form of a URL. Kalong does not require a specific format of URLs, as addresses are only forwarded to the network adapter.

7.3. Agent Model

7.3.1. Agents and Agent Contexts

In Kalong, an agent must be a Java object of type *Serializable* or any subclass, as an agent's object state must be marshaled to be sent to a destination agency. The

object state comprises of all agent's attributes, which must be serializable too.

Agent Names

An agent must have a globally unique name that does not change during its life-time. To be globally unique means that there must not be two agent instances with the same name in the whole agent system independent of time. Kalong does not specify how to obtain such a name and does not define any structure for a name, except that a name must be codeable into a Java *String* object. Kalong can only locally and temporally verify that no two agent instances with the equal name exist, when new agents are registered with Kalong and when agents migrate in. The agent manager is responsible to guarantee uniqueness by an appropriate algorithm to generate agent names.

Data Items

Besides the agent's object state, agents also have an *external state* which is defined as a set of serializable Java objects which are accessible by the agent but are not part of the object state. Each element of the external state must have a unique name to be stored and accessed by its owner. A name must be codeable as Java *String* object. Data items of the external state must not be accessible by other agent instances. A single data item should not be shared with another agent instance, as it is copied in order to migrate it.

Each data item has a status that can be *defined* or *undefined*. If a data item is transferred to another agency, it is locally set to *undefined* and set to *defined* at the destination agency. Thus, it is possible to let data items remain at the home agency for example, although the agent has migrated to another agency. The agent can request those data items from its home agency later. Kalong must assure that a data item currently set to *undefined* is never read or written by the agent manager.

Code Units

An agent's code is transferred in form of code units. A code unit contains one or many classes, which will always be transferred together. Each code unit has a locally unique identifier, i.e. no two units of the same agent instance have the same identifier.

Each agent instance defines its own distribution of classes onto units. It is allowed to include the same class in more than a single unit. It is the task of the agent manager to make sure that classes are completely spread onto units. Code distribution cannot be modified, after the the *agent definition block* (see below) was sent the first time. Each code unit has a list of *code bases* from which it can be loaded. A code base is an agency and is described by one or many URL addresses. The home agency should not be member of any code base as this would be redundant.

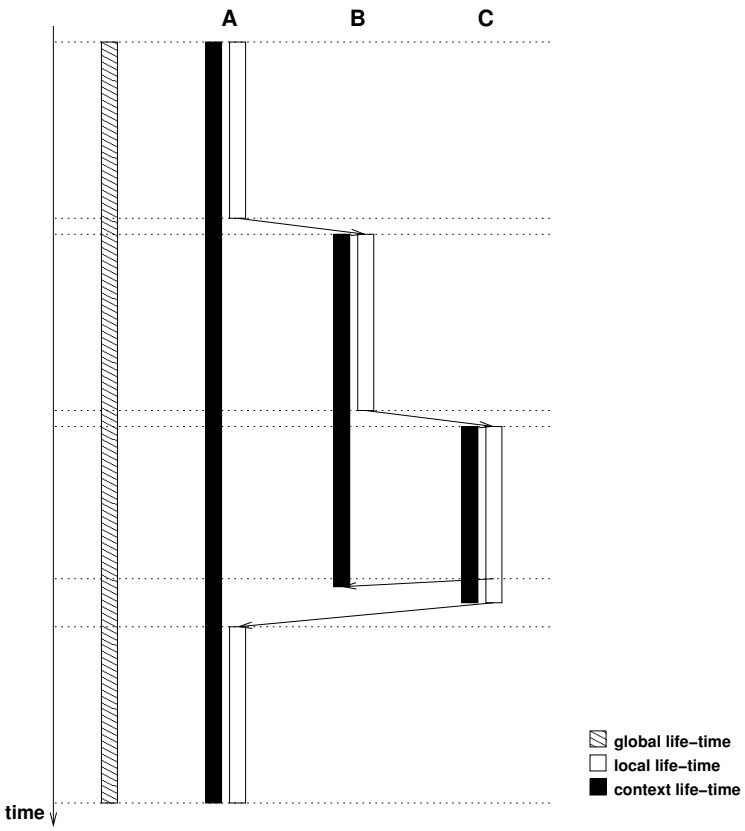


Figure 7.2.: Comparison of the global and local life-time of a agent with the life-time of an agent's context.

Agent Context

All information about the external state and code units of an agent are stored within Kalong in a data structure named *agent context*. There exists a single agent context for each agent instance. Kalong must provide methods to access data items of the external state by the agent manager, but Kalong does not specify how an agent manager provides access to its data items for an agent instance. Kalong does not define the detailed structure of an agent context. All implementation details are left to the programmer.

It is important to understand the difference between the life-time of an agent instance and an agent context. An agent is created by a user within the agent manager. It has a *global* life-time which lasts from its creation till its termination (which is not necessary at the same agency). The time visiting an agency, starting

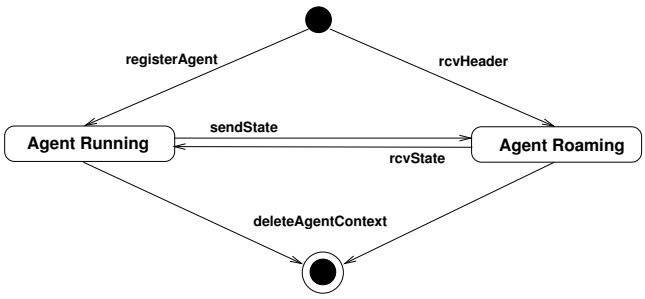


Figure 7.3.: State diagram for an agent context.

after the agent has migrated in and stopping after the agent has migrated out, is called the *local* life-time. Thus, the local life-time is bound to a single agency.

Agent context objects are created when the agent manager registers an agent with Kalong or when an agent migrates in. However, the life-time of an agent context might not terminate with the local life-time of an agent. For example, a home agency is supposed to keep information about code units so that the agent can load necessary classes later. We will see in the next section two more examples for agencies that retain agent information, even beyond an agent's local life-time. To summarize, the life-time of an agent context is not bound to either the local or the global life-time of an agent, but it is at least as long as the local life-time of the corresponding agent.

This can be best explained using an example, compare Figure 7.2. The figure shows global and local life-time for an agent visiting three agencies A, B, and C. An agent's life-time is drawn as striped bar, an agent's local life-time is drawn with non-filled bar, and an agent's context life-time is drawn with a solid bar. The agent is created at agency A, which automatically becomes the agent's home agency. It now migrates to two other agencies, named B and C and finally returns to agency A to terminate there. After the agent was created, also an agent context object is created for it. When the agent migrates to agency B, the local life-time terminates, but the agent context's life-time continues. At agency B, a local life-time and an agent context's life-time are started when the agent migrates in. After it has been migrated out, the local life-time terminates, whereas the agent context's life-time continues. The reason for this is that the agent has defined agency B to become a code server, i.e. some code units of this agent are still available at agency B for later download. Before the agent leaves agency C, it releases the code server again, which terminates the context's life-time at agency B. Finally, after the agent has returned home, the context's life-time at agency A terminates too.

At last, we can also describe the life-time of an agent context using a state diagram, compare Figure 7.3. After creation, an agent context is first in state *Agent Running*,

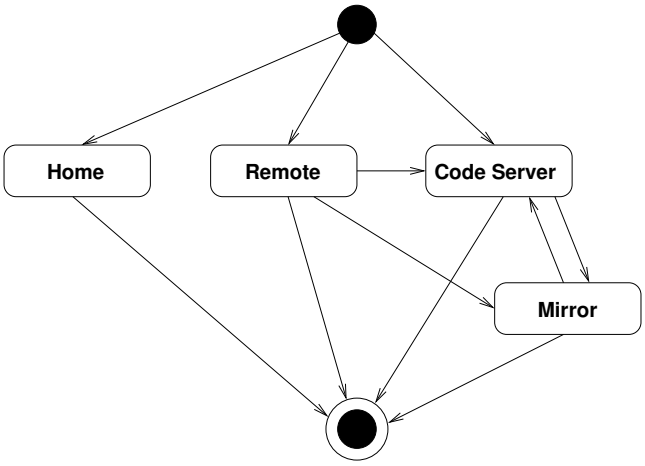


Figure 7.4.: The four agencies types in Kalong and how agency types can change during life-time.

which indicates that the agent is currently been executed. After the state was sent to another agency, the context switches to state *AgentRoaming*, which means that the agent is not executed at this agency any more. If an object state is received later, it switches back to the first state again. On a remote agency, the context is first in state *AgentRoaming* (because we allow to initialize a remote agency as code server by only sending code units), until an object state is received. Then, it switches to state *AgentRunning*.

7.3.2. Agencies

An agency is the place where agents are executed. Kalong does not specify how an agency is further structured, for example using the concept of logical *places*. This must be done within the agent manager component.

Each agency must have at least a single address in form of a URL, for example: *tcp://tatjana.cs.uni-jena.de:4155/whyte/penthouse*, where *tcp* is the protocol name, the number 4155 is the port number on which the agency can receive agents using this protocol, and *whyte/penthouse* is the name of the place to which the migration is directed to. Kalong does not specify any format of this URL, but it requires that is consists at least of a protocol name, a host name, and a port number. All other parts of a URL are not considered by Kalong.

Addresses are obtained by the underlying network adapter component, which manages a set of different network protocols and defines a server for each protocol listening to a specific port. Therefore, agencies can be addressed using many different

Agency Type	Data items	Code units	Cardinality
Home	✓	✓	1
Remote			1
Code server		✓	0..n
Mirror	✓	✓	0..1

Table 7.1.: Comparison of the four agency types with regard to their ability to store data items and code units and the number of agencies of this type.

URLs. For example, the same agency can be addressed using *tcp://tatjana.cs.uni-jena.de:4155* and *ssl://tatjana.cs.uni-jena.de:4156*. The first address must be used to communicate to this agency over a plain TCP connection, whereas the second URL must be used to have a secure connection using the SSL protocol. The addresses of an agency must only differ in the protocol and the port number, all other URL elements must be equal. This requirement must be verified by Kalong.

Kalong distinguishes the following roles for agencies from the view of a single agent instance. This role information is transparent for the agent manager and only used within Kalong, thus, it might be a little bit confusing that we speak of *agency roles* where we really mean *Kalong roles*.

1. The agency on which the agent was started becomes automatically the *home* agency. An agent must only have a single home agency and an agent's home agency must not be redefined. The home agency does not delete any information about the agent except of the agent's object state, after the agent has migrated out.

2. A *remote* agency is every agency that an agent visits while executing an itinerary. Usually, a remote agency drops all information about an agent, after the agent has left it.

3. A *code server* agency is able to store code units, even after the agent has left the agency. Thus, code can be downloaded from this agency later. There can exist multiple code server agencies in parallel.

4. The *mirror* agency is able to keep information about code unit and data items, even after the agent has left the agency. The mirror is a complete copy of all information stored about an agent at the home agency. There exists always only a single mirror agency at a specific point in time.

The role of an agency with regard to a specific agent instance might change during its life-time as can be seen in Figure 7.4. An agency becomes a home or remote agency by starting or receiving an agent. A home agency releases its role, when

the agent terminates or the agent manager deletes the corresponding agent context object. A remote agency releases its role when the agent leaves it and migrates to another one, or the agent defines this agency to become a code server or mirror agency. Code server and mirror agencies release their role only by an agent's order. A mirror agency can be defined to become a code server afterwards. We show in the next section, how an agent can define a code server or mirror agency. Table 7.1 compares the agency types with regard to their ability to keep information about agents.

7.4. Interface IKalong

This interface defines the main functions of the Kalong component. It is used by the agent manager to access an agent's context and to define a migration strategy. The protocol definition is given as a set of methods with arguments and results in Java syntax. A description of the function of each method should provide enough information to allow its implementation.

The agent manager at the sender agency communicates with Kalong in two phases. First, it registers an agent with Kalong, so that Kalong can verify that all used classes are available for transmission and the agent itself as well as all its components are serializable. Second, it uses the services of this interface to conduct the migration process.

We divide the methods of *IKalong* in four groups:

1. Methods to start and stop transactions.
2. Methods to register agents and define code units.
3. Methods to modify a local agent context.
4. Methods to transfer agent information via network.

Almost all methods receive an agent name as parameter to identify the agent context. All methods of interface *IKalong* are assumed to work synchronous.

In the following the term *current agency* always refers to the agency on which the commands are executed.

7.4.1. Manage Transactions

A migration strategy might comprise of several transfers. As it must be an atomic command that is only executed with all its transfers or none at all, Kalong provides a common technique for transaction management, called *two-phase-commit* (2PC) protocol.

The idea of the 2PC protocol is the following. All commands that modify the agent context information either locally at the current agency or remote at any of the destination agencies, must be bracket by the following transaction management commands. To start a transaction, the following method must be used.

public void *startTransaction*(*String agentName*)**throws** *KalongException*
Start a transaction. Throws an exception, if a transaction is running already.

After this method was called, no other thread can start a transaction using this agent, until the transaction terminates.

To explain the 2PC protocol, we assume a case where several connections have been opened to different remote agencies. After all messages have been sent, the agent manager must send a so-called *prepare* message (first phase) using the following method.

public boolean *prepare*(*String agentName*)
Send a prepare message to all receiver agencies. The reply informs about success.

Kalong maintains a list of all connections that were opened since the last call of method *startTransaction*. Method *prepare* sends a *prepare* message to all receiver agencies. Each receiver answers, whether the last transfer was successful or not. Method *prepare* collects these reply messages and returns **true** if all receiver agencies have accepted the transfer and **false**, if at least a single agency has not accepted the transfer.

The agent manager must now send either a *commit* or *rollback* message to all receiver agencies (second phase) with regard to the result of the previous method. Both methods close all network connections and terminate the transaction.

public void *commit*(*String agentName*)
Send a commit message to all receiver agencies.

A commit message applies all changes made during the last transfer. This might have the consequence that an agent is started at a remote agency.

public void *rollback*(*String agentName*)
Send a rollback message to all receiver agencies.

A rollback message recovers at the receiver agency the last stable state before the last transfer was started. Of course, the last three methods must assure that a transaction has been started before. If no transfer was done since the last start of a transaction, method *prepare* must return **true**.

7.4.2. Information about Agencies and Defining Agent Contexts

As already described, agency addresses are defined by the network adapter component. As this information is the only one that must be available at the agent manger about the network layer, Kalong must pass on this information.

Using the following method, all addresses of this agency can be requested.

public *java.net.URL*[] *getURLs*()
Return an array of URLs of the current agency or **null**, if this agency has no addresses yet.

It simply uses the corresponding method of *INetwork* to obtain this information, see Section 7.6 on page 134.

Before agents can migrate, they must be registered with Kalong. If this method is successful, a new agent context exists in Kalong which can be accessed using the given *agentName*. If the method fails for any reason, no agent context is created and an exception is thrown.

public void *registerAgent*(*String agentName*, *Serializable agent*)**throws** *KalongException*
Registers an agent with name *agentName* and object state *agent* with Kalong.

This method first checks whether an agent context with the given name *agentName* already exists and throws an exception in this case. The current agency becomes the home agency for this agent. All classes of the class closure are determined by analyzing object *agent*. It throws an exception, if no address of the current agency can be determined.

After context creation, the code units for this agent must be defined, before any migration can happen.[3] The main method to define code units is the following.

public int *defineUnit*(*String agentName*, *String*[] *classNames*)
Defines a new unit with classes *classNames*. Returns the code unit identifier.

All classes whose name is given in the array *className* are bundled into a single code unit. Parameter *classNames* must not be **null** or the empty array. Kalong must assign a unique identifier to this unit, which is returned to the caller. The definition of code units cannot be changed or modified afterwards. The agent manager is responsible that all necessary classes of the class closure are distributed on code units. Classes that are not part of any code unit cannot be transferred to remote agencies, either by pushing or pulling. The agent manager can request the classes of the class closure by using the following method.

public *String*[] *getClassNames*(*String agentName*)
Returns an array of Strings containing the names of all classes that the current agent uses.

[3] In rare cases, when all classes of the agent can be assumed to already exist at any destination agency, it is allowed to skip unit definition.

The class closure determined by Kalong contains all classes that are used by the agent, even common Java classes, as for example *java.lang.String*, etc. It is the task of the agent manager to implement a filter function for ubiquitous classes (compare p. 75) and not to add these to any code unit. In contrast, it is allowed to add classes to code units that were not part of the class closure. This might be important in rare cases where a class is used, but the class name is not part of the agent's Java byte code. Consider the following example:

```
1 public class TestAgent implements Serializable
2 {
3   public void run()
4   {
5     Class aClass = Class.forName( "OtherClass" );
6     SomeInterface object = aClass.newInstance();
7   }
8 }
```

In this example, class *OtherClass* is assumed to implement interface *SomeInterface*. The class is defined by using method *Class.forName*, which gets a *String* object as parameter containing the name of the class. A new instance of this class is created by using method *newInstance* and assigned to a variable of the super type. As a consequence, the byte code of class *TestAgent* does not contain the full class name for class *OtherClass*, except as *String* representation and this cannot be distinguished from other *String* objects without a semantic analysis of the byte code. To make class *OtherClass* able to migrate, it must be added to some code unit manually.

There is another method, by which the agent manager can obtain a list of all classes for which at least a single object exists in the serialized agent.

public *String*[] *getClassesInUse*(*String agentName*)
Returns an array of class names. Each class is used in the serialized agent.

The agent manager can also request the size of a class' byte code.

public int *getClassSize*(*String className*)
Returns the size of the given class.

7.4.3. Modifying Agent Context

The following methods are mostly for retrieving information about the agent context resp. the agent itself. They must be used within a transaction.

First, to request the address of an agent's home agency, the following method must be used.

public *URL*[] *getHomeAgency*(*String agentName*)
Return the addresses of the agent's home agency.

Data Items

The following methods are to access the data items of the external state and to define the object state.

public *String*[] *getDataItems*(*String agentName*)
Returns an array containing the names of all data items.

The returned array contains all data items, without regard to their current state. If the agent has no data items in the external state, the return value is the empty array. If only undefined data items are to be requested, then the following method can be used.

public *String*[] *getUndefinedDataItems*(*String agentName*)
Returns an array containing the names of all undefined data items.

If no undefined data items exist, the return value is the empty array. If only all defined data items are to be requested, then the following method can be used.

public *String*[] *getDefinedDataItems*(*String agentName*)
Returns an array containing the names of all defined data items.

To define and retrieve the value of a data item, the following two methods can be used.

public void *setData*(*String agentName*, *String name*, *Serializable object*)
throws *KalongException*
Set the data item with name *name* to the value given as *object*.

This method throws an exception, if the data item is currently undefined and, therefore, cannot be overwritten. To delete a data item permanently, it must be set to the **null** value.

public *Serializable getData*(*String agentName*, *String name*)**throws** *KalongException*
Returns the value of the data item with name *name*.

The method throws an exception, if the data item does not exist or is currently undefined. To check whether a data item is accessible, its state can be requested using this method:

public byte *getDataItemState*(*String agentName*, *String name*)**throws** *KalongException*
Returns the current state of the data item with name *name*.

The return value equals 0, if the data item is defined, and 1, if it is undefined. If the data item does not exist, an exception is thrown. In some cases it might be necessary to know the size of a serialized data item in order to decide, if it should migrate to the next destination or remain at the current one.

public int *getDataSize*(*String agentName*, *String name*)**throws** *KalongException*
Returns the size of the serialized data item with name *name*. Throws an exception, if the requested data item does not exist.

The last method must be used to define the object state of an agent.

public void *setObjectState*(*String agentName*, *Serializable state*)
Define the agent's object state.

This method must be used before a migration is started.

Code Units

The next methods are to retrieve information about code units.

public int[] *getUnits*(*String agentName*)**throws** *KalongException*
Returns an array with the identifiers of all units.

If no units were defined yet, the return value equals the empty array.

public *String*[] *getClassesInUnit*(*String agentName*, **int** *id*)**throws** *KalongException*
Returns an array with the names of all classes that are connected with the given unit identifier.

If the given unit identifier is invalid, an exception is thrown. The return value is never **null** or the empty array. The next method is used, when a specific class is to be downloaded.

public int[] *getUnitForClassName*(*String agentName*, *String className*)
Returns the identifiers of all units that contain the given class name.

If the given class name is not member of any unit, the empty array is returned.

Defining Code Server Agencies

To mark a unit to remain at the current agency after a migration, the addresses of the current agency must be added to the unit's code base.

public void *addCodeBases*(*String agentName*, **int** *id*, *URL*[] *url*)**throws** *KalongException*
Add the given URLs as new code bases for the given unit. Throws an exception, if the given identifier is invalid.

The new code bases must be appended to the existing list, because the order of the code base addresses must not be changed. With the following method, the current code bases can be requested.

public *URL*[] *getCodeBases*(*String agentName*, **int** *id*)**throws** *KalongException*
Returns all code bases of the given unit. Throws an exception, if the given identifier is invalid.

With the last method, agency addresses can be deleted from a unit's list of code bases.

public void *deleteCodeBases*(*String agentName*, **int** *id*, *URL*[] *url*)**throws** *KalongException*
Delete the given URLs from the code bases for the given unit. Throws an exception, if the given identifier is invalid.

Defining Mirror Agencies

Finally, the last three methods are to define and delete mirror agencies. It is important to understand that these methods have only a local effect and changing a remote agency to a mirror agency means more than only calling method *setMirrorAgency*. It is necessary to load all data items and code units from the current mirror or home agency before.

public *URL*[] *getMirrorAgency*(*String agentName*)
Returns the addresses of the current mirror agency, if defined. Otherwise return **null**.

public void *setMirrorAgency*(*String agentName*, *URL*[] *mirror*)**throws** *KalongException*
Define the current mirror agency. Throws an exception, if a mirror is defined already.

public void *deleteMirrorAgency*()
Delete the currently defined mirror agency.

Also the last method has only a local effect and it does not send a message to the current mirror agency to release its role.

7.4.4. Sending Messages to Receiver Agencies

The following methods are to conduct a transfer of messages to a single receiver agency. It is not allowed to open more than one connection to the same receiver agency during the same transaction. All transfer messages must be part of a transaction.

To start a transfer, the following method must be called.

public *Object startTransfer*(*String agentName*, *URL receiver*)**throws** *KalongException*
Open a connection to the receiver agency whose address is given as parameter. Return a handler object for this connection.

Note, that the address of the receiver agency must be given as a single URL. The agent manager is responsible to select the correct address from the list of all known URLs for the destination. The return value of this method is a so-called *handler* object which is used to identify this transfer. This specification does not define how this handler object is determined, but it must be unique for all transfers during the same transaction. There does not exist any method to stop a transfer explicitly, as network connections are closed by using the two methods *commit* and *rollback*.

All other methods process according to the following pattern.

1. The agent's context is accessed to obtain further information.
2. This information is sent as a SATP request message to the receiver agency which always has to answer with a reply message. The type of request depends on the method, compare Table 7.2 on page 136.
3. The reply message is analyzed and the result is stored in the agent's context if necessary.

In the following, we will focus on the semantic of each function. The structure of each message is defined in the following section.

There are a few constraints in sending messages, as not all messages must be sent as part of a single transfer. We describe these rules by using a finite state machine, whose graphical representation can be found in Figure 7.5. In the figure, states are named from the viewpoint of the receiver agency, so that for example a state is named *ADB Rcv*, which should express that the agent definition block (ADB) was received successfully. For the sender side, states should be renamed accordingly.

public boolean *ping*(*Object handler*, **byte**[] *data*)
Send the given byte sequence *data* to the receiver agency.

This method can be used to check the availability of the remote agency or to check connection quality. The receiver is supposed to send back the byte array *data* unchanged. The method returns **true** if the same byte sequence was received, otherwise **false**. Ping messages may be sent at any point during a transfer and may also be sent multiple times.

The following methods do not have a return value but throw an exception in the case of any error.

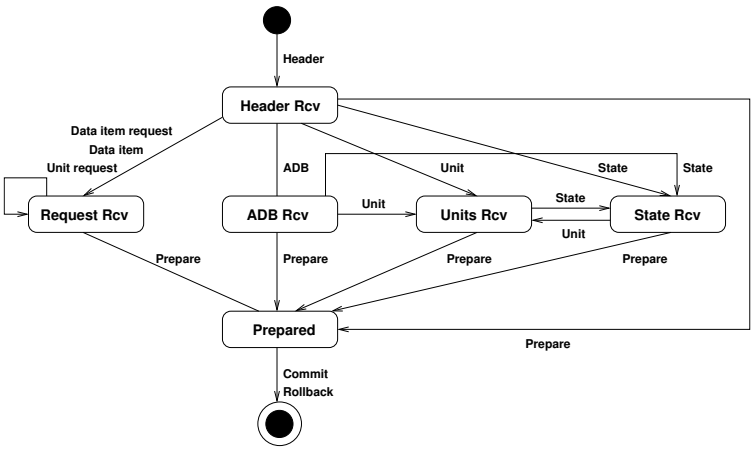


Figure 7.5.: State diagram for a SATP transfer. We omit to draw that at each state a Ping message can be sent/received, which does not change the state.

public void *sendHeader*(*Object handler*, **byte** *command*)**throws** *KalongException*

Send the header to the receiver agency. Throws an exception, if the receiver agency does not accept the header.

The header message contains information about the agent and its home agency. The receiver agency must answer, whether it accepts further messages for this agent or not. The sender should terminate the transfer, if the receiver did not accept the header. The receiver must reject any further messages except a ping message, if the header was not accepted.

The header message must be sent prior to all other messages during a transfer, except a ping message. It must not be sent more than once during the same transfer. A header message should be followed by any other message. Header messages can be sent between all agency types.

At the receiver agency the agent's name given in the header is used to select the agent context. If no agent with the given name exists, it must be decided, whether the transfer should be accepted or not, for example using the addresses of the home or sender agency. Kalong must ask the agent manager as part of this decision using method *verifyAgent* of interface *IAgentManager*. The reply code is set accordingly.

The *command* parameter contains the code of a process that should be executed *after* transfer is completed.

No operation (noop) (value 0) In most cases, this command is sent, which has no effect at the remote agency. In case of usual agent migration, it is *not* necessary

to specify to start the agent – even in this case, this command should be selected.

Release code server agency (value 1) Is sent to a code server agency in order to instruct it to release its role and delete all code units.

Release mirror agency (value 2) Is sent to a mirror agency in order to instruct it to release its role and delete all data items and code units. If there are *defined* data items left at the mirror agency, the header must be rejected.

Start code server (value 3) Using this command, it is possible to create a code server remotely, i.e. without migrating to it. All code units sent afterwards are immediately copied at the receiver agency.

Release mirror agency to code server (value 4) Is sent to a mirror agency in order to instruct it to release its role and become a code server agency. The effect is that all data items are deleted, but code units are still available at this agency. If there are *defined* data items left at the mirror agency, the header must be rejected.

public void *sendADB*(*Object handler*, **boolean** *classCache*)**throws** *KalongException*
Send the agent definition block (ADB) to the receiver agency.

This message is used to transmit information about code units and classes without the code itself. The parameter *classCache* defines whether the receiver should answer with information about class availability.

The header must have been sent before an ADB message. It is allowed to send a ADB message multiple times to the same agency, as it is possible that code bases have changed. After an ADB message, no request messages must be sent to the receiver agency.

public void *sendUnits*(*Object handler*, **int**[] *ids*)**throws** *KalongException*
Send code units with the given identifiers to the receiver agency.

This message is used to transmit a set of code units to the receiver agency. The receiver replies whether it accepts the transmission or not.

A header message must have been sent before a unit message. An agent definition block must be available at the receiver agency, before a unit can be accepted. To transmit many code units, they should be sent as one message. After a unit message, any unit request or data request message must not be sent, as it does not make sense to request units or data items from an agency to which units or data items were sent to right before.

public void *sendUnitRequest*(*Object handler*, **int**[] *ids*)**throws** *KalongException*

Send a request to download units with the given identifiers to the receiver agency.

This message is used to request units from a home, code server, or mirror agency. The receiver replies the requested units, unless one of the following reasons.

1. The receiver is not a home, code server, or mirror agency for this agent.
2. Any of the sent unit identifiers is invalid.

A header message must have been sent before this message. Messages of type ADB, unit, or state must not have been sent before or after this message. To request many data code units, their identifiers should be bundled into a single message of this type.

public void *sendState*(*Object handler*, *String*[] *names*)**throws** *KalongException*

Send the state to the receiver agency.

This message is used to transmit an agent's object state and optional some data items to the receiver agency. The names of the data items to transmit along the state are given as parameter *names*. This parameter might be **null**. The receiver answers, whether it accepts this message or not.

The effect of this method is that in fact the names of all data items are sent to the destination agency, but some of them without their value. The reason for this is that data items can only be distributed between two types of agencies: the current one and the home (resp. a mirror) agency. Even, if a data item shall remain at the home agency, the agent must have knowledge about this data item, at least to prevent creation of a new data item with the same name, which would rise conflicts when the agent migrates back to its home agency.

Therefore, the state contains the names of all data items. All data items given as parameter *names* are sent with state *defined* and their current value. The state is locally set to *undefined*. All other data items are sent with state *undefined* and without their value.

If the sender agency is not allowed to store data items (i.e. it is a remote or code server agency), all data items are transferred with their value to the destination agency without regard to the value of parameter *names*.

A header message must have been sent before a state message. An agent definition block must be available at the receiver agency before a state message can be received. It is not necessary that the ADB was sent during the same transfer. A state message must not be sent to an agency, where an agent is currently executed. After a state message, no unit request or data request message must be sent.

After a transaction in which a state message has been sent, some or even all agent related information have to be dropped at the sender agency. If the sender was a

remote agency and the agent has copied some code units, the agency becomes a code server. If the agent has not copied code units, all agent related information are deleted. If the sender was a home or mirror agency, they remain their role.

public void *sendDataUpload*(*Object handler*, *String*[] *names*)**throws** *KalongException*
Send data items whose names are given as parameter to the receiver agency.

This message is used to upload data items from the current agency to a home or mirror agency. It must not be sent to the home agency, if there exists a mirror agency. The receiver answers, whether the new data values are accepted. The receiver must reject a data item upload for the following reasons:

1. The receiver is not a home or mirror agency.

2. An uploaded data item is already *defined* at the receiver agency.

A header message must have been sent before this message type. Messages of type ADB, unit, or state must not have been sent before or will be sent after this message. To transmit many data items, they should be bundled into a single message.

Only data items that are *defined* at the current agency can be uploaded. At the receiver agency, these uploaded data items must be *undefined* before, and set to *defined* after receiving them. After a successful transmission, sent data items must be set to *undefined* locally.

public void *sendDataRequest*(*Object handler*, *String*[] *names*)**throws** *KalongException*
Send a request to download data items with the given names to the receiver agency.

This message is used to request data items from an agent's home or mirror agency. It must not be sent to the home agency, if there exists a mirror agency. The receiver must reply with the requested data items. The receiver must reject the message for any of the following reasons:

1. The receiver is not a home or mirror agency for this agent.

2. Any of the data items is *undefined* or does not exist at the receiver agency.

A header message must have been sent before this message. Messages of type ADB, unit, or state must not have been sent before or will be sent after this message. To request many data items, their names should be bundled into a single message.

7.5. Interface IAgentManager

This interface is used by Kalong to communicate to the agent manager. It is used during an in-migration of an agent to this agency to check whether reception of the agent is allowed. The agent manager could for example reject any migration that comes from a host supposed to be malicious. Therefore, it should use the information about the *lastAgency* given in the following method.

public boolean *verifyAgent*(*String agentName*, *URL*[] *homeAgency*, *URL*[] *lastAgency*)
Checks whether an agent with the given name and addresses is allowed to migrate in this agency.

After Kalong received an agent's state, the agent manager is asked to start agent execution. For this task, the following method is used:

public void *startAgent*(*String agentName*, *Serializable object*)
Start the given agent.

The parameter *object* contains the deserialized agent object. All classes not already available at the current agency must have been downloaded before.

7.6. Interface INetwork

The second interface defines methods to access the network adapter. The first method is used to get all addresses under which the network adapter is accessible.

public *URL*[] *getURLs*()
Return the URLs for all network protocols, or **null**, if no addresses are defined.

The next three methods are to handle network connections. The first method is used to open a communication channel to a remote agency.

public *Object openTransfer*(*URL receiver*)**throws** *KalongException*
Open a network connection to the given receiver agency and return an object to identify this transfer. Throws an exception, if the connection cannot be opened.

The second method is used to send a message to the destination agency. The return value contains the reply message, which must be processed now.

public byte[] *send*(*Object handle*, **byte**[] *data*)**throws** *KalongException*
Send the given byte sequence to the receiver, wait for a reply and return it. Throws an exception, if the method cannot be sent.

Finally, at the end of a transfer, the network connection must be closed again.

public void *closeTransfer*(*Object handle*)
Close a network connection.

7.7. Interface IServer

The last interface defines methods that must be used by the network component to access Kalong. Kalong must provide an implementation of this interface.

These methods are the counterparts to the ones described in the last section. For example, if method *send* of interface *INetwork* is called at the sender, method *receive* of this interface is called at the receiver.

public *Object openTransfer*()**throws** *KalongException*
Open a network connection, returns an object to identify this transfer.

The method returns **null**, if the connection cannot be opened.

public byte[] *receive*(*Object handle*, **byte**[] *data*)**throws** *KalongException*
Receive the given byte sequence, wait for a reply and return it.

public void *closeTransfer*(*Object handle*)
Close a network connection.

7.8. SATP Migration Protocol

This section defines the SATP migration protocol, version number 1.0.

7.8.1. Introduction

Each method of interface *IKalong* to transfer agent information uses one of the message types defined in this section. Although it might be clear from the names, which method uses which message, Table 7.2 gives an overview. A reply message named "Ok/Nok" stands for one that only contains information whether the receiver has accepted the request or not.

To describe the message format we use the Extended Backus-Naur Form as introduced in Chapter 5 on page 63. First of all, we define the following non-terminal symbols: a <Byte> represents a single byte with value range from 0 to 255. Symbol <Short> is used for numbers and is two byte long, and <Integer> is also used for numbers and is four byte long. To code <Short> and <Integer> symbols, we use the big-endian format, where the highest byte is stored first, i.e. at the lowest storage address. For example, a four-byte integer is stored in the following order:

Byte3	Byte2	Byte1	Byte0
0	1	2	3

Interface method	Request message	Reply message
ping	Ping	Ping
sendHeader	Header	Ok/Nok
sendADB	ADB	ADBReply
sendUnits	Unit	Ok/Nok
sendUnitRequest	Unit Request	Unit
sendState	State	Ok/Nok
sendDataUpload	Data Item	Data Item Key
sendDataRequest	Data Request	Data Item
prepare	Prepare	Ok/Nok
commit	Commit	none
rollback	Rollback	none

Table 7.2.: Mapping of interface methods to message types.

The first line shows the sequence of bytes in memory, the second line shows the byte offset.

In the following, it will be sometimes necessary to express a byte literal. In this case, we will use hexadecimal numbers, e.g. "0x15" to express the decimal number 21. In addition to the meta symbols introduced above, we define n{<A>}m as a repetition of symbol <A> between n and m times, where $0 \leq n \leq m$. Mostly, we will use this new meta symbol in the form where we have to define a repetition of exactly n times, so that we write n{<A>}n.

Sometimes, it is necessary to refer to a value of a <Byte>, <Short>, or <Integer> symbol. In this case, we will write $<A_n>$ to express that symbol <A> contains the value n, which we will use later on.

For example, the following <Message> comprises of $n + 5$ byte, where the number n is described by symbol <A>.

1. <Message> ::= $<A_n>$ + <B> + <C>
2. <A> ::= <Byte>
3. <B> ::= n{<Byte>}n
4. <C> ::= <Integer>

7.8.2. SATP Request and Reply Messages

The general operation of the SATP protocol is that the sender transmits a request to the receiver, which answers with a reply message. A request always has the following format:

1. <Request> ::= <RequestCode> + <Length$_n$> + <RequestParameter>

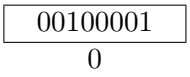
2. <RequestCode> ::= [<RcPing> | <RcHeader> | <RcUnit> | <RcUnitReq> | <RcData> | <RcDataReq> | <RcADB> | <RcState> | <RcPrepare> | <RcCommit> | <RcRollback>]

3. <Length> ::= 1{<Byte>}4

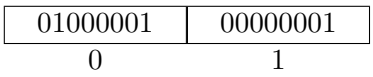
4. <RequestParameter> ::= n{Byte}n

A request starts with a command byte, which is followed by a sequence of one up to four bytes in which a number of bytes is coded as described below. The format of a <RequestParameter> depends on the message type and is defined later for each type.

To transmit a sequence of bytes, we use a byte-count oriented technique, where we send the number of the following bytes prior to the raw byte sequence. The advantage of this technique is that the receiver can read data from the network very fast, especially in Java. Every *String* or byte array is transmitted as such a byte sequence in SATP. Usually, coding a value of type **int** would result in a sequence of four bytes, but up to three of them might be wasted, because the number to be coded is less than 2^8 or 2^{16} or 2^{24}. To optimize these cases, we propose to code a byte length in the following way. We name the original number of bytes the *length*, and the resulting sequence of bytes which contains this number, the *code*. The idea is to use the two highest bits in the first byte of the code to contain the code length. A value of 0 means that the code is only one byte long (0 bytes following), a value of 3 means that the code is four bytes long (three bytes following). So, for example the following code stands for the number 33.



It can be seen that with a single byte it is only possible to code numbers from 0 to 63. The following two bytes are the code for number 257.



Therefore, in our approach the highest value for a length can be $2^{30} - 1$, which will be sufficient for all cases in SATP.

The format of a reply message depends on whether the request was successful or not.

5. <Reply> ::= [<ReplyOk> | <ReplyNok>]

6. <ReplyOk> ::= “0x6F” + <ReplyParameter>

7. <ReplyParameter> ::= <Length$_n$> + n{<Byte>}n

8. <ReplyNok> ::= “0x70” + <ErrorCode> + <ErrorText>

9. <ErrorCode> ::= <Short>

10. <ErrorText> ::= <String>

11. <String> ::= <Length$_m$> + m{<Byte>}m

If the request message was accepted, then a <ReplyOk> answer is sent. It comprises of a single byte which must have the value 0x6F and a byte sequence which contains the reply parameter. For example, if a unit request was sent, then this byte sequence will contain the units. If the request message was not accepted or the message format was not as accepted, then a <ReplyNok> answer is sent. The <ErrorCode> and the <ErrorText> contain a detailed error description.

7.8.3. Other SATP Messages

Ping

The ping message sends a sequence of bytes to the receiver agency which is supposed to reply with a <ReplyOk> message with the unchanged byte sequence as parameter.

12. <RcPing> ::= “0x76”

13. <PingParameter> ::= n{<Byte>}n

The parameter is a sequence of arbitrary bytes. The receiver must send back this byte sequence as <ReplyOk> message without any modification. The receiver answers with a <ReplyNok> message to indicate that it does not accept this ping message. The receiver is allowed to terminate a transfer, for example, if the sender tries to flood the receiver with ping messages.

Header

The header message contains information about the agent and its home agency. The receiver agency must answer, whether it accepts (<ReplyOk> without any parameter) further messages for this agent or not (<ReplyNok> with an error message).

14. <RcHeader> ::= “0x66”

15. <HeaderParameter> ::= <Vendor> + <Major> + <Minor> + <AgentName> + <HomeAgency> + <SenderAgency> + <Command>

16. <Vendor> ::= <String>

17. <Major> ::= <Byte>

18. <Minor> ::= <Byte>

19. <AgentName> ::= <String>

20. <HomeAgency> ::= <PackedURLArray>

21. <SenderAgency> ::= <PackedURLArray>

22. <Command> ::= [<Noop> | <ReleaseCodeServer> | <ReleaseMirror> | <ReleaseMirrorToCodeServer>]

23. <Noop> ::= "0x00"

24. <ReleaseCodeServer> ::= "0x01"

25. <ReleaseMirror> ::= "0x02"

26. <StartCodeServer> ::= "0x03"

27. <ReleaseMirrorToCodeServer> ::= "0x04"

The first part of the header is the SATP protocol version number. To assure that two agencies are able to exchange messages correctly, both should have the same version of the SATP protocol. The sender declares its version number as part of the header. The receiver should only accept a transfer if its version number equals the sender's one or is higher. We use a major–minor scheme to describe the version of the protocol. The minor number is incremented, when changes were made to the protocol that do not apply to the general message format, but to the semantics for example. The major number is incremented, if substantial modification were made with regard to the message format. Both values are coded in a byte each. For example, a version 0.1 is earlier than 0.11, which is earlier than 1.0.

The third part is the agent's name. The next two parts contain addresses of the agent's home agency and of the sender agency. A complete URL, for example *tracy://tatjana.cs.uni-jena.de:4567/fortknox/gold#abc*, consists of the following parts: protocol (tracy), host name (tatjana.cs.uni-jena.de), port number (4567), path to the resource (fortknox), file name of the resource (gold), and reference within the resource (abc). As Kalong does not define the structure of a URL, all parts of a valid URL must be transmitted. As we can assume that all addresses of the same agency have the same host name, path name, file name and reference, we only store these parts once. The first element of a <PackedURLArray> is the number of URLs following. We name this number n.

28. <PackedURLArray> ::= <NumberOfURLs$_n$> + <URLHostName> + <URLPath> + <URLFile> + <URLRef> + n{ <URLProtocol> + <URLPortNumber> }n

29. <NumberOfURLs> ::= <Short>

30. <URLHostName> ::= <String>

31. <URLPath> ::= <String>

32. <URLFile> ::= <String>

33. <URLRef> ::= <String>

34. <URLProtocol> ::= <String>

35. <URLPortNumber> ::= <Short>

ADB

This message is used to transmit information about code units and classes without the code itself. The receiver should answer with a <ReplyOk> and information about which classes are already available at the receiver's class cache. The receiver should not answer with a <ReplyNok>, except that the ADB message has had a wrong format.

36. <RcADB> ::= "0x67"

37. <ADBParameter> ::= <NumberOfUnits$_n$> + <CacheUsage> + <UnitDescriptions>

38. <NumberOfUnits> ::= <Short>

39. <CacheUsage> ::= [<UseCache> | <DoNotUseCache>]

40. <UseCache> ::= "0x00"

41. <DoNotUseCache> ::= "0x01"

42. <UnitDescriptions> ::= n{<UnitDescription>}n

The ADB contains the number of units and information about each code unit. The part <CacheUsage> defines whether the receiver agency should check all class descriptions against the local code cache and return information about class availability. In the last rule, n is the number of units (<NumberOfUnits>).

Each unit and each class of an agent has a unique identifier, which is assigned during agent creation at the agent's home agency and not changed later. Unit identifiers are used for example for downloading units, class identifiers are used in the reply to an ADB message to give information about class availability. Each unit description has the following format:

43. <UnitDescription> ::= <UnitId> + <ClassesDescription> + <CodeBases>

44. <UnitId> ::= <Short>

45. <ClassesDescription> ::= <NumberOfClasses$_n$> + n{<ClassDescription>}n

46. <NumberOfClasses> ::= <Short>

47. <ClassDescription> ::= <ClassId> + <ClassName> + (<Digest>)

48. <ClassId> ::= <Short>

49. <ClassName> ::= <String>

50. <Digest> ::= <Length$_d$> + d{<Byte>}d

51. <CodeBases> ::= <NumberOfCodeBases$_c$> + c{<CodeBase>}c

52. <NumberOfCodeBases> ::= <Short>

53. <CodeBase> ::= <NumberOfURLs$_u$> + u{<URL>}u

54. <URL> ::= <String>

Each unit description consists of the identifier, the number of classes in this unit, and a list of code bases for this unit. For each class, the class identifier, the class name, and a digest is sent (only if the cache is activated). To code the list of URLs in a code base, we do not use the packed form for URLs as described above, as a list of code bases will mostly contain different addresses where compression is not worthwhile.

If the receiver agency should check classes against its local cache, the class name and digest are used. If there is already a class with the given name for which the local digest equals the given digest, then it is assumed that the identical class is already available, otherwise, it is not.

The ADB reply object contains the identifier for each all classes that are already available at the destination agency.

55. <ADBReplyParameter> ::= <NumberOfClasses$_n$> + n{ <ClassId> }n

Unit

This message is used to transmit a set of code units to the receiver agency. The receiver replies whether it accepts the transmission or not. The receiver should only reject unit transmission, if the message cannot be parsed due to any format error.

56. <RcUnit> ::= "0x68"

57. <UnitsParameter> ::= <NumberOfUnits$_n$> + n{<Unit>}n

58. <Unit> ::= <UnitId> + <NumberOfClasses$_c$> + c{<Class>}c

59. <Class> ::= <ClassName> + <ClassCode>

60. <ClassCode> ::= <Length$_m$> + m{<Byte>}m

A unit message contains at least one or many units, where each unit is uniquely identified by a number. Each unit contains the code of at least one class. The set of classes that is transmitted for a specific unit depends on the reply of the ADB message. If the receiver already has the code for a specific class and transmits this information to the sender agency, then the sender should not send this class. As a consequence, the set of classes that is transmitted as a unit can be a subset of the classes that do belong to this unit.

State

This message is used to transmit an agent's state to the receiver agency. It consists of some URLs, the serialized agent, and optional some data items. The receiver answers, whether it accepts this message or not.

61. <RcState> ::= "0x6C"

62. <StateParameter> ::= <MirrorAgencies> + <DestinationAgency> +SerializedAgent + <DataItems> + <DataItemKey>

63. <MirrorAgencies> ::= <PackedURLArray>

64. <DestinationAgency> ::= <String>

65. <SerializedAgent> ::= <Length$_n$> + n{<Byte>}n

66. <DataItems> ::= <NumberOfDataItems$_m$> + m{<DataItem>}m

67. <DataItem> ::= <DataItemName> + <DataItemState> + (<SerializedDataItem>)

68. <NumberOfDataItems> ::= <Short>

69. <DataItemName> ::= <String>

70. <DataItemState> ::= [<Defined> | <Undefined>]

71. <Defined> ::= “0x10”

72. <Undefined> ::= “0x11”

73. <SerializedDataItem> ::= <Length$_l$> + l{<Byte>}l

The part <MirrorAgencies> is a list of addresses of the mirror agency if it exists. Otherwise, the number of URLs in the <PackedURLArray> equals 0. The second part contains the address of the agency to which the transfer is directed to. This address is important at the receiver agency, for example, if the agency consists of more than one place and the name of the destination place is part of the URL. The third part is the serialized agent, which is a sequence of bytes. The last part contains data items of the external state. Each data item has a name, a state, and optionally the serialized object as byte sequence. Symbol <DataItemKey> is defined later in rule 78.

Unit Request

This message is used to request units from a home, code server, or mirror agency. The receiver answers with a <ReplyOk> message and the requested units as parameter. The receiver must answer with <ReplyNok> in case of any error.

74. <RcUnitReq> ::= “0x69”

75. <UnitRequest> ::= <NumberOfUnits$_n$> + n{<UnitId>}n

Symbol <UnitId> was already defined in rule 44. The reply message has the format as defined in rule 57.

Data Item

This message is used to upload data items from the current agency to a home or mirror agency. The receiver answers with a <ReplyOk>, if the new data values are accepted, otherwise <ReplyNok>.

76. <RcData> ::= “0x6A”

77. <DataParameter> ::= <DataItems> + <DataItemKey>

78. <DataItemKey> ::= <Length$_n$> + n{<Byte>}n

Symbol <DataItems> was defined above. Symbol <DataItemKey> is a byte sequence that contains a key necessary to upload data items. This key protects data items to be overwritten accidentally at a home agency, when the agent already has defined a mirror agency. Additionally, it is necessary, as data uploading creates a security problem that we have to consider now. A malicious server might send forged data upload messages to a home or mirror agency and manipulate data items by this. The problem even occurs, if a malicious server requests a data item from a home server for example. As the state of this data item is set to *undefined* at the home server, the agent is not able to download the same data item later. Unfortunately, we cannot solve this problem completely, i.e. protect the home agency against any malicious access. However, we can provide a technique, so that the agent under all circumstances notices a malicious access and is able to react on this. The technique works in two steps:

1. Downloading and uploading data items has always the effect that states change. If a data item is downloaded, then its state is set to *undefined* at the home agency. If the agent later wants to download this data item again, it receives an error message.

2. If a malicious server loads a data item, modifies it, and then uploads it again, the state is set to *defined* at the home agency again. The agent would not notice the manipulation in this case. Therefore, we introduce a *data item key* that is necessary for data uploading. This key is created at the home agency when the agent is started and is carried by the agent as part of the state. In case of a data upload message the key must be sent to the home agency where it is compared to the key locally stored. If both are equal, the upload is successful. In the other case, the upload is rejected. In case of a successful upload, the reply message contains a new key that was computed by the home agency. A malicious server might have stolen the key that the agent carries. Using this key, the server is able to upload a data item. However, if the agent later wants to upload a data item by itself, the message is rejected as it knows an outdated key only.

Data Request

This message is used to request data items from an agent's home or mirror agency. The receiver answers with a <ReplyOk> message and the requested data items as parameter. The receiver must answer with a <ReplyNok> in case of any error.

79. <RcDataReq> ::= "0x6B"

80. <DataReqParameter> ::= <NumberOfDataItems$_n$> + n{<DataItemName>}n

Symbol <DataItemName> was define above in rule 69.

Prepare

The prepare message is sent to a receiver agency to check whether any error has been occurred during the current transfer. The receiver agency must answer with <ReplyOk> without parameter, if it accepts the whole transfer, and <ReplyNok> otherwise with an appropriate error message.

81. <RcPrepare> ::= "0x64"

Commit

The commit message is used to indicate the receiver agency to commit all changes done during the current transfer. The last stable state of the agent's context can be dropped.

82. <RcCommit> ::= "0x65"

The commit message is not supposed to send a valid reply message.

Rollback

The rollback message is used to indicate the receiver agency to release all changes made during the current transfer and to restore the last stable state.

83. <RcRollback> ::= "0x6D"

The rollback message is not supposed to send a valid reply message.

8. Implementation of Kalong

This chapter describes the implementation of the Kalong mobility model as the Kalong software component. It is the reference implementation of the Kalong specification.

8.1. Introduction

We start with a brief introduction to two very important aspects of Kalong as software component. First, Kalong in itself is not a complete mobile agent system but is designed to be an independent software component for agent migration to be used with (almost) any existing mobile agent server architecture. Second, Kalong is designed to work as a virtual machine for the task of agent migration. Therefore, it defines a minimal set of commands or functions, which are in sum sufficient to control the entire process of agent migration as defined in the Kalong mobility model.

8.1.1. Kalong as Software Component

Earlier in this thesis we already stated that about 70 mobile agent systems were developed by the community over 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.

Today's mobile agent systems are almost all stand-alone systems unable to communicate with each other, and sometimes not more than prototypes tailored to a specific research issue. The reason for this is the lack of any reference architecture for mobile agent systems as well as the absence of an open and extendable implementation. Therefore, each research group is compelled to develop its own prototype. Due to 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 system.

This is also the reason for 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, which defines several basic functions for mobile agents like communication, migration, etc., or just any serializable object?

- What level of communication is necessary: only a simple one between agents residing at the same agency or a complex one which also allows remote communication?
- What level of security is necessary? Is it sufficient to protect hosts against malicious agents, or is it necessary to protect agents against malicious servers too?
- 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, for example software components that can be installed in other mobile agent systems. Sometimes, even the general research idea cannot be adopted to another mobile agent systems due to differences in basic concepts, as described above. One practical downside is the number of different migration protocols currently existing. Except for two systems (Aglets and Grasshopper) which support the MASIF migration protocol proposed as OMG standard in 1998, it is virtually impossible to make two systems interoperable. Even for the two named systems, nobody has proofed yet whether they really can exchange agents. Although with the MASIF standard there exists a common migration protocol, almost no research group uses it, which might have to do with difficulties in implementing the complete standard because of its complexity and the lack of any independent software component offering MASIF as service.

In contrast to the big number of prototype implementations, few systems have already been developed as full-featured mobile agent systems that might be used in real-world applications too. These offer techniques for all important issues of agent programming, like migration, communication, security, management, etc. However, even these system are not willingly used, because of their complexity and size. They are built as monolithic systems, with a very high number of features and are not easy to configure and handle. Besides the impossibility to extend these systems by own implementations of research results, even mere usage might be prevented by functional overloading in certain application scenarios.

Some people see all this as a very important reason for harming the spread and acceptance of mobile agents.[1] If a system is not adaptable to real-world requirements, potential users are not willing to use this mobile agent system or mobile agents at all. To get hold of an industrial partner or sponsor for a real-world project becomes a forlorn hop in this situation.

To amend this situation, one of the most important challenges of our Tracy project is to develop a reference architecture for mobile agent systems in coordination with the Special Interest Group on Mobile Agents of the European AgentLink network.

[1] Compare for example `http://dsonline.computer.org/0208/f/kot.htm`.

Figure 8.1.: The Kalong software component and its interfaces.

This architecture makes intensive use of the concept of software components, which are all held together by a micro kernel [Buschmann et al., 1996]. It is leveraging off of previous work done by the Tracy team in designing the first Tracy architecture and benefits from experiences learned when porting it to mobile platforms and investigating feasibility to use Tracy within an electronic commerce application [Kowalczyk et al., 2002]. A first implementation of this reference architecture is currently under way as a diploma thesis and will be presented as Tracy2. A brief architecture description can be found in the appendix, see Section A.6 on page 279.

Such an open architecture has many advantages. As it has only a very small imperative core, as a basis for many software components to be added on, mobile agent systems become very modular. Each component offers specific services, as for example agent communication, agent migration, persistence, etc. This modular architecture makes it very easy to port a mobile agent system to other devices, as features no longer needed can be simply removed. If a feature is too heavy-weighted for a resource limited mobile device for example, it can replaced by another component with less services and of less size. Code reuse is also supported as the architecture guarantees that software components are usable at any mobile agent system built on this architecture.

One very important component in a *mobile* agent system is concerned with agent migration of course. Kalong provides such a component. It was developed independently of the Tracy2 architecture and bases only on very few assumptions about the environment. Therefore, it should be usable in (almost) all mobile agent software architectures and has already been successfully adapted to work with the Tracy2 architecture. It is planned to integrate Kalong into other systems, as for example, Grasshopper and Semoa in the near future.

Kalong defines four interfaces, compare Figure 8.1. On the left side of Kalong, there are two interfaces used to communicate to the agent manager. Interfaces on the right side are to communicate with the network adapter. Interface *IKalong* defines the functions of Kalong, whereas interface *IAgentManager* defines functions of the agent manager object used by Kalong. On the other side, class *INetwork* defines functions of the network adapter used by Kalong and class *INetworkServer* define the functions of Kalong that can be used by the network server component to be

called when messages are received from the network.

Kalong can be easily adapted to any mobile agent system as it is the result of reducing all requirements on a migration component to a common denominator. For example, Kalong only requires mobile agents to be a Java object of type *Serializable*, which is at least necessary in any mobile agent system to marshal an object's state.

Besides the pure functional advantages of Kalong, its flexible migration technique, and possibility to define fine-grained migration strategies, we see a major advantage of the concept of a migration component in the ability to make two different mobile agent systems interoperable. This usually has two distinct challenges. First, mobile agent systems must be able to communicate, i.e. they must understand the same migration protocol. Second, mobile agents of one system must be executable at the other one. The first challenge is taken on by Kalong. The second challenge must be resolved by the designer of the mobile agent system. First promising results have been reported by the Semoa research group at Fraunhofer Society, Darmstadt, Germany, who were able to adapt their system to run Tracy agents [Pinsdorf and Roth, 2002].

8.1.2. Kalong as Virtual Machine

The second aspect, we want to mention here, is the basic idea of Kalong as a *virtual machine* or *engine*[2] for agent migration.

Kalong provides a basic set of functions to describe the migration of a mobile agent. For example, it comprises of commands to define which units should be transferred, which data items shall be part of the state, and it contains commands to load code units or data items. Besides, it offers additional services, for example for transaction management, security, and persistence.

With this concept of a virtual machine, it now becomes obvious how to define a *migration strategy* in detail. In the last chapters we have always used this term to describe the effect of an agent migration, without going into details of defining it.

For example, a push strategy was defined to transmit all code units of an agent to the destination agency, together with the agent's state and all data items currently defined at the source agency. Using the commands defined in interface *IKalong* in the last chapter (p. 122), we can now give a first impression how a migration strategy may look like.

```
1 void sendAgent( String agentName, URL destination )
2 {
3   Object handle = null;
4   String[] allDataItems = kalong.getDataItems( agentName );
```

[2] We prefer the term *virtual machine*, although we run the risk to confuse the Kalong virtual machine with the Java virtual machine. In the following chapters, we will always refer to the Kalong virtual machine except otherwise mentioned.

```
5    int[] allUnits = kalong.getUnits( agentName );
6
7    kalong.startTransaction( agentName );
8    handle = kalong.openTransfer( agentName, destination );
9    kalong.sendHeader( handle, IKalong.NOOP );
10   kalong.sendADB( handle, true );
11   kalong.sendUnits( handle, allUnits );
12   kalong.sendState( handle, allDataItems );
13   kalong.prepare( agentName );
14   kalong.commit( agentName );
15 }
```

The migration commands of Kalong are very low-level, of course. To make programming of migration strategies easier, the mobile agent system can define new levels of abstraction on top of Kalong, for example to bundle often used sequences of commands into new commands, so that programming of migration strategies becomes more comfortable for the programmer. We will show the Tracy2 approach for this in a later section.

8.2. Using the Kalong Component

In this chapter we will give an introduction to use Kalong as a software component and to program migration strategies. This chapter does not contain the full documentation of all classes of the Kalong component. Some deeper introduction into the main classes and the overall design of Kalong can be found in the following section. The full documentation of all classes can be found on the enclosed CD-ROM, which also contains the complete source code for Kalong and all other components and classes necessary to execute the examples presented in this chapter.[3]

8.2.1. Starting and Configuring Kalong

The main class of the Kalong software component is class *Kalong* in package *org.taf.kalong*. It has the following two constructors:

public *Kalong*()
Creates a new Kalong component.

[3]The documentation of Kalong consists of a set of HTML files generated by the JavaDoc tool. The main file is ./docs/index.html, which also gives an overview of the content of the CD-ROM. If you read this thesis electronically (for example using the Acrobat Reader), you can use Acrobat's WebLink function to open the documentation by clicking on the file name. Additionally, we underlay all class names with a link to the class documentation. Not all PDF viewer programs are able to handle links correctly.

public *Kalong*(*INetwork network*)
Creates a new Kalong component that uses the given *network* component.

To embed Kalong in an existing mobile agent system, it must be connected to the agent manager and the network (as long as not done with the constructor already).

public void *registerNetwork*(*INetwork network*)
Register a network component with this instance of Kalong.

public void *registerListener*(*IAgentManager listener*)
Register an agent manager with this instance of Kalong.

At last, the network component must be able to inform Kalong about incoming messages. For this task, Kalong offers the interface *INetworkServer*, for which Kalong already provides an implementation. This implementation can be requested using the following method:

public *INetworkServer getNetworkServerInterface*()
Returns an implementation of interface *INetworkServer*.

The following method is to check, whether an agent context already exists.

public boolean *existsAgentContext*(*String agentName*)
Returns **true**, if an agent context with the given name already exists.

The last method is used to delete an agent context.

public void *deleteAgentContext*(*String agentName*)
Delete an agent context locally.

To delete an agent context has the consequence that no information about data items, object state, and code units do exist anymore at the current agency. Calling this method does not have the consequence that an agent currently roaming the agent system is killed. However, this agent cannot use this agency for downloading data items or code units anymore, which might cause an unexpected behavior or might even crash the agent. The agent manager must make sure that probably existing code servers must be informed to release, before the agent context is deleted.

In the following example, we show how to start and configure a Kalong instance.

```
1 package test;
2
3 import org.taf.kalong.Kalong;
4 import org.taf.kalong.IKalong;
5 import org.taf.network.Network;
6 import org.taf.network.ProtocolEngine;
7 import org.taf.network.tcp.TCPEngine;
```

```
8
9  public class StartKalong
10 {
11   public static void main( String[] args )
12   {
13     Kalong kalong = null;
14     IKalong iKalong = null;
15     Network network = null;
16     ProtocolEngine tcpProtocol = null;
17
18     kalong = new Kalong();
```

An example for a network component is also part of the CD-ROM. The main class of this component is *Network*, which works as a manager for several network transmission protocols. Each transmission protocol must be implemented by extending class *ProtocolEngine*.

```
19     network = new Network();
20     tcpProtocol = new TCPEngine();
21     network.registerProtocolEngine( tcpProtocol );
22     tcpProtocol.startServer( 5555 );
```

Each protocol engine defines a protocol name that can be used to define URLs. For example, class *TCPEngine* defines the protocol with name "tcp". In line 22 a new thread is started that will accept incoming messages on port 5555.

Now, Kalong must be connected to the network component. As the network component is an independent software component, it does not implement the interfaces of Kalong, but provides interfaces on its own. To allow communication between these two interfaces, we implement adapter classes. For example, from the view of Kalong we create an adapter that implements the Kalong interface *INetwork* and accesses the network component transparently. For the other communication direction, we need another adapter class that implements an interface of the network components and directs all method invocations to Kalong. We omit to print the source code of both classes here, the source code can be found on the CD-ROM.[4]

```
23     NetworkAdapter nAdapter = new NetworkAdapter( network );
24     KalongAdapter kAdapter = new KalongAdapter( kalong.
           getNetworkServerInterface());
25     kalong.registerNetwork( nAdapter );
26     network.registerListener( kAdapter );
```

[4]See *KalongAdapter* and *NetworkAdapter* in package *org.taf.mdl*.

```
27      kalong.registerListener( new KalongListener() );
28    }
29  }
```

In line 27, we register a listener object with Kalong, which will be informed in case of received agents. This listener must implement interface *IAgentManager* and we will show an example for this listener later in Section 8.2.3.

The last method of class *Kalong* is used by the agent manager to request an implementation of interface *IKalong*, which contains all functions to access an agent context and program migration strategies.

public *IKalong getKalongInterface*(*String agent*)**throws** *KalongException*
Return an implementation of interface *IKalong*.

As this interface has some minor differences as compared to the one presented in the specification (p. 122), we will present its definition in the following section.

8.2.2. Interface IKalong

The difference of this interface as compared to the one defined in the last chapter is that it is personalized to a single agent. Whereas in the last chapter, almost all functions required a parameter *agentName* of type *String*, we now give the agent's name only once when obtaining the interface.

```
1  public interface IKalong
2  {
3      public static final byte NOOP = 0x00;
4      public static final byte REL_CODESERVER = 0x01;
5      public static final byte REL_MIRROR = 0x02;
6      public static final byte START_CODESERVER = 0x03;
7      public static final byte REL_MIRROR_TO_CODESERVER = 0x04;
8
9      public static final byte DATA_DEF = 0x00;
10     public static final byte DATA_UNDEF = 0x01;
11
12     // transaction management
13     public void startTransaction() throws KalongException;
14     public boolean prepare();
15     public void commit();
16     public void rollback();
17
18     // registering agents
19     public URL[] getHomeAgency() throws KalongException;
20     public URL[] getURLs() throws KalongException;
21     public void registerAgent( Serializable agentObject ) throws KalongException;
22     public int defineUnit( String[] classNames ) throws KalongException;
23     public String[] getClassNames() throws KalongException;
24     public String[] getClassesInUse() throws KalongException;
25     public int getClassSize( String className ) throws KalongException;
```

```
26      public URL[] getLastAgency() throws KalongException;
27
28      // data items
29      public String[] getDataItems() throws KalongException;
30      public String[] getDefinedDataItems() throws KalongException;
31      public String[] getUndefinedDataItems() throws KalongException;
32      public byte getDataItemState( String dataItem ) throws KalongException;
33      public void setDataItem( String dataItem, Serializable dataObject ) throws KalongException;
34      public Serializable getDataItem( String dataItem ) throws KalongException;
35      public int getDataSize( String dataItem ) throws KalongException;
36      public void setObjectState( Serializable agentObject ) throws KalongException;
37
38      // code units and code servers
39      public int[] getUnits() throws KalongException;
40      public String[] getClassesInUnit( int id ) throws KalongException;
41      public int[] getUnitForClassName( String className ) throws KalongException;
42      public void copyUnit( int id ) throws KalongException;
43      public URL[] getCodeBases( int id ) throws KalongException;
44      public void addCodeBases( int id, URL[] url ) throws KalongException;
45      public void deleteCodeBase( int id, URL[] url ) throws KalongException;
46      public byte[] getByteCode( String className ) throws KalongException;
47
48      // mirrors
49      public URL[] getMirrorAgency() throws KalongException;
50      public void setMirrorAgency( URL[] mirror ) throws KalongException;
51      public void deleteMirrorAgency() throws KalongException;
52
53      // transfers
54      public Object startTransfer( URL destination ) throws KalongException;
55      public boolean ping( Object handle, byte[] data ) throws KalongException;
56      public void sendHeader( Object handle, byte command ) throws KalongException;
57      public void sendADB( Object handle, boolean classCache ) throws KalongException;
58      public void sendUnits( Object handle, int[] unitIds ) throws KalongException;
59      public void sendUnitRequest( Object handle, int[] unitIds ) throws KalongException;
60      public void sendState( Object handle, String[] dataItems ) throws KalongException;
61      public void sendDataUpload( Object handle, String[] dataItems ) throws KalongException;
62      public void sendDataRequest( Object handle, String[] dataItems ) throws KalongException;
63  }
```

In lines 3–7, all valid header commands are defined, which can be used in method *sendHeader* to release a code server or mirror agency. In lines 9–10, all valid states for data items are defined. A description of each method can be found in the documentation on the CD-ROM.

8.2.3. Interface IAgentManager

During the process of receiving an agent from the network, Kalong communicates to the agent manager using interface *IAgentManager*.

```
1   public interface IAgentManager
2   {
```

```
3     public Object receivedInMigration( Object handle, String agentName, URL[]
          homeAgency, URL[] lastAgency );
4     public ProtectionDomain getProtectionDomain( Object handle );
5     public void startAgent( Object handle, Serializable agent, URL destination );
6     // ...
7 }
```

The first method *receivedInMigration* is called after a SATP header was received. The agent manager has now the chance to verify, if this transfer should be accepted. For this decision, it can use the given parameters, the agent's name, the agent's home agency, and the addresses of the sender agency. If the agent manager returns a **null** value, the sender is informed about header rejection. In the other case, the return value is an object, by which the agent manager can identify this transfer in future.

The second method *getProtectionDomain* is called by Kalong before the received agent is deserialized. The return value must be an object of type *ProtectionDomain*, which is a Java class from package *java.security*. A protection domain is a grouping of a code source and permissions granted to all code from this code source. Protection domains are used to specify the permissions of an agent on the current agency. It is given to the class loader that will assign this protection domain to all classes of the agent.

For example, the following method creates a protection domain, which grants permission to read all files in the user's home directory.

```
1     public ProtectionDomain getProtectionDomain(Object handle)
2     {
3       // ...
4       PermissionCollection coll = new PermissionCollection();
5       coll.add( new FilePermission("${user.home}/-", "read") );
6       CodeSource cs = new CodeSource( handle.homeAgency[0], null );
7       ProtectionDomain pd = new ProtectionDomain( cs, coll );
8       return pd;
9     }
```

Finally, the third method *startAgent* is called by Kalong after the agent was initialized. The second parameter *agent* contains a reference to the deserialized agent object and the third parameter contains the URL of the migration destination. The agent manager might need this URL to dispatch the incoming agent to a specific place, whose name is stored in the URL.

Later, in Chapter 9 we show other methods of this interface that can be used to sign and encrypt messages.

8.2.4. Examples to Use Interface IKalong

We will now show how to use the Kalong component in some typical use-cases.

Registering an Agent

Before an agent can migrate or use any other function of Kalong, it must be registered with the component. This is done using method *registerAgent*. The agent object must have been initialized before.

```
1  package test;
2
3  // ...
4
5  public class TestKalong
6  {
7    public static void main( String[] args )
8    {
9      Kalong kalong = new Kalong();
10     IKalong iKalong = null;
11     Runnable agent = new Agent();
12     agent.run();
13
14     // connection Kalong and the other components
15
16     try
17     {
18       iKalong = kalong.getKalongInterface( ``Scaramanga'' );
19       iKalong.startTransaction();
20       iKalong.registerAgent( agent );
21
22       String[] allClasses = ikalong.getClassNames();
23       String[] filterClasses = new String[] { ``java.*'', ``javax.*'', ``org.xml.*'' }
24       String[] agentClasses = ArrayUtils.filter( allClasses, filterClasses );
25       iKalong.defineUnit( agentClasses );
26
27       iKalong.commit();
28     } catch( KalongException e ) {
29       e.printStackTrace();
30       iKalong.rollback();
31     }
32   }
33 }
34
35 class Agent implements Serializable, Runnable
36 {
```

```
37    private Integer value = new Integer(100);
38
39    void run()
40    {
41      // ...
42    }
43  }
```

As can be seen, an agent can be any object of a class that implements at least interface *Serializable*. In the example, class *Agent* also implements interface *Runnable* and, therefore, must provide a method with name *run*. In line 18, the Kalong interface for an agent with name *Scaramanga* is requested and in line 20 the initialized agent object is registered with Kalong. After registering, the given agent object is accessible under the name *Scaramanga*. Please note that for sake of readability we chose short and human-readable agent names in all examples. A real implementation must guarantee that agent names are unique in the whole agent system.

The effect of registering is that Kalong has read the agent's class file and has determined the class closure of the agent's main class. This list of class names can be requested using method *getClassNames* of the Kalong interface. In the example above, this class list would comprise of classes *test.Agent*, *java.lang.Object*, *java.lang.Integer*, *java.io.Serializable*, and *java.lang.Runnable*.

Now, we must define the agent's code units. As already said, the user of Kalong is responsible to filter classes that are ubiquitous and, therefore, need not to migrate to other agencies. The utilities package *org.taf.util* contains a class *ArrayUtils* that provides a method for filtering class names.

String[] *ArrayUtils.filter*(*String*[] *source*, *String*[] *pattern*)
Returns the source parameter without Strings that match any of the given pattern.

To describe *pattern*, regular expression can be used. In lines 22–24, we use this method to filter out all base Java classes from the list of all agent's classes. In line 25 a single unit is defined that contains all agent's classes.

Accessing Data Items

Kalong provides functions to store data items in the agent's context. To store a data item, method *setDataItem* must be used. After a new data item was stored, it has the state *DATA_DEF* which is a constant defined in interface *IKalong*. To retrieve a data item, method *getDataItem* must be used. A data item can be any object that is serializable.

```
1       try
```

```
2       {
3         iKalong.setDataItem( ‘‘firstDataItem’’, new Integer( 100 ) );
4         assert( iKalong.getDataItemState( ‘‘firstDataItem’’ ) == IKalong.DATA_DEF
                ));
5
6         Integer anInteger = (Integer)iKalong.getDataItem( ‘‘firstDataItem’’ );
7
8         iKalong.getDataItem( ‘‘secondDataItem’’ );
9
10        iKalong.commit();
11      }
12      catch( KalongException e )
13      {
14        e.printStackTrace();
15        iKalong.rollback();
16      }
```

In line 8, a data item is requested that does not exist. An exception is thrown in this case. The Kalong interface provides other methods to retrieve an array of all data items names or to determine the size of a single serialized data item. The latter can be important to decide, which data items shall migrate or not.

Simple Migration

We now present the implementation of a simple migration. It is in fact the one that we have referred to as *push-all-to-next*, which transmits all agent's code and all its data to the next destination.

For the following, we assume that the agent was registered and code units were already defined. The agent might also have stored some data items in its context. Variable *iKalong* contains a reference to the agent's Kalong interface.

```
1       try
2       {
3         URL destination = new URL( ‘‘tcp://tatjana.cs.uni-jena.de:5555’’ );
4         int[] unitIds = iKalong.getUnits();
5         String[] dataItems = iKalong.getDataItems();
```

Variable *destination* contains the address of the destination agency. The array of integer values with name *unitIds* contains all identifiers of the agent's code units and the array of Strings with name *dataItems* contains the names of all data items the agent owns.

```
6         iKalong.setObjectState( agent );
7         iKalong.startTransaction();
```

In line 6 the agent object state is stored in the agent's context and the next statement starts the transaction.

```
8         Object handle = iKalong.startTransfer( destination );
9         iKalong.sendHeader( handle, IKalong.NOOP );
10        iKalong.sendADB( handle, true ); // true means to use cache
11        iKalong.sendUnits( handle, unitIds );
12        iKalong.sendState( handle, dataItems );
13      }
14      catch( Exception e )
15      {
16        e.printStackTrace();
17      }
18      finally
19      {
20        if( iKalong.prepare() )
21        {
22          iKalong.commit();
23        } else
24        {
25          iKalong.rollback();
26        }
27      }
```

This migration strategy consists of a single transfer. The connection to the destination agency is opened using method *startTransfer*. The return value is an object that is used to identify this transfer when sending further messages. The first message that is sent now must be a SATP header, which transmits the agent's name and some other important information about the agent to the destination agency. The second parameter of this method is the *header command* which specifies the process that will be executed at the end of this transfer at the destination agency. In this case, *NOOP* stands for no process.

In the following, the agent definition block is sent. The second parameter of method *sendADB* defines whether the remote agency should check for already available classes and reply this information. Then, all units and the state together with all data items is sent. Finally, the whole transaction is prepared and then committed or rolled back.

Loading Data Items From the Home Agency

The following example shows how to download a single data item from the agent's home agency.

```
1   try
2   {
3     URL homeAgency = iKalong.getHomeAgency()[0];
4     String[] dataItemsToLoad = new String[] { ''secondDataItem'' };
5
6     iKalong.startTransaction();
7
8     Object handle = iKalong.startTransfer( homeAgency );
9     iKalong.sendHeader( handle, IKalong.NOOP );
10    iKalong.sendDataRequest( handle, dataItemsToLoad );
11
12    Integer second = (Integer)iKalong.getDataItem( ''secondDataItem'' );
13  }
14  catch( Exception e )
15  {
16    e.printStackTrace();
17  }
18  finally
19  {
20    if( iKalong.prepare() )
21    {
22      iKalong.commit();
23    } else
24    {
25      iKalong.rollback();
26    }
27  }
```

The addresses of the agent's home agency can be obtained by calling method *getHomeAgency*. If the home agency is accessible by different network protocols, the returned array contains more than a single URL. As a simplification, we chose the first address by default. The name of the data item to load is defined in line 4 and the request to load a data item is sent in line 10. Immediately after this method terminated, the data item is available using method *getDataItem*.

The effect of loading a data item is that its state is set to *DATA_UNDEF* at the agent's home agency and to *DATA_DEF* at the current agency. If the same data item is loaded once again from the home agency, the transfer would be not successful and an exception would be thrown locally.

Uploading a Data Item and Migrate to the Next Destination

The next example shows how to handle more than a single transfer during a single transaction. The task is to migrate to agency *tatjana.cs.uni-jena.de* but not carry data item with name *firstDataItem*. Instead, it is uploaded to the agent's home agency before.

```
1   try
2   {
3     URL destination = new URL( ‘‘tcp://tatjana.cs.uni-jena.de:5555’’ );
4     URL homeAgency = iKalong.getHomeAgency()[0];
5     String[] dataItemsToUpload = new String[] { ‘‘firstDataItem’’ };
6
7     iKalong.startTransaction();
8
9     Object handleHome = iKalong.startTransfer( homeAgency );
10    iKalong.sendHeader( handleHome, IKalong.NOOP );
11    iKalong.sendDataUpload( handleHome, dataItemsToUpload );
12
13    Object handleDest = iKalong.startTransfer( destination );
14    iKalong.sendHeader( handleDest, IKalong.NOOP );
15    iKalong.sendADB( handleDest, false ); // true means to use cache
16    iKalong.sendUnits( handleDest, null );
17    iKalong.sendState( handleDest, null );
18  }
19  catch( Exception e )
20  {
21    e.printStackTrace();
22  }
23  finally
24  {
25    if( iKalong.prepare() )
26    {
27      iKalong.commit();
28    } else
29    {
30      iKalong.rollback();
31    }
32  }
```

The type of migration that is used to transmit the agent to the destination agency is comparable to a *pull* strategy, as no code units are sent in line 16.

Defining Code Servers and a Mirror Agency

The next example shows how to make the current agency to become a code server agency. The goal is that after the next migration all code units shall remain at the current agency and can be downloaded from this agency later.

```
1       try
2       {
3         int[] unitIds = iKalong.getUnits();
4
5         for( int i=0; i<unitIds.length; i++ )
6         {
7           iKalong.copyUnit( unitIds[i] );
8         }
9       }
10      catch( Exception e )
11      {
12        e.printStackTrace();
13      }
```

The important statement is in line 7, where the unit with the given identifier is marked not to be deleted after the next migration. The effect of this method is that the addresses of the current agency are added to the list of code bases of the unit.

To define a mirror agency is only a little bit more complicated. A mirror agency must hold all data items and all code units per definition. Therefore, before a mirror agency can be activated, the agent manager has to load all missing data items and code units. Kalong does not provide a method for this.

We assume a situation, where currently no mirror agency is defined. If this is not true, then it would be necessary to release the current mirror agency first.

```
1       try
2       {
3         URL homeAgency = iKalong.getHomeAgency()[0];
4         String[] dataItemsToLoad = iKalong.getUndefinedDataItems();
5         int[] unitsToLoad = iKalong.getUndefinedUnits();
6
7         iKalong.startTransaction();
8         Object handleHome = iKalong.startTransfer( homeAgency );
9         iKalong.sendHeader( handleHome, IKalong.NOOP );
10        iKalong.sendDataRequest( handleHome, dataItemsToLoad );
11        iKalong.sendUnitRequest( handleHome, unitsToLoad );
12      }
13      catch( Exception e )
```

```
14      {
15        e.printStackTrace();
16      }
17      finally
18      {
19        if( iKalong.prepare() )
20        {
21          iKalong.commit();
22        } else
23        {
24          iKalong.rollback();
25        }
26      }
27
28    iKalong.setMirrorAgency();
```

In line 28 the current agency is defined to be a mirror agency from now on.

Release a Code Server or Mirror Agency

Finally, we show an example how to release a remote code server or mirror agency by sending a header command. To release code servers is in important task to free resources at the other agencies that currently hold a copy of the agent's code. Releasing mirror agencies is necessary, before defining a new mirror as mentioned above.

We only show an example to release a code server.

```
1       try
2       {
3         URL codeServerToRelease = new URL(“tcp://tatjana.cs.uni-jena.de:5555”);
4
5         iKalong.startTransaction();
6
7         Object handle = iKalong.startTransfer( codeServerToRelease );
8         iKalong.sendHeader( handle, IKalong.REL_CODESERVER );
9
10        int[] allUnits = iKalong.getUnits();
11        URL[] deleteCodeBase = new URL[] { codeServerToRelease };
12        for( int i=0; i<allUnits.length; i++ )
13        {
14          iKalong.deleteCodeBase( allUnits[i], deleteCodeBase );
15        }
16      }
17      catch( Exception e )
```

```
18      {
19        e.printStackTrace();
20      }
21      finally
22      {
23        if( iKalong.prepare() )
24        {
25          iKalong.commit();
26        } else
27        {
28          iKalong.rollback();
29        }
30      }
```

To simplify the code, we assume to know the address of the code server that should be released. In fact, this address must be obtained from the agent's context using for example method *getCodeBases* which returns all known code bases for a given unit. To release a code server is defined as header command in line 8. No more messages must be sent to the destination in this case. After the transfer the code server must be deleted from the list of code bases of all units. This is done in lines 10–15.

If a mirror agency is to release, then the header command must be changed to *REL_MIRROR* and finally method *deleteMirrorAgency* must be called.

8.3. Implementation Details

In this section we will give a brief overview of the implementation of the Kalong software component. The implementation consists of 32 Java classes with in sum about 7000 lines of code (without documentation). Kalong uses some other utility classes of the Tracy project, which have in sum about 1600 lines of code.

These classes are organized in the following packages:

org.taf.kalong Contains the main classes of Kalong, such as for example, class *Kalong* and the main interface *IKalong*.

org.taf.kalong.util Contains Kalong specific utility classes, for example the Kalong class loader class *KalongClassLoader*.

org.taf.kalong.util.classcache Contains all classes that belong to the Kalong class cache.

org.taf.util Contains some imperative classes from the Tracy project, as for example class *ThreadPool* and class *ByteBuffer*, which provides methods to code primitive Java data types into a flat byte array.

To use Kalong, a network component is required that implements the interfaces as described in the Kalong specification. Our implementation of a network component is organized in the following packages:

org.taf.network Contains the main class *Network* and the implementation of an abstract class *ProtocolEngine* which provides the skeleton for the implementation of transmission protocol engines.

org.taf.network.tcp Contains the implementation of a protocol engine that uses the TCP transmission protocol.

org.taf.network.ssl3 Contains the implementation of a protocol engine that uses the SSL transmission protocol.

The implementation of the network component has about 1500 lines of code (without documentation).

We will now describe some very important classes that are mandatory to understand the internal processes of Kalong. For a documentation of the other classes, especially the network component, we refer to the documentation in HTML on the enclosed CD-ROM.

8.3.1. Important Classes of the Kalong Component

The design of Kalong was done along the following principles:

- For each agent residing at the current agency, there exists an object of class *Context*, which holds all information about data items and code units of the agent. Context objects are managed by an object of class *ContextManager* which is responsible to verify that the same context is not used within more than a single transaction in parallel.

- A single transaction is completely handled by a single so-called *session* object. A session object is created for every transaction. It works as a *mediator* that uses several other objects (its colleagues) for subtasks. Compare Gamma et al. [1994] for a description of the *mediator design pattern*.

- The colleagues, i.e. class *ContextManager*, class *Codec*, and class *SATP* are singleton.

Figure 8.2 shows a class diagram using UML notation that contains the main classes of Kalong.

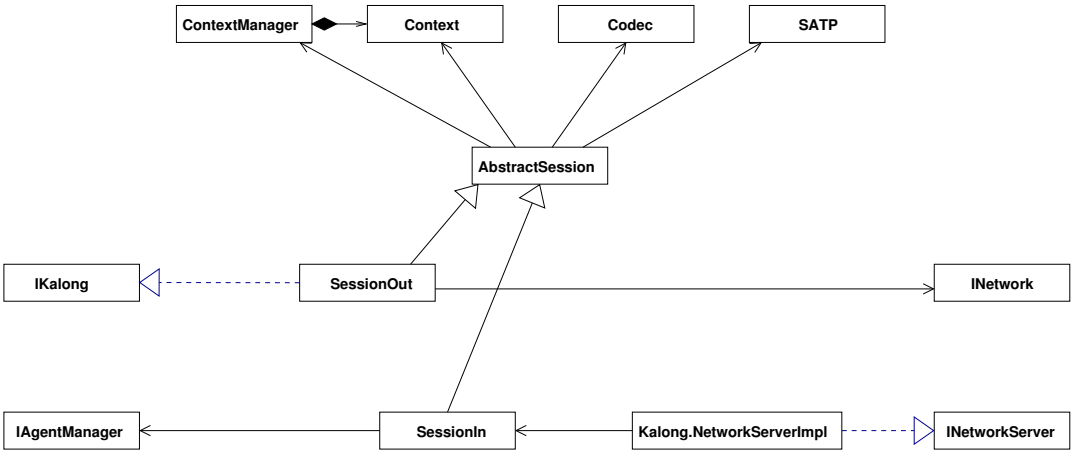


Figure 8.2.: Overview of the main classes of the Kalong software component.

AbstractSession

Class *AbstractSession* is the base class for the two classes *SessionOut* and *SessionIn*. Both session classes handle transfers, class *SessionOut* handles out-migrations and class *SessionIn* handles in-migrations.

This abstract class defines common behavior of both session types. Especially, it defines an inner class *AbstractSession.States*, which is responsible to control a transaction according to the rules of the SATP migration protocol (compare Figure 7.5).

SessionOut

This class implements interface *IKalong* and provides access to a single agent's context. It is also responsible to handle all tasks related to transaction management and transfer of agent information to remote agencies.

An instance of this class is returned by method *getKalongInterface* of class *Kalong*. It uses class *ContextManager* to get a reference to the agent's context object and delegates subtasks to other classes as described below.

SessionIn

This class is responsible to handle incoming SATP messages. It is created by class *Kalong.NetworkServerImpl*, which is the default implementation of *INetworkServer*, when a remote agency opens a connection to this agency. The object is deleted, after the transaction has terminated.

It delegates subtasks to other classes as described below. During the reception of an agent, class *IAgentManager* is called, as described in Section 8.2.3.

ContextManager

The *ContextManager* is responsible for managing agent's contexts. It provides two methods *lock* and *unlock*, which a session object uses to get access to a single context object. This class is also responsible for context persistence. It exists only a single instance of this class.

Context

Class *Context* is the container for all agent related information that must be known in Kalong. There exists a single context object for each agent currently residing at this agency. Context objects are created and deleted by the context manager.

Codec

The *Codec* class implements the message structure as described in Section 7.8. It provides for each SATP message type two methods, which are called by a session object to code or decode messages. It exists only a single instance of this class.

SATP

Finally, class *SATP* provides functions to create request messages that consist of a request type and a request parameter. On the other hand, it provides methods to decode and process reply message. For example, method *decodeReply* returns the reply parameter in case of a <ReplyOk> message and throws an exception of type *KalongException* in case of a <ReplyNok> message. It exists only a single instance of this class.

8.3.2. Sequence Diagram for Sending a Header Message

To give an impression how Kalong works internally, we will now present a sequence diagram for the process of opening a network connection and sending a SATP header message, compare Figure 8.3. The sequence diagram shows the communication between the objects described above. We assume that the agent is registered with the context manager already. For the following, the *client* is the instance that uses Kalong for agent migration, for example the agent manager.

First, the client calls *Kalong* to return a reference to a *IKalong* interface for the agent whose name is given as parameter. A new instance of class *SessionOut* is created. The client starts the transaction by calling method *startTransaction*, which

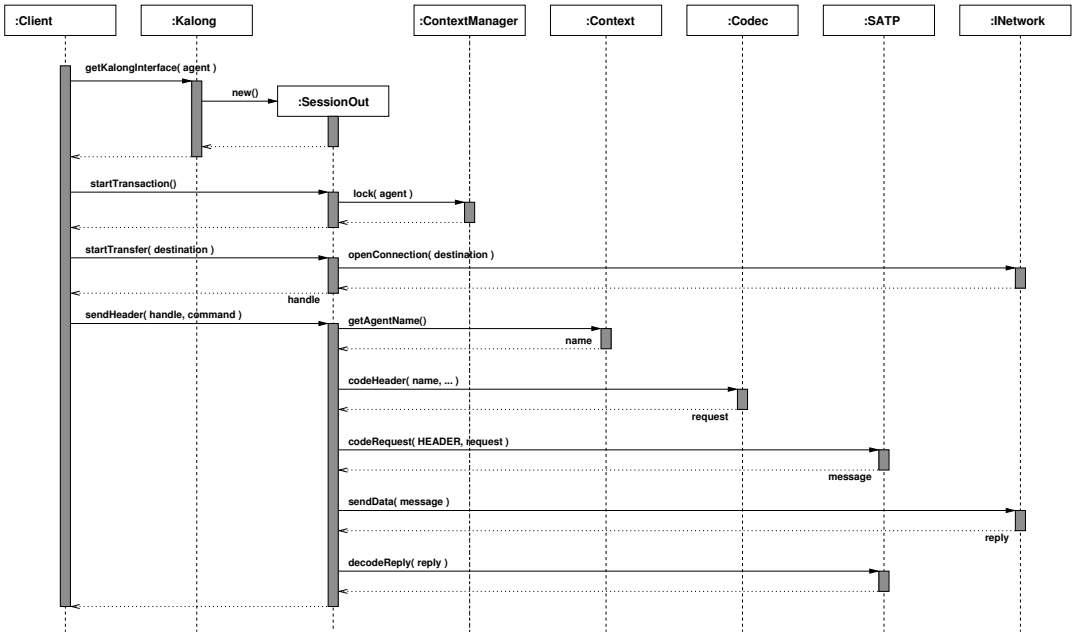


Figure 8.3.: Sequence diagram for the task of opening a network connection and sending a SATP header.

locks the agent context object. After that, the network connection is opened to the destination agency using method *startTransfer*. The session object calls method *openConnection* of the network component to establish a communication channel to the remote agency.

Next, the process of sending a header message is started by the client. The session object handles this process by sequentially communicating to other objects and finally sending the request message to the network component.

It first requests the agent's name by calling the agent's context object. Then it calls class *Codec* to construct a header message according to the SATP specification rule 15, p. 138. It calls class *SATP* to create a request message, where the first byte contains the message type as described in rule 1, p. 137.

Then it calls method *sendData* of the network component to send the message to the remote agency. The return value of this method contains the reply message of the remote server. Finally, this reply message is decoded using a method of class *SATP* again. As a header reply message is either an acknowledge or reject message, no further decoding must happen.

After the header, further SATP messages are to be sent and the processing of these messages works comparable.

9. Coupling Kalong to an Existing Mobile Agent System

In the last two chapters we presented the basic Kalong migration protocol and its reference implementation as Kalong software component. The main advantage of Kalong is that it provides a new optimized migration technique, which allows the programmer of mobile agents to describe a migration strategy in a very fine-grained way. We gave first examples to illustrate how migration strategies can be implemented. In the last part of this thesis (pp. 201), we will present results of several real-world performance measurements.

As we have already mentioned many times, Kalong was designed to work with many, if not all mobile agent systems. It was of particular importance for us not to depend on any design issues that are part of the *programmer's view* of a mobile agent system. However, the process of installing Kalong in an existing mobile agent system is not trivial due to the fact that the main Kalong interface is very low-level.

This chapter is dedicated to the problem of adapting Kalong to an existing mobile agent system and the goal of this chapter is to show an example, how this process can be carried out in practice. Therefore, we have to face two problems:

1. The interface of Kalong is very low-level. We introduced the Kalong migration component as a virtual machine for agent migration. It provides a minimal set of functions or commands to describe a migration strategy. As the focus of Kalong lies on migration optimization, it does not provide any mean for the problem of mobile agent security for example.

   Therefore, we present a technique to extend the basic Kalong protocol. We will show that in fact the Kalong specification only defines the common portions of a protocol family and the Kalong software component offers a way to define new migration protocols on basis of Kalong. Using this technique, it is possible to add security issues to Kalong in a very modular way.

2. We show how to connect Kalong to an existing mobile agent system. We present a new component, named MDL that aggregates Kalong and the network dispatcher component and provides a small and easy-to-use interface. This new component also defines a new abstraction to define and handle migration strategies in a more comfortable way.

9.1. Extending Kalong

In this section, we will present a technique to extend the basic Kalong migration protocol as defined in Chapter 7. The extension mechanism is a very powerful technique to complement Kalong with other services related to agent migration. The extension mechanism allows for example to compress each SATP message before Kalong sends it to the destination agency, or to inspect each incoming class code to detect malicious agents[1]. The extension mechanism defines some selected points, where a user of a Kalong component can modify the structure of each SATP message. It is *not* possible to define *new* SATP message types, but each SATP message can be modified before it is sent to the destination and immediately after it is received at the destination platform. We will now first describe the interface of the extension mechanism and, after that, present some examples how this technique can be used to supplement Kalong with solutions for some selected security problems of mobile agents.

9.1.1. The Kalong Extension Interface

The extension mechanism of Kalong consists of a single interface *IAgentManager* defined in package *org.taf.kalong*. It defines methods that are called by Kalong during the process of an in- or out-migration. For the following we denote the central component of a mobile agent system that controls agent execution, the *agent manager*. An instance of interface *IAgentManager* is part of the agent manager and must be registered after starting the Kalong component using method *registerListener* as described in Section 8.2.1 on page 151. Without this listener, Kalong does not accept in-migrations and cannot start any out-migration.

By registering a listener object, the agent manager is able to modify the structure of each SATP message. To distinguish such a new migration protocol from the basic Kalong migration protocol, each listener must define a *vendor name*, and a *version* in form of a *major* and *minor* number. As already defined above, a migration can only be successful, if both the sender and receiver agency support the same protocol.

The following three methods must be implemented by the agent manager to define the new migration protocol version information.

public *String getVendorName*()
Returns the protocol vendor name.

public byte *getMajorVersion*()
Returns the protocol major version number.

[1] A malicious agent is one that tries to pilfer information from its host environment or tries to damage its host by mean of so-called denial-of-service attacks. We give further examples for malicious agents on pp. 177.

public byte *getMinorVersion*()
Returns the protocol minor version number.

The general communication protocol between Kalong and its listener is that Kalong first informs the listener about the beginning of a migration process (either in- or out-migration). The listener must return a so-called *handle* object to identify this migration process later. This handle object must be given as first parameter in all other methods.

public *Object startOutMigration*(*String agentName*, *URL destination*)
An out-migration has been started. The method must return an object by which this transfer can be identified later.

At a destination agency, Kalong calls the following method, immediately after it has accepted a network connection from a sender agency.

public *Object startInMigration*()
An in-migration has been started. The method must return an object by which this transfer can be identified later.

Methods *codeMessage* and *decodeMessage* are used to code and decode messages or parts of messages. Please note, that a listener must always implement both variants. If it provides a method to *code* a message for example, then there also must be an analogous method to *decode* it again.

The following method is called, whenever Kalong is going to send a SATP message to the destination agency. This is not only the case during out-migrations, as all reply messages are coded using this method too. Thus, for example, if a sender agency has requested a unit for downloading, the destination agency calls this method to code the real unit transfer. The listener can modify this message, for example, by compressing it or signing it digitally. During this process, the listener might need more information about the agent, so a parameter *context* is given, by which the listener can access some important methods of the agent's context object.

public byte[] *codeMessage*(*Object handle*, **byte** *messageType*, **byte**[] *message*, *IContext context*)**throws** *KalongException*
Codes a SATP message that is given in parameter *message* and return it.

The corresponding method that is called after receiving a SATP message, is:

public byte[] *decodeMessage*(*Object handle*, **byte** *messageType*, **byte**[] *coded*, *IContext context*)**throws** *KalongException*
Decodes a received message that is given as parameter *coded*.

The type of message is given as parameter *messageType*. Interface *IContext* defines constant values for all SATP message types.

The next pair of methods is called to code and decode class codes. Parameter *classCode* contains the original byte code of the class with name *className*.

public byte[] *codeClassCode*(*Object handle*, *String className*, **byte**[] *classCode*)**throws** *KalongException*
Codes the given class that is given as Java byte code.

The corresponding method to decode a class is the following.

public byte[] *decodeClassCode*(*Object handle*, *String className*, **byte**[] *codedClass*)**throws** *KalongException*
Decodes the given class code and return a valid Java byte code.

This method must return the original byte code of the class with name *className*, so that it can be loaded and defined by a Java class loader object.

The next two methods are to code and decode the serialized object state of an agent.

public byte[] *codeObjectState*(*Object handle*, **byte**[] *state*)**throws** *KalongException*
Codes the agent's object state.

public byte[] *decodeObjectState*(*Object handle*, **byte**[] *codedstate*)**throws** *KalongException*
Decodes the agent's object state.

The last method must return a byte sequence that can be deserialized using the standard Java deserialization mechanism.

The next three methods are only called during an in-migration. The first one is called by Kalong after a *Header* message is received. The purpose of this method is to decide whether the migration request should be accepted or not. The listener can use all information that were received along the *Header* message.

public boolean *receivedInMigration*(*Object handle*, *String agentName*, *URL*[] *homeAgency*, *URL*[] *lastAgency*)
Decides, whether an in-migration for this agent is allowed or not.

The method must return **true**, if the migration request is accepted, otherwise **false**. In many cases, the given information about the agent's home agency and the agency from which the agent comes from, is not sufficient to make a qualified decision. However, these are the only information, which are sent as part of a SATP header. If more information are needed, for example, certificates of the agent's owner, these must be added to the SATP header message using methods *codeMessage* resp. *decodeMessage*.

The purpose of the last two methods was already explained in Chapter 8.2.3 on page 155.

public *ProtectionDomain getProtectionDomain(Object handle)*
This method is called when Kalong deserializes a received agent. The returned protection domain defines the permissions of this agent.

public void *startAgent(Object handle, Serializable agent, URL destination)*
The given agent was deserialized successful and must now be started by the agent manager.

9.1.2. A First Example: Compression of all SATP Messages

To give a first impression of the range of application of Kalong's extension mechanism, we will present an example, where each SATP message is compressed before sent to the remote agency.

The following example shows parts of the source code of class *ZIPAgentManager* that implements message compression using standard Java techniques provided by classes *GZIPInputStream* and *GZIPOutputStream* defined in package *java.util.zip*. To implement message compression is very easy, as the listener must only implement both methods to code and decode SATP messages, which are *codeMessage* and *decodeMessage*. All other methods to code and decode object states or Java classes, immediately return the given information unchanged.

```
1  public class ZIPAgentManager implements IAgentManager
2  {
3      public String getVendorName()
4      {
5          return "TRACYZIP";
6      }
7
8      public byte getMajorVersion()
9      {
10         return 0x01;
11     }
12
13     public byte getMinorVersion()
14     {
15         return 0x00;
16     }
17
18     public byte[] codeMessage(Object handle, byte messageType, byte[] raw, IContext context)
           throws KalongException
19     {
20         try
21         {
22             ByteBuffer bb = new ByteBufferList();
23             bb.putInt( raw.length );
24
25             ByteArrayOutputStream baos = new ByteArrayOutputStream();
26             GZIPOutputStream zos = new GZIPOutputStream( baos );
27             zos.write( raw, 0, raw.length );
28             zos.close();
```

```
29
30              bb.putBytesWithLength( baos.toByteArray() );
31              return bb.toByteArray();
32
33          } catch( Exception e )
34          {
35              throw new KalongException( e );
36          }
37      }
38
39      public byte[] decodeMessage(Object handle, byte messageType, byte[] coded, IContext context)
            throws KalongException
40      {
41          try
42          {
43              ByteBuffer bb = new ByteBufferList( coded );
44              int length = bb.getInt();
45
46              byte[] zipped = bb.getBytesWithLength();
47              byte[] unzipped = new byte[ length ];
48
49              ByteArrayInputStream bais = new ByteArrayInputStream( zipped );
50              GZIPInputStream zis = new GZIPInputStream( bais );
51
52              int pos = 0;
53              do
54              {
55                  pos += zis.read( unzipped, pos, length-pos );
56              } while( pos < length );
57              zis.close();
58
59              return unzipped;
60          } catch( Exception e )
61          {
62              throw new KalongException( e );
63          }
64      }
65
66      public byte[] codeObjectState(Object handle, byte[] state) throws KalongException
67      {
68          return state;
69      }
70
71      public byte[] decodeObjectState(Object handle, byte[] codedstate) throws KalongException
72      {
73          return codedstate;
74      }
75
76      public byte[] codeClassCode(Object handle, String name, byte[] classCode) throws
            KalongException
77      {
78          return classCode;
79      }
80
81      public byte[] decodeClassCode(Object handle, String name, byte[] codedClass) throws
            KalongException
82      {
83          return codedClass;
```

```
84      }
85
86      // some methods are missing
87  }
```

Both methods work without regard to the message type, as simply all SATP messages are to be compressed. The original message is written to an instance of class *GZIPOutputStream*, which itself uses an instance of class *ByteArrayOutputStream* to store the compressed data. The format of the compressed message consists of a four-byte integer that contains the length of the original message, followed by a byte array that contains the compressed message. To create this message format, class *ByteBuffer* is used that is part of the Tracy project and defined in package *org.taf.util*. It provides several methods to code Java's primitive data types into flat byte arrays. To decode a compressed message is very straightforward. The message is given to an instance of class *GZIPInputStream*, from which the inflated data is read until all bytes are received. Finally, the original message is returned to Kalong.

As this example shall only give a first impression how to use this interface, we do not show the implementation to verify an in-coming agent or to start one.

9.1.3. Security Problems of Mobile Agents

We now come to a very important part of mobile agents' research. Until now, we did not consider mobile agents security aspects within this thesis, as that was not a major research issue within Kalong. However, we are aware of the fact that mobile agent security is of major interest and the absence of a comprehensive security technique prevents the widespread use in real-world applications.

In the last years, many research groups world-wide have focused on security aspects of mobile agents and a huge amount of papers were published. It is not the goal of this section to give an overview of the state-of-the-art and the interested reader is pointed to the following book by Vigna [1998a], the thesis of Karnik [1998] and the following papers [Farmer et al., 1996a,b; Jansen, 2000; Karjoth et al., 1997; Karnik and Tripathi, 2001; Roth and Conan, 2001; Roth and Jalali, 2001].

In this section, we will first give an impression of what can go wrong when a mobile agent roams a network and briefly discuss security attacks. After that, we will show how the extension mechanism of Kalong can be used to supplement the basic Kalong migration protocol by some solutions for fundamental security problems.

Passive Attacks Passive attacks are directed to the communication link between agencies. The adversary does not interfere with the messages sent over the network. No data is modified by such an attack, but the data is monitored to extract useful information. As neither communication partner does notice this attack, it is normally

This box explains some fundamental notions with regard to security services and the notation used in this section.

Symmetric cryptosystems use a common key K that is shared between the sender Alice and the receiver Bob to encrypt messages but must be kept secret against other entities. Alice uses key K to encrypt a message, which is then sent to Bob, who is able to decrypt it with the same key K. An eavesdropper Eve may be able to read Alice's message but cannot understand it, because she does not have the key K. The advantage of symmetric cryptosystems is speed, whereas secret key distribution between Alice and Bob is a considerable technical problem.

In *asymmetric cryptosystems* or *public-key cryptosystems* each principal has two keys – a *public key* which is shared with all other principals (for example by posting it on a public key directory service) and a corresponding *private key*, which must be kept secret by the owner. The concept of public-key cryptosystems is that messages can be encrypted using either key and can only be decrypted using the other key of the key pair. For example, Alice encrypts a message to Bob with Bob's public key K_B^+, which she obtains by looking at a public directory. This message can only be deciphered using the corresponding private key K_B^-, which is only known by Bob. The advantage of asymmetric cryptosystems is a comparatively easy key distribution, as the public key can be published elsewhere in the network and only the private key must kept secret. The disadvantage is its low speed. In practice, public key distribution is supplemented by techniques to verify the identity of the owner using so-called *certificates*, so that Alice can trust that the public key she has obtained from the directory service actually belongs to Bob.

Using *digital signatures* it is possible to verify that a message was originated by the sender and to check the integrity of a message. Alice encrypts a message with her private key K_A^-. When Bob receives the message, he can decrypt the message using Alice's public key K_A^+. If it was successful, he can be sure, that the message was encrypted using the corresponding private key, which is only known by Alice. Thus, Bob can be sure, that Alice is the author of the received message. Due to the low speed of public-key cryptosystems, message signing is often combined with *digests*.

A *digest* or *hash value* is the result of a *one-way hash function* computed over a message. Such a hash function maps any arbitrary sized byte sequence to a fixed sized byte sequence, for example 16 byte in the case of the MD5 function. An important feature of a hash function is that there does not exist a simple technique to find the original value x, when only the hash value $h(x)$ is known. Additionally, it also computationally infeasible to find another y so that $h(x) = h(y)$. In other words, if two hash values are the same, it can be inferred that the two original values are identical. Digests are often used in combination with digital signatures. As digest are a good mean to condense data while keeping its uniqueness, not the original message is encrypted using private key K_A^-, but the message's digest.

Figure 9.1.: The notation used in this section for cryptographic terms. Inspired by Lange and Ishima [1998].

difficult or impossible to detect. The most simplest variant of a passive attack is *eavesdropping*, where the adversary monitors the communication link between two agencies and captures agents to extract useful information from the agent's state or code for example. This might result in a leakage of sensitive information. Another form of attack is *traffic analysis*, which also works in the case when each message is encrypted, because it is not important that data is readable (understandable) to the attacker. Here, the adversary attempts to find pattern in the communication between two agencies, which might bring him in a situation where he can derive certain assumptions based on these patterns.

Active Attacks Active attacks include security threats where an agency tries to manipulate agent code or data for its own good or an agent tries to attack its hosting agency by deliberately using resources, for example memory or a CPU. Most prominent examples of this kind of attack are *alteration*, where an agent's data is deleted or tampered with by an agency, and *impersonation*, where a malicious agent impersonates another agent instance, which has more comprehensive permissions than the malicious agent itself.

Thus, we distinguish between *malicious agents* and *malicious agencies* as the active entity. The first case can be solved to some extent using a combination of cryptographic techniques and basic security mechanisms provided by the Java virtual machine. The technique works in four steps.

1. First, an agency tries to detect malicious agents migrating into it by inspecting agent code and rejecting agents that infringe programming rules for benevolent mobile agents. For example, an agent should not implement method *finalize*, which is invoked by the garbage collector to clean up resources used by this object instance. A malicious agent might use this method to attack the garbage collector by blocking the current thread, which has the effect that no memory is freed any more and the whole agency eventually crashes.

2. The second step is *agent authentication*, which includes verifying the developer of the agent and checking the integrity of the agent's code.

3. The next step is *agent authorization*, where the agent is given certain permissions according to its principals.

4. The last step is to execute an agent in a separate environment, where each access to host resources is verified against the agent's permissions and no agent can access any other agent instance on the level of Java objects.

   In Java this restricted execution environment is called *sandbox*. Due to the limitation of the Java programming language, this kind of host protection is not complete. For example, Java does not provide a means to control how much

memory an agent uses, so a possible attack might be for an agent to allocate constantly memory. This is also called a denial of service attack.

The second case considers malicious agencies that try to tamper with agent code or data. Unfortunately, this type of attack is much more difficult to solve as an agent must disclose all its code and at least part of its data to its host, if it wants to be executed. Therefore, no general guarantee can be given that an agent is not maliciously modified [Farmer et al., 1996b]. For example, a malicious agency can steal useful information stored in the agent, modify data carried by the agent to its own advantage, or modify an agent's code in a way that the agent acts maliciously on other agencies and its home agency. Other examples can be found in Lange and Ishima [1998] and Karnik [1998]. However, the situation is not so irremediably, as it seems to be. Using cryptographic techniques, especially data signatures and data encryption, a situation can be achieved where illegal access to an agent's code and data can be detected – but not prevented.

The basic concept is to rely on the notion of *trust* as usual in public key infrastructures. Benevolent agencies help benevolent agents by providing such security services and to protect themselves from malicious agents and malicious agencies. To make this concept more tangible, we give two examples that we adopted from Karnik [1998]. In the first one, we present a technique to protect a data item of an agent against illegal manipulations. In the second example, we present a technique to protect a data item so that it is only usable on selected agencies, all of them assumed a priori to be non-malicious.

Read-only Data Items In frequent cases, an agent carries data items whose value remain constant over the whole life-time of the agent. For example, the itinerary was given by the agent's owner and contains addresses of all agencies the agent should visit. This information should be unmodifiable[2]. To achieve such *read-only* data items, the following technique can be used.

At the agent's home agency, this data item is signed with the private key of the owner. According to the notion presented in Figure 9.1 on page 178, we can write $signature = Sig_A(data) = K_A^-(h(data))$, where $h(data)$ is the digest of the data item, computed using a one-way hash function, for example MD5, and A is the owner of the agent. The signature must be computed at the agent's home agency, as the owner's private key is only available there. The signature becomes part of the agent.

When the agent accesses this data item, the host agency verifies whether the read-only data item has been tampered with. To do this, the agency needs the public key K_A^+ of the owner, usually in form of a trusted certificate. It computes the digest

[2]Of course, a malicious agency has other techniques to manipulate an agent's route, for example by just sending the agent to another destination than requested.

equally to the home agency and compares it to the signature that the agent carries. Thus, it checks, if $h(data)$ equals $K_A^+(signature)$. If both match, the agency can assume that the data item was not modified.

A malicious agency has two ways to attack this technique. First, it could modify the data item in a way that the digital signature is still valid. Although, this is not impossible, due to the hash function used it is assumed to be computationally infeasible to find another data item with the same digest. Second, the agency modifies the data item and the signature so that the data items seems to be valid, although it is not. As it is a fundamental concept of the whole public key cryptography that the private key is only known to the owner, this attack can be ruled out.

Protect a Data Item for Selected Agencies The next problem is to protect data items in a way so that they can only be read at certain agencies but not at all agencies the agent visits. The necessity occurs in a case where data items are defined at the home agency but shall only be read at some other agencies as well as in the case that a data item is defined at any agency and shall only be readable at the agent's home agency.

This problem can be solved by encrypting the data item with the public key $K_T^+(data)$ of the agency for which the data item is targeted. An additional signature $Sig_A(K_T^+(data))$ that binds the address of the target agency to the encrypted data item using the agent's private key can be used to ensure that the target address has not been tampered with.

This solution has a disadvantage, as the data item must be encrypted n times, if it should be readable at n agencies. A better solution is proposed by Roth and Conan [2001], who use a *hybrid* encryption technique. The data item is encrypted using symmetric encryption, where only key k (which is much shorter than the original data item) is encrypted n times using the public key of all target agencies.

If the agent gets a new data item at an arbitrary agency, which should not be readable at any other agency in the following but only at the agent's home agency, then it can be encrypted with the public key of the home agency using the same technique as described above. Besides, using the new features of the Kalong migration protocol, it is also possible to send this data item immediately back to the agent's home agency, so that no other agency has a chance to read it anyway.

9.1.4. How to Implement Security Solutions With Kalong

Now we will show how Kalong's extension mechanism can be used to implement these security solutions.

Class Code Filtering Code filtering is used to inspect in-coming classes and check whether they implement code fragments considered to be malicious. An example for

this was already presented above. When an agent implements method *finalize*, it might attack the garbage collector thread.

Class code filtering can be implemented using the Kalong extension mechanism. When ever Kalong receives a code unit, it calls method *decodeClass* of the Kalong listener object for each class. This method gets the class name and the class byte code, as it was received from the network, as parameter. It must return a valid Java byte code for this class. For the following we assume that no other class coding is implemented so that parameter *codedClass* already contains valid Java byte code, which is only inspected within this method.

```
1  class FilterAgentManager implements IAgentManager
2
3    // ...
4
5    public byte[] decodeClass(String name, byte[] codedClass) throws
         KalongException
6    {
7      filterClass( name, codedClass );
8      return codedClass;
9    }
10
11   void filterClass(String name, byte[] bytecode) throws KalongException
12   {
13     try
14     {
15       ClassFileStructure cfs = new ClassFileParser(new BycalDataInputStream(new
             ByteArrayInputStream(bytecode))).parseClassFile();
16       ClassStructure cs = new ClassStructure(cfs);
17       Method fin = cs.getMethod("finalize()");
18
19       if(fin != null)
20       {
21         throw new KalongException("class␣has␣a␣finalize␣method");
22       }
23     } catch(IOException e)
24     {
25       throw new KalongException("class␣code␣cannot␣be␣analyzed␣and␣is,␣therefore
             ,␣not␣accepted");
26     } catch(AccessFlags_Exception f)
27     {
28       throw new KalongException("class␣code␣cannot␣be␣analyzed␣and␣is,␣therefore
             ,␣not␣accepted");
29     }
30   }
31 }
```

Method *decodeClass* calls method *filterCode* and returns the byte code as it was received, if the byte code filter did not find any malicious code. Otherwise, this method throws an exception with an appropriate error message, which is sent back to the sender agency.

Method *filterClass* uses the ByCAl tool, which was developed as part of the Tracy project to analyze Java classes on the level of byte code. The class file is read using class *ClassFileStructure* and analyzed using class *ClassStructure*. The latter class provides a method to check, whether a method with a given name exists (*getMethod*). If a method with name *finalize* is found, it throws an exception, otherwise it returns silently.

Agent Authentication Agent authentication can be done by verifying a digital signature of the agent's owner or the last agency, the agent came from. Digitally signing with the agent's owner private key can only be done at the agent's home agency. Therefore, only the immutable or static part of an agent can be digitally signed, all mutable data as for example the agent's object state or data items of the external state cannot be signed with the owner's key.

We show an example, where the static parts of a SATP header message, which comprise of the agent's name and its home agency are digitally signed at the agent's home agency. At each host the agent visits, this signature is verified against the owner's public key.

To sign a header message can be implemented using method *codeMessage* of interface *IAgentManager*. This method is called by Kalong whenever a SATP message is to be sent to a destination agency. The message type is given as parameter *messageType*. The header as it was created by Kalong is given as parameter *message*.

```
1  class SigningAgentManager implements IAgentManager
2  {
3    // ...
4
5    public byte[] codeMessage(byte messageType, byte[] message, IContext context)
         throws KalongException
6    {
7      if(messageType == IContext.HEADER)
8      {
9        if(agentCertificate != null)
10       {
11         try
12         {
13           byte[] codedAgentCertificate = agentCertificate.getEncoded();
```

```
14
15              ByteBuffer bb = new ByteBufferList();
16              bb.putBytesWithLength(codedAgentCertificate);
17              bb.putBytesWithLength(message);
18
19              if(agentNameSignature == null)
20              {
21                ByteBuffer buffer4sig = new ByteBufferList();
22                buffer4sig.putString(context.getAgentName()).putURLArray(context.
                      getHomeAgency());
23                agentNameSignature = signBytes(signEngine, agentPrivateKey, buffer4sig
                      .toByteArray());
24              }
25
26              bb.putBytesWithLength(agentNameSignature);
27              return bb.toByteArray();
28
29            } catch(Exception f)
30            {
31              f.printStackTrace();
32              return null;
33            }
34          } else
35          {
36            return null;
37          }
38        } else
39        {
40          return message;
41        }
42    }
43 }
```

This code excerpt does not show how to obtain certificates or private keys from a local *keystore* file, as this is done using fundamental Java security mechanisms. We assume that variable *agentCertificate* contains the agent owner's certificate and *agentPrivateKey* already contains the private key of the agent's owner. For sake of simplicity, we send the agent owner' certificate as part of the header message too. In real applications, only some information about this certificate would be part of the header and the destination agency would have to load the certificate from a public key server.

This method creates a new header message that comprises of three parts. First, it contains the owner' certificate (line 16), second the original header message (line 17), and third, the digital signature (line 26). The conditional in line 19 decides whether

the agent's signature must be created (because the current agency is the home agency) or can be reused. In the latter case, variable *agentNameSignature* already contains the agent's signature. Otherwise, the agent's name and its home agency URLs are signed in line 23. To verify a signature, the destination agency must implement method *decodeMessage* of interface *IAgentManager*, as shown in the following excerpt.

```
1  public byte[] decodeMessage(byte messageType, byte[] message, IContext context
       ) throws KalongException
2  {
3    if(messageType == IContext.HEADER)
4    {
5      ByteBuffer bb = new ByteBufferList(message);
6      byte[] codedAgentCertificate = bb.getBytesWithLength();
7      byte[] msg = bb.getBytesWithLength();
8      agentNameSignature = bb.getBytesWithLength();
9
10     try
11     {
12       CertificateFactory cf = CertificateFactory.getInstance("X509");
13       ByteArrayInputStream bais1 = new ByteArrayInputStream(
             codedAgentCertificate);
14       agentCertificate = (X509Certificate) cf.generateCertificate(bais1);
15       agentPublicKey = agentCertificate.getPublicKey();
16
17     } catch(Exception e)
18     {
19       agentCertificate = null;
20       agentPublicKey = null;
21       remoteAgencyCertificate = null;
22       remoteAgencyPublicKey = null;
23     }
24
25     return msg;
26   } else
27   {
28     return message;
29   }
30 }
```

First, the header message is split into the three components: certificate, original header message, and digital signature (lines 6–8). After that, the owner's certificate is initialized and the owner's public key is requested (lines 12–15).

The process of verifying the digital signature is done in an other method, when the header message is checked to decide whether an agent shall be accepted.

```
1  public boolean validateHeader(String agentName, URL[] homeAgency, URL[]
       lastAgency)
2  {
3    try
4    {
5      if( keystore.getCertificateAlias(agentCertificate) == null )
6      {
7        return false;
8      } else
9      {
10       ByteBuffer buffer4sig = new ByteBufferList();
11       buffer4sig.putString(agentName).putURLArray(homeAgency);
12       return verifySignature(signEngine, agentPublicKey, buffer4sig.toByteArray
             (), agentNameSignature);
13     }
14   } catch(Exception e)
15   {
16     e.printStackTrace();
17     return false;
18   }
19 }
```

First, it is checked, whether the owner's certificate is trusted (line 5), and second (line 12), the signature is verified. Using the same technique, it is possible to sign all classes using the owner's public key and verify their integrity.

Read-only Data Item The next two examples focus on protecting data items against illegal modifications or illegal access. This service can also be implemented using the extension mechanism of Kalong but we will show here a version that implements an adapter to access the main Kalong interface. The general concept is that agents can not only use the two methods *setDataItem* and *getDataItem* as defined in interface *IKalong*, but also use two new methods *setReadOnlyDataItem* and *setEncryptedDataItem*.

Here is the code to sign a data item. We assume that the agent has already defined the keystore alias of its owner in variable *alias* and the the keystore password in variable *password*.

```
1  public void setReadOnlyDataItem(String name, Serializable value) throws
       MDLException
2  {
3    String alias = null;
4    char[] password = null;
```

```
5     SignedObject signedObject = null;
6
7     // request of owner's alias and password is not shown here
8
9     try
10    {
11      priKey = (PrivateKey) keystore.getKey( alias, password );
12      signedObject = new SignedObject( value, priKey, signEngine );
13    } catch( Exception e )
14    {
15      throw new MDLException( e );
16    }
17
18    setDataItem( name, signedObject );
19  }
```

The owner's private key is read from the keystore file in line 11 and the data item is encapsulated together with its signature by an object of class *SignedObject*, which automatically signs the data item in line 18.

When the data item is accessed using method *getDataItem*, it must be checked of what type the data item is. If the data item is an object that is an instance of *SignedObject*, then the signature is verified and the data value returned to the caller. The following source code shows how to access Kalong to read a data item with name *name* (line 13). In line 22 it is checked whether the data item is of type *SignedObject*. We assume that variable *pubKey* is defined outside this method and already contains the public key of the agent's owner. Finally, in line 28, the object is verified and the original data item returned to the caller in line 41.

```
1   public Serializable getDataItem(String name) throws MDLException
2   {
3     Serializable dataValue = null;
4     SignedObject signedObject = null;
5     boolean dataVerified = false;
6
7     /*
8      * Read the data item from the external state.
9      */
10    try
11    {
12      kalongInterface.startTransaction();
13      dataValue = kalongInterface.getDataItem(name);
14    } catch(Exception e)
15    {
16      throw new MDLException(e.getMessage());
```

```
17      } finally
18      {
19        kalongInterface.commit();
20      }
21
22      if( dataValue instanceof SignedObject )
23      {
24        signedObject = (SignedObject)dataValue;
25
26        try
27        {
28           dataVerified = signedObject.verify( pubKey, signEngine );
29        } catch( Exception e )
30        {
31          throw new MDLException( "signed data item cannot be verified due to: "
                + e.getMessage() );
32        }
33
34        if( ! dataVerified )
35        {
36           throw new MDLException( "signed data item was tampered with" );
37        }
38
39        try
40        {
41          return (Serializable)signedObject.getObject();
42        } catch( Exception e )
43        {
44          throw new MDLException( e );
45        }
46      }
47
48      // ...
49    }
```

Protect Data Items for a Target Agency Finally, we present the code to encrypt a data item so that it can be only read at a single target agency. The agent calls this method to store a data item under the given name, which is encrypted with the public key of the agency whose local keystore alias is given in parameter *targetAlias*.

Data encryption is done in Java using objects of class *Cipher* and we assume that an object with name *rsaCipher* has been initialized to use asymmetric RSA encryption. In line 9 the cipher is initialized for *encryption* using the public key of the target agency which is obtained from the local keystore file. Data encryption

works using the same technique as described above with signed object. An object of type *SealedObject* is used, which serializes the data item and encrypts it using the given cipher object (line 10). Finally, this object is stored in the agent's external data state (line 17).

```
1  public void setEncryptedDataItem(String name, Serializable value, String
       targetAlias) throws MDLException
2  {
3    SealedObject sealedObject = null;
4    PublicKey targetPublicKey = null;
5
6    try
7    {
8      targetPublicKey = keystore.getCertificate( targetAlias ).getPublicKey();
9      rsaCipher.init( Cipher.ENCRYPT_MODE, targetPublicKey );
10     sealedObject = new SealedObject( value, rsaCipher );
11   } catch( Exception e )
12   {
13     e.printStackTrace();
14     throw new MDLException( e );
15   }
16
17   setDataItem( name, sealedObject );
18 }
```

Data decryption is implemented in method *getDataItem*. In addition to the source code presented above, we show here what must be done when the data item is of type *SealedObject*.

```
1  public Serializable getDataItem(String name) throws MDLException
2  {
3
4    // ...
5
6    } else if( dataValue instanceof SealedObject )
7    {
8      try
9      {
10       rsaCipher.init( Cipher.DECRYPT_MODE, agencyPrivateKey );
11       return (Serializable)((SealedObject)dataValue).getObject( rsaCipher );
12     } catch( Exception e )
13     {
14       throw new MDLException( "data item cannot be decrypted due to: " + e
             .getMessage() );
```

```
15          }
16        } else
17
18        // ...
19
20    }
```

In line 10 the cipher object is initialized for *decryption* using the private key of the current agency. Finally, in line 11 the object is decrypted and returned to the caller.

9.2. Coupling Kalong to Tracy2

At the end of this part of the thesis we will now describe how Kalong was adapted to work with the Tracy2 mobile agent system. The detailed architecture of a Tracy2 agent server is presented in the appendix. The general concept is that of a *micro kernel* that provides basic services of a mobile agent server, as for example agent execution and thread management. On top of the micro kernel, several so-called *features* are responsible to provide other services, as for example agent migration (which is implemented using Kalong), agent communication, a blackboard etc. The goal of this section is to give an impression how Kalong was adapted to work as a Tracy2 feature.

9.2.1. MDL

As described already, Kalong uses another software component, named network adapter, to access the network using different transmission protocols. Kalong only implements the basic Kalong migration protocol and uses the network adapter for all tasks that are related to network communication. We presented an example to connect these two components with each other on Page 152.

To reduce the complexity of the migration feature for Tracy2, we decided to aggregate these two components into a new one, which is named MDL (Migration Definition Layer). MDL provides several services, which in sum simplify using Kalong. All classes that belong to this component are defined in package *org.taf.mdl*.

- MDL starts instances of the Kalong component and the network adapter component and connects both components with each other. MDL provides two adapter classes for this task.
- MDL can be configured using a *Map* data structure, which contains the network transmission protocols to start. MDL relieves the user from manually registering network protocols with the network adapter and start a network server for each protocol.

- MDL has a very small interface as compared to Kalong. There is only a single method *startMigrationStrategy* that the client of a MDL component must use to initialize the whole migration process. MDL defines a new level of abstraction and introduces class *MigrationStrategy*.
- MDL registers an own listener object of type *IAgentManager* with Kalong, which already implements all the security solutions presented in the last section. MDL defines an own listener interface, which only defines a single method that is called to start agent execution for in-coming agents.

The main method of MDL is the following one that is used to initialize a migration process.

public void *startMigrationStrategy*(*String agentName*, *Serializable agent*, *String name*, *Map properties*)**throws** *MDLException*
Start the migration process for agent *agentName* using migration strategy *name* and the given migration properties.

The first two parameters contain the agent's name and the agent object. The third parameter contains the name of a migration strategy. Under this name, the MDL component must know a class that extends class *MigrationStrategy*, which is defined in package *org.taf.mdl*. A class of type *MigrationStrategy* aggregates all commands that describe the migration process for a specific migration strategy. Classes that describe a migration strategy must be registered with MDL under a user-defined name, so that agents can simply select the migration strategy to use for the next hop by providing this name as parameter *name* in method *startMigrationStrategy*.

For example, the *push-all-to-next* migration strategy is implemented as shown in the following source code:

```
1  public class PushAgent implements MigrationStrategy
2  {
3      public void run(Serializable agent, IKalong kalong, Map properties) throws
           MDLException
4      {
5          int[] allUnits = null;
6
7          try
8          {
9              kalong.startTransaction();
10             kalong.setObjectState( agent );
11
12             allUnits = kalong.getUnits();
13             if( allUnits.length == 0 )
14             {
```

```
15                  String[] allClasses = kalong.getClassNames();
16                  String[] filtered = ArrayUtils.filter( allClasses, new String[] { "java.*"
                        , "javax.*", "org.taf.*" } );
17                  if( filtered.length > 0 )
18                  {
19                      kalong.defineUnit( filtered );
20                  }
21              }
22
23              URL destination = (URL)properties.get( MigrationStrategy.
                    MDL_DESTINATION );
24
25              int[] allUnits = kalong.getUnits();
26              String[] allDataItems = kalong.getDataItems();
27
28              Object handle = kalong.startTransfer( destination );
29              kalong.sendHeader( handle, IKalong.NOOP );
30              kalong.sendADB( handle, true );
31
32              if( allUnits.length > 0 )
33              {
34                  kalong.sendUnits( handle, allUnits );
35              }
36
37              kalong.sendState( handle, allDataItems );
38
39          } catch( Exception e )
40          {
41              e.printStackTrace();
42              throw new MDLException( e.getMessage() );
43
44          } finally
45          {
46              if( kalong.prepare() )
47              {
48                  kalong.commit();
49              } else
50              {
51                  kalong.rollback();
52              }
53          }
54
55      }
56  }
```

This class describes the common pattern for this kind of migration and it must be configured to work for a specific agent by the forth parameter given to method *startMigrationStrategy*, and which is given to the concrete migration strategy object as third parameter of method *run*. This object of type *Map* might contain several key–value pairs by which the agent manager can configure the migration process. At least, this map must contain a pair where the key equals the String literal *mdl.migration* and the corresponding value contains a URL object of the next destination.

The whole documentation of this component can be found on the enclosed CD-ROM.

9.2.2. Migration as Tracy2 Feature

Tracy2 defines an abstract class named *Feature* which must be implemented by each concrete feature. The complete definition of class *Feature* is omitted and we focus on the few methods that are used within the migration process.

The following source code gives an impression on how the migration feature is implemented. It shows the constructor, where the feature configures and starts an instance of the MDL component.

```
1  public class Kalong extends Feature
2  {
3      final private MDL mdl;
4      final private IMDLListener listener;
5
6      /** Creates a new instance of Kalong */
7      public Kalong( String featureName ) throws Exception
8      {
9          super( featureName );
10
11         Map properties = new HashMap();
12         properties.put( MDL.NETENGINECLASS + ".1", "org.taf.network.tcp.
               TCPEngine" );
13         properties.put( MDL.NETENGINEPORT+ ".1", new Integer( 5555 ) );
14         properties.put( MDL.NETENGINECLASS + ".2", "org.taf.network.ssl3.
               SSLEngine" );
15         properties.put( MDL.NETENGINEPORT+ ".2", new Integer( 6666 ) );
16
17         mdl = new MDL( properties );
18
19         /*
20          * Register a simple migration strategy.
21          */
22         mdl.registerMigrationStrategy( "push", org.taf.mdl.strategies.PushAgent.class
               );
```

```
23
24          /*
25           * Create a listener object for the MDL component, which is called by MDL in
                 the case of
26           * an in-migration. Register this listener object with MDL.
27           */
28          listener = new MDLListener();
29          mdl.registerListener( listener );
30      }
31
32      // ...
```

In Tracy2 any kind of agent is an object that implements interface *Runnable*. This design decision was made in conformance to the general concept that agents should be exchangeable with other mobile agent systems – and interface *Runnable* is the least common denominator for this. Mobile agents must additionally be marked as *Serializable*. As a consequence, an agent does not have its own methods to communicate to other agents or to initialize the migration process. A new kind of communication must be established for agents to be able to use feature services.

The concept used in Tracy2 is that of a so-called *context* object. Each feature defines an own interface that must extend interface *IFeatureContext*. This interface defines methods by which the agent can communicate to the feature. The feature maintains a single context object for each agent that uses the feature. For example, the migration feature provides the following context interface:

```
1  public interface IMigrationContext extends IFeatureContext
2  {
3      public void setDestination( URL destination );
4      public void setMigrationStrategy( String name );
5      public void setDataItem( String name, Serializable value ) throws
           KalongException;
6      public Serializable getDataItem( String name ) throws KalongException;
7      public void setReadOnlyDataItem( String name, Serializable value ) throws
           KalongException;
8      public void setEncryptedDataItem( String name, Serializable value, String target
           ) throws KalongException;
9  }
```

The first two methods are used to define the URL of the next destination and the name of the migration strategy to use. All other methods are used to access the external data state of the agent that is maintained by Kalong. An agent can request its own context object by calling a static method of class *Context*, which is defined

as part of Tracy2. This method accepts the name of the feature as parameter and dispatches the request to the desired feature. In the following example, an agent is shown that requests its migration context object, defines a data item, and sets the migration properties.

```
1  public class SampleAgent implements Runnable
2  {
3    public void run()
4    {
5      try
6      {
7        IMigrationContext migrationContext = Context.getContext( "migration" );
8        migrationContext.setDataItem( "data1", new Integer( 100 ) );
9        migrationContext.setDestination( new URL("tcp://tatjana.cs.uni-jena.de:4040"
             ));
10       migrationContext.setMigrationStrategy( "push" );
11     } catch( Exception e )
12     {
13       e.printStackTrace();
14     }
15   }
16 }
```

As can be seen, the migration context does not provide a method to initialize the migration process directly. The reason for this is that it is not the agent on its own that can decide, whether it can migrate, because other features might disapprove this action. For example, a communication feature might refuse an agent from migration, if it still wants to deliver messages to the agent. Therefore, every time when method *run* of an agent terminates, the micro kernel of Tracy2 carries out a voting protocol, where each feature is asked whether the agent might be killed or fall asleep (for example for wait for new messages) or to migrate. Only if all features agree, the migration feature is asked to initialize the migration process.

The method to initialize the migration process is named *removeAgent* and the source code is shown in the following:

```
1  public void removeAgent(Runnable agent)
2  {
3      MigrationContext context = (MigrationContext)getContext0( agent );
4
5      if( context != null )
6      {
7          URL destination = context.getDestination();
8          String strategy = context.getMigrationStrategy();
```

```
9           String agentName = context.getAgentName();
10
11          if( destination == null || strategy == null )
12          {
13              removeAgentContext( agent );
14              return;
15          }
16
17          Map migrationProperties = new HashMap();
18          migrationProperties.put( MigrationStrategy.MDL_DESTINATION,
                destination );
19
20          try
21          {
22              mdl.startMigrationStrategy( agentName, (Serializable)agent, strategy,
                    migrationProperties );
23              removeAgentContext( agent );
24          } catch( MDLException e )
25          {
26              /*
27               * Migration failed. Restart the agent now.
28               */
29          }
30      } else
31      {
32          removeAgentContext( agent );
33      }
34  }
```

First, it is checked whether the agent has already requested a context object at all. If not, the method terminates immediately. Otherwise, the URL of the next destination and the name of the migration strategy are read from the context object. If both are defined, the migration process is started by invoking the appropriate method of component MDL. If the migration process was not successful, the agent must be registered locally and the micro kernel must be informed to execute the agent again. The latter is not shown in the source code above.

If an agent has been received by a Kalong instance, the MDL component is informed about this event, which immediately forwards it to the migration feature. The migration feature implements interface *IMDLListener* and defines the following method to start an agent.

```
1   class MDLListener implements IMDLListener
2   {
```

```
3   public void startAgent(String agentName, Serializable agent, URL destination,
        IContext context)
4   {
5     if( kernel != null )
6     {
7       if( agent instanceof Runnable )
8       {
9         MigrationContext migContext = (MigrationContext)getContext( (Runnable)
              agent );
10        migContext.setKalongContext( context );
11
12        kernel.runAgent( (Runnable)agent );
13      } else
14      {
15        throw new IllegalArgumentException("given agent is not of type Runnable
              ");
16      }
17    }
18  }
19 }
```

The given object of type *IContext* provides methods to access the agent's external data state. This reference is given to the agent's migration context, so that the context can forward requests to the external data state directly to Kalong.

9.3. Summary of Part III

In this part of the thesis we specified the new Kalong mobility model and the SATP protocol for agent transmission. We introduced the Kalong software component that is the reference implementation of the Kalong mobility model. The main advantages of the Kalong software component are:

- Kalong can be seen as a virtual machine for agent migration. It provides generalized low-level functions that can be used to control the migration process of a mobile agent in a very fine-grained way.
- The Kalong software component is independent of any mobile agent system. It does not rely of specific design issues made by the agent system and should, therefore, be usable in almost any mobile agent system.
- The Kalong software component is extendable inasmuch as it defines several points in the migration process where the agent manager is called to modify or extend the structure of each SATP message. We have shown how basic security issues can be implemented using the extension mechanism of Kalong.

The next part of the thesis will focus on a first evaluation of Kalong using several real-world experiments.

Part IV.

Evaluation

In the last part of this thesis, we will evaluate our new migration model Kalong. We will first provide several examples for migration strategies and point out other application areas where the new features of Kalong may be used to increase the performance of mobile agent migration. After that, we will discuss the results of several performance experiments we have done in order to demonstate the benefit of Kalong in terms of saved network transmission time. Finally, the last chapter of this thesis gives a conclusion and an outlook to further developments.

10. Some Thoughts on the Applicability of Kalong

In this chapter we will reason about the applicability of Kalong to improve the overall performance of mobile agents. We will provide several examples how the migration process can be improved using the new features of Kalong. In a first step, we will present in Section 10.1 a catalog of simple migration strategies. This catalog contains all migration strategies mentioned in Chapter 5.1 on page 61 and, therefore, covers the current state of the art. Then, in Section 10.2, we will present several examples to illustrate the possibilities of Kalong to influence and adapt the migration process by the programmer, for example to fetch classes in parallel. Finally, in Section 10.3 we will reason about more sophisticated migration strategy and we will present first results on the way towards a fully automated migration planner that always chooses the best migration strategy.

10.1. Examples for Simple Migration Strategies

In this section we will present a catalog of migration strategies to show that it is possible to implement all those strategies presented in the chapters before by using Kalong. All migration strategies are located in package *org.taf.mdl.strategies*. The class diagram for all implemented migration strategies is depicted in Figure 10.1. It includes an abstract class *AbstractMigrationStrategy* that implements interface *MigrationStrategy* of package *org.taf.mdl*. The abstract class provides several methods that are common to all migration strategies, for example to define code units.

The main method of class *AbstractMigrationStrategy* is method *run* as defined in interface *MigrationStrategy*. It expects as parameter the migrating agent, a reference to the corresponding Kalong interface, and a map (key-value pairs) that contains all migration properties. To define the migration properties, the following keys must be used:

- `MDL_DESTINATION` The corresponding value must be a URL that contains the destination address.
- `MDL_UBICLASSES` The corresponding value must contain an array of String objects that are regular expressions to be matched by fully qualified Java class

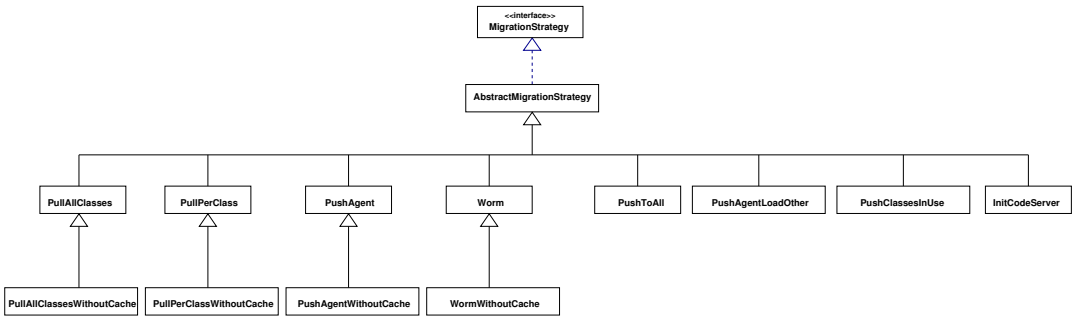


Figure 10.1.: Class diagram for all implemented migration strategies.

names, which should not migrate. This key-value pair is optional. A default value is defined in class *AbstractMigrationStrategy* (*SYSTEMCLASSES*).

- `MDL_USECACHE` If the migration property object contains an entry with this key, the Kalong code cache is activated for the next migration without regard to the default value of the migration strategy.
- `MDL_NOTUSECACHE` If the migration property object contains an entry with this key, the Kalong code cache is disabled for the next migration without regard to the default value of the migration strategy.
- `MDL_OPTIONS` The corresponding value must contain a map, which contains further migration properties used by some migration strategies. This key-value pair is optional.

```
1  package org.taf.mdl.strategies;
2
3  import org.taf.kalong.IKalong;
4  import org.taf.kalong.KalongException;
5  import org.taf.mdl.MDLException;
6  import org.taf.util.ArrayUtils;
7
8  import java.io.Serializable;
9  import java.net.URL;
10 import java.util.Map;
11
12 public abstract class AbstractMigrationStrategy implements org.taf.mdl.MigrationStrategy
13 {
14     protected static String[] SYSTEMCLASSES = new String[]{"java.*", "javax.*", "org.taf.*"};
15     protected static final String ERROR_NO_DESTINATION = "no_destination_address_defined";
16     protected static final String ERROR_NO_PROPERTIES = "no_migration_properties_defined";
17     protected static final String ERROR_CACHEUSAGE_NOT_UNIQUE = "ambitious_cache_
           usage";
```

```
18      protected static final String ERROR_NO_OPTIONS = "no_migration_propertiy_options_
            defined";
19
20      protected boolean useClassCache = true;
21      protected IKalong kalong;
22      protected Serializable agent;
23      protected URL destination = null;
24      protected Map options = null;
25      protected String[] userDefinedUbiClasses = SYSTEMCLASSES;
26
27      public final void run(final Serializable agent, final IKalong kalong, final Map properties)
            throws MDLException
28      {
29          this.agent = agent;
30          this.kalong = kalong;
31
32          checkMigrationProperties( properties );
33
34          try
35          {
36              kalong.startTransaction();
37              kalong.setObjectState(agent);
38
39              if (kalong.getUnits().length == 0)
40              {
41                  defineUnits();
42              }
43
44              migrateAgent();
45
46          } catch (Exception e)
47          {
48              throw new MDLException(e.getMessage());
49
50          } finally
51          {
52              if (kalong.prepare())
53              {
54                  kalong.commit();
55              } else
56              {
57                  kalong.rollback();
58              }
59          }
60      }
61
62      private void checkMigrationProperties( final Map properties ) throws MDLException
63      {
64          if( properties == null )
65          {
66              throw new MDLException( ERROR_NO_PROPERTIES );
67          }
68
69          destination = (URL)properties.get( MDL_DESTINATION );
70          options = (Map)properties.get( MDL_OPTIONS );
71
72          if( properties.containsKey( MDL_UBICLASSES ))
73          {
```

```
74              userDefinedUbiClasses = (String[])properties.get( MDL_UBICLASSES );
75          }
76
77          if( properties.containsKey( MDL_USECACHE) && properties.containsKey(
                MDL_NOTUSECACHE ))
78          {
79              throw new MDLException( ERROR_CACHEUSAGE_NOT_UNIQUE );
80          } else if( properties.containsKey( MDL_USECACHE ))
81          {
82              useClassCache = true;
83          } else if( properties.containsKey( MDL_NOTUSECACHE ))
84          {
85              useClassCache = false;
86          }
87      }
```

Variable *useClassCache* (line 20) decides, whether the class cache should be used or not. By default, a migration strategy uses the cache, but this can be modified by defining this variable **false** or using a migration property as described above. Method *run* first verifies the given migration properties and copies stored values to instance variables. It then starts a transaction, defines the agent object state, and calls method *defineUnits*, if not already done before. Finally, it invokes method *migrateAgent* to start the migration process.

Class *AbstractMigrationStrategy* defines the following two abstract methods that a concrete migration strategy must implement. Their purpose was already explained above.

```
88      protected abstract void defineUnits() throws MDLException;
89
90      protected abstract void migrateAgent() throws MDLException;
```

To simplify the process of unit definition, this class provides two methods to define either all classes as a single unit (method *defineUnitForAllClasses*) or a single unit for each class (method *defineUnitForEachClass*).

```
91      protected final void defineUnitForAllClasses() throws MDLException
92      {
93          try
94          {
95              final String[] agentClasses = filterAgentClasses(kalong.getClassNames());
96              kalong.defineUnit(agentClasses);
97          } catch (KalongException e)
98          {
99              throw new MDLException(e);
100         }
101     }
102
103     protected final void defineUnitForEachClass() throws MDLException
104     {
```

```
105         try
106         {
107             final String[] agentClasses = filterAgentClasses(kalong.getClassNames());
108             for (int i = 0; i < agentClasses.length; i++)
109             {
110                 kalong.defineUnit(new String[]{agentClasses[i]});
111             }
112         } catch (KalongException e)
113         {
114             throw new MDLException(e);
115         }
116     }
117
118     protected final String[] filterAgentClasses(final String[] classNames)
119     {
120         return ArrayUtils.filter(classNames, userDefinedUbiClasses );
121     }
```

Finally, this class defines a method *migrate*, that can be used by concrete migration strategies to start the migration process in a very flexible way. It expects as parameter the destination of the next migration, an array of units and an array of data items that should be transmitted, and a boolean value that indicates, whether the code cache should be used. This method opens a new network transfer and sends the ADB as well as all units and data items, as specified in the parameters.

```
122     protected final void migrate(final URL destination, final int[] units, final String[] dataItems,
            final boolean useCache) throws MDLException
123     {
124         final Object handle;
125
126         if (destination == null)
127         {
128             throw new MDLException( ERROR_NO_DESTINATION );
129         }
130
131         try
132         {
133             handle = kalong.startTransfer(destination);
134             kalong.sendHeader(handle, IKalong.NOOP);
135             kalong.sendADB(handle, useCache);
136
137             if (units != null && units.length != 0)
138             {
139                 kalong.sendUnits(handle, units);
140             }
141
142             kalong.sendState(handle, dataItems);
143
144         } catch (KalongException e)
145         {
146             throw new MDLException(e);
147         }
148     }
149 }
```

10.1.1. Push to Next

The PushToNext migration strategies defines a single unit for all classes of the agent and sends them in one shot to the next destination.

```
1  package org.taf.mdl.strategies;
2
3  import org.taf.kalong.KalongException;
4  import org.taf.mdl.MDLException;
5
6  public class PushToNext extends AbstractMigrationStrategy
7  {
8      protected void defineUnits() throws MDLException
9      {
10         defineUnitForAllClasses();
11     }
12
13     protected void migrateAgent() throws MDLException
14     {
15         final int[] unitIds;
16         final String[] definedDataItems;
17
18         try
19         {
20             unitIds = kalong.getUnits();
21             definedDataItems = kalong.getDefinedDataItems();
22         } catch (KalongException e)
23         {
24             throw new MDLException(e);
25         }
26
27         migrate(destination, unitIds, definedDataItems, useClassCache);
28     }
29 }
```

In line 20 all code units and in line 21 all data items that are currently defined are requested. In line 27 the migration process is started.

Class *PushToNextWithoutCache* extends this class and initializes *useClassCache* with **false**, which deactivates the class cache. However, it is still possible to use keys `MDL_USECACHE` to activate the cache again. For most of the following migration strategies, there exists such a derived class that deactivates the class cache per default, compare Figure 10.1.

10.1.2. Push Agent Class and Load Other Classes

This migration strategy defines a single code unit for each class and only transmits the main agent class to the next destination, while all other classes are loaded dynamically during runtime from the agent's home server.

```
1  package org.taf.mdl.strategies;
2
3  import org.taf.kalong.KalongException;
4  import org.taf.mdl.MDLException;
5
6  public final class PushAgentLoadOther extends AbstractMigrationStrategy
7  {
8      protected void defineUnits() throws MDLException
9      {
10         defineUnitForEachClass();
11     }
12
13     protected void migrateAgent() throws MDLException
14     {
15         try
16         {
17             final String agentClassName = agent.getClass().getName();
18             final int[] units = kalong.getUnitForClassName(agentClassName);
19             final String[] definedDataItems = kalong.getDefinedDataItems();
20
21             migrate( destination, new int[] { units[0] }, definedDataItems, useClassCache );
22
23         } catch( KalongException e )
24         {
25             throw new MDLException( e );
26         }
27     }
28 }
```

In line 17 the class name of the agent's main class is determined and in the following line the corresponding code unit is requested that contains this class. This implementation must be extended, if the agent itself extends other classes or interfaces. In line 21 the migration process is started and the first unit that contains the agent's base class is selected for transmission.

10.1.3. Push Classes in Use

This migration strategy is our equivalence to the Aglets' migration strategy, where all classes are transmitted for which an object exists in the serialized agent (classes in use). It defines a single code unit for each class. The migration process of this strategy is more difficult than in the previous strategies as a search for the smallest set of units that contain the used classes is necessary. In line 33 an array of all used classes is requested. Method *findUnitsToSend* selects the units that contain all these

classes. If more than a single unit qualifies for transmission, the smallest unit is selected.

```
1  package org.taf.mdl.strategies;
2
3  import org.taf.kalong.IKalong;
4  import org.taf.kalong.KalongException;
5  import org.taf.mdl.MDLException;
6  import org.taf.util.ArrayUtils;
7
8  import java.util.HashSet;
9  import java.util.Iterator;
10 import java.util.LinkedList;
11 import java.util.List;
12 import java.util.Set;
13
14 public final class PushClassesInUse extends AbstractMigrationStrategy
15 {
16     public PushClassesInUse()
17     {
18     }
19
20     protected void defineUnits() throws MDLException
21     {
22         defineUnitForEachClass();
23     }
24
25     protected void migrateAgent() throws MDLException
26     {
27         try
28         {
29             /*
30              * Determine the names of all classes for which an object exists in the serialized agent.
31              * Select units, which should be transmitted to achieve this.
32              */
33             final String[] classesInUse = kalong.getClassesInUse();
34             final String[] filtered = ArrayUtils.filter(classesInUse, new String[]{"java.*", "javax.*", "
                       \\[Ljava.*", "org.taf.*"});
35             final int[] unitsToSend = findUnitsToSend(kalong, filtered);
36             final String[] definedDataItems = kalong.getDefinedDataItems();
37
38             migrate( destination, unitsToSend, definedDataItems, useClassCache );
39
40         } catch( KalongException e )
41         {
42             throw new MDLException( e );
43         }
44
45     }
46
47     private int[] findUnitsToSend(final IKalong kalong, final String[] classes) throws
           KalongException
48     {
49         List list = arrayToList(kalong.getUnitForClassName(classes[0]));
50
51         for (int i = 1; i < classes.length; i++)
52         {
```

```
53              list = union(list, arrayToList(kalong.getUnitForClassName(classes[i])));
54          }
55
56          final Set setOfUnits = findSmallest(kalong, list);
57          return ArrayUtils.copy((Integer[]) setOfUnits.toArray(new Integer[0]));
58      }
59
60      private LinkedList union(final List list1, final List list2)
61      {
62          final LinkedList result = new LinkedList();
63
64          try
65          {
66              final Iterator iterList1 = list1.iterator();
67              while (iterList1.hasNext())
68              {
69                  final HashSet set1 = (HashSet) iterList1.next();
70
71                  final Iterator iterList2 = list2.iterator();
72                  while (iterList2.hasNext())
73                  {
74                      final HashSet set2 = (HashSet) iterList2.next();
75                      final Set newSet = new HashSet();
76                      newSet.addAll(set1);
77                      newSet.addAll(set2);
78                      result.add(newSet);
79                  }
80              }
81              return result;
82          } catch (Exception e)
83          {
84              e.printStackTrace();
85              return null;
86          }
87      }
88
89      private List arrayToList(final int[] units)
90      {
91          final List result = new LinkedList();
92
93          for (int i = 0; i < units.length; i++)
94          {
95              final Set set = new HashSet();
96              set.add(new Integer(units[i]));
97              result.add(set);
98          }
99          return result;
100     }
101
102     private Set findSmallest(final IKalong kalong, final List list) throws KalongException
103     {
104         Set minimalSet = null;
105         int minimum = 0;
106         int sizeOfUnits = 0;
107
108         final Iterator iterList = list.iterator();
109         while (iterList.hasNext())
110         {
```

```
111             final Set element = (Set) iterList.next();
112
113             sizeOfUnits = 0;
114             final Iterator iterSet = element.iterator();
115             while (iterSet.hasNext())
116             {
117                 final int unitid = ((Integer) iterSet.next()).intValue();
118                 sizeOfUnits += kalong.getUnitSize(unitid);
119             }
120
121             if (minimum == 0 || minimum > sizeOfUnits)
122             {
123                 minimalSet = element;
124                 minimum = sizeOfUnits;
125             }
126         }
127
128         return minimalSet;
129     }
130 }
```

10.1.4. Initialize Code Server

This migration strategy can be configured to initialize a code server agency without migrating to it. The address of the code server must be stored under key `codeserver.url` in the `MDL_OPTIONS` map of the migration properties.

In method *defineUnits* this migration strategy defines a single unit for each class and deploys all agent's code to the given code server. Method *deployAgentCode* opens a network transfer to this host and transmits all code units. After that, a marker is stored for all these code units to indicate they can be downloaded from this code server in the future. As deploying agent code is part of the definition of code units. Code deployment can only be done once after the agent has been started at the home agency.

The migration process is started by invoking method *migrateAgent*. This method has now to decide, whether the given code server happens to be equal to the next destination (line 54). If this is true, then only the state must be sent, as the network connection to the code server is still open. Otherwise, it starts a new network transfer to the given destination address.

```
1  package org.taf.mdl.strategies;
2
3  import org.taf.kalong.IKalong;
4  import org.taf.kalong.KalongException;
5  import org.taf.mdl.MDLException;
6
7  import java.net.URL;
8
9  public final class InitCodeServer extends AbstractMigrationStrategy
```

```
10 {
11     public static final String CODESERVER = "codeserver.url";
12
13     protected Object handleForCodeserver = null;
14     protected URL codeServer = null;
15
16     public InitCodeServer()
17     {
18         /* Class cache should be activated so that a migration to an agency notices the code server.
               */
19         useClassCache = true;
20     }
21
22     protected void defineUnits() throws MDLException
23     {
24         defineUnitForEachClass();
25
26         /*
27          * Deploy agent's code to the agency and make it to a code server agency.
28          * Check that such an address is defined.
29          */
30         if( options == null )
31         {
32             throw new MDLException( ERROR_NO_OPTIONS );
33         }
34
35         codeServer = (URL) options.get( CODESERVER );
36         if( codeServer == null )
37         {
38             throw new MDLException( "no_codeserver_address_defined" );
39         }
40
41         try
42         {
43             deployAgentCode( codeServer, useClassCache );
44         } catch( Exception e )
45         {
46             throw new MDLException( e );
47         }
48     }
49
50     protected void migrateAgent() throws MDLException
51     {
52         try
53         {
54             if( codeServer != null && codeServer.equals( destination ) && handleForCodeserver !=
                       null )
55             {
56                 /* connection to code server is already open, reuse it */
57                 kalong.sendState( handleForCodeserver, null );
58
59             } else
60             {
61                 migrate( destination, kalong.getUnits(), null, useClassCache );
62             }
63
64         } catch( KalongException e )
65         {
```

```
66                  throw new MDLException( e );
67              }
68
69          }
70
71          private void deployAgentCode( final URL codeserver, final boolean useCache ) throws
                KalongException
72          {
73              int[] allUnits = kalong.getUnits();
74
75              final Object handle = kalong.startTransfer(codeserver);
76              handleForCodeserver = handle;
77
78              kalong.sendHeader(handle, IKalong.START_CODESERVER);
79              kalong.sendADB(handle, useCache );
80              kalong.sendUnits(handle, allUnits);
81
82              /* store information about the new code server for each unit */
83              allUnits = kalong.getUnits();
84              for( int i=0; i<allUnits.length; i++ )
85              {
86                  kalong.addCodeBases(i, new URL[] {codeserver} );
87              }
88          }
89      }
```

10.1.5. Push to All

The next migration strategy is an extension of the last one, as it sends the code not only to a single but to all hosts that the agent will visit while executing its itinerary.

The itinerary must be given as an array of URLs stored in the options of the migration properties. In line 42 it is checked that the given destination address is member of this array. Method *deployAgentCode* deploys all code units to all these hosts. Again, as in the last migration strategy, this process of deploying an agent's code can only be done once in the agent's life-time.

The migration process is defined in method *migrateAgent*. It must decide, whether it is the first migration leaving the home agency. In this case, an already open network connection can be reused. Otherwise, a new network transfer must be opened.

```
1   package org.taf.mdl.strategies;
2
3   import org.taf.kalong.IKalong;
4   import org.taf.kalong.KalongException;
5   import org.taf.mdl.MDLException;
6   import org.taf.util.ArrayUtils;
7
8   import java.net.URL;
9
10  public final class PushToAll extends AbstractMigrationStrategy
11  {
```

```
12      public static final String ALL_DESTINATIONS = "all.destinations";
13      protected static final String ERROR_NO_ALLDEST = "migration_property_options_does_not_
            contain_key_\"" + ALL_DESTINATIONS + "\"";
14
15      protected Object handleForFirstDestination = null;
16
17      public PushToAll()
18      {
19      }
20
21      protected void defineUnits() throws MDLException
22      {
23          defineUnitForAllClasses();
24
25          /*
26           * The addresses of all destinations is stored in the migration properties.
27           * Check that the given migration destination is element of this array.
28           */
29          if( options == null )
30          {
31              throw new MDLException( ERROR_NO_OPTIONS );
32          }
33
34          final URL[] allDestinations = (URL[]) options.get( ALL_DESTINATIONS );
35
36          if( allDestinations == null || allDestinations.length == 0 )
37          {
38              throw new MDLException( ERROR_NO_ALLDEST );
39          } else if( destination == null )
40          {
41              throw new MDLException( ERROR_NO_DESTINATION );
42          } if( ! ArrayUtils.hasElement( allDestinations, destination ))
43          {
44              throw new MDLException( "given_destination_address_\"" + destination + "\"_is_not_
                    element_of_the_list_of_all_destinations" );
45          }
46
47          try
48          {
49              deployAgentCode( allDestinations, destination, useClassCache );
50          } catch( Exception e )
51          {
52              throw new MDLException( e );
53          }
54      }
55
56      protected void migrateAgent() throws MDLException
57      {
58          Object handle = null;
59          try
60          {
61              /*
62               * If the migration is directed to any of the agencies to which code
63               * was shipped before, reuse this transfer. Otherwise (means, this is
64               * not the first migration with this strategy), start a new transfer.
65               */
66              if( handleForFirstDestination == null )
67              {
```

```
68                  handle = kalong.startTransfer( destination );
69              } else
70              {
71                  handle = handleForFirstDestination;
72              }
73
74              kalong.sendState( handle, kalong.getDefinedDataItems() );
75
76          } catch( KalongException e )
77          {
78              throw new MDLException( e );
79          }
80
81      }
82
83      private void deployAgentCode( final URL[] allDestinations, final URL destination, final
            boolean useCache ) throws Exception
84      {
85          final int[] allUnits = kalong.getUnits();
86
87          for (int i = 0; i < allDestinations.length; i++)
88          {
89              final Object handle = kalong.startTransfer(allDestinations[i]);
90
91              if( destination.equals( allDestinations[i] ))
92              {
93                  handleForFirstDestination = handle;
94              }
95
96              kalong.sendHeader(handle, IKalong.START_CODESERVER);
97              kalong.sendADB(handle, useCache );
98              kalong.sendUnits(handle, allUnits);
99          }
100     }
101 }
```

10.1.6. Pull Per Class

This migration strategy does not transmit any code units but only the migrating agent's object state during the migration process. It defines a single code unit for each class. In line 14 the migration process is started and the two **null** values indicate that no code units and no data items shall be transmitted per default.

```
1  package org.taf.mdl.strategies;
2
3  import org.taf.mdl.MDLException;
4
5  public class PullPerClass extends AbstractMigrationStrategy
6  {
7      protected void defineUnits() throws MDLException
8      {
9          defineUnitForEachClass();
10     }
```

```
11
12      protected void migrateAgent() throws MDLException
13      {
14          migrate( destination, null, null, useClassCache );
15      }
16  }
```

10.1.7. Pull All Classes

This migration strategy only differs from the last one by defining a single code unit for all classes.

```
1   package org.taf.mdl.strategies;
2
3   import org.taf.mdl.MDLException;
4
5   public class PullAllClasses extends AbstractMigrationStrategy
6   {
7
8       protected void defineUnits() throws MDLException
9       {
10          defineUnitForAllClasses();
11      }
12
13      protected void migrateAgent() throws MDLException
14      {
15          migrate(destination, null, null, useClassCache );
16      }
17  }
```

10.1.8. Worm

The last migration strategy works like a worm that roams the network and initializes each agency that it visits to become a code server. It does not transmit any code along with its state transmission, but always loads necessary classes from the last agency it has been visited before. It defines a single code unit for all classes.

The migration process has to store information about the new code server for all code units and then migrates the agent without code and external data items.

```
1   package org.taf.mdl.strategies;
2
3   import org.taf.kalong.IKalong;
4   import org.taf.kalong.KalongException;
5   import org.taf.mdl.MDLException;
6
7   import java.net.URL;
8
```

```
9  public class Worm extends AbstractMigrationStrategy
10 {
11     protected void defineUnits() throws MDLException
12     {
13         defineUnitForAllClasses();
14     }
15
16     protected void migrateAgent() throws MDLException
17     {
18         try
19         {
20             /*
21              * Initialize code server, mark all units so that they are copied here.
22              */
23             final URL[] thisAgency = kalong.getURLs();
24             final int[] allUnits = kalong.getUnits();
25             for (int i = 0; i < allUnits.length; i++)
26             {
27                 kalong.addCodeBases(allUnits[i], thisAgency);
28             }
29
30             migrate( destination, null, null, useClassCache );
31
32         } catch( KalongException e )
33         {
34             throw new MDLException( e );
35         }
36     }
37 }
```

10.2. Adapting the Migration Strategy

In the last section, we introduced several migration strategies, where the user cannot influence the migration process in detail. By selecting the migration strategy it is already defined which code units shall be transmitted. None of these migration strategies selects elements of the external state to migrate. They always transmit the whole agent object state and all defined data items. At this point of our discussion, we will describe techniques, that enable the user to influence the migration process in more detail.

10.2.1. Generic Migration

In Section 10.1.3 we presented the migration strategy that sends all classes of an agent that are currently *in-use* to the next destination, whereas all other classes must be loaded dynamically from an agent's code server. This migration strategy is comparable to the one that is implemented in the Aglets system. It is correct that classes of objects that are currently accessed must be available at the remote

destination, compare Section 6.1 on page 83. However, as we also have shown there, it is not necessarily the case that all theses object are accessed at the destination.

If it is known that on the next destination agency only specific data items and their corresponding classes are used, then the following pattern can be used: Put all data that will not be used at the destination agency in a data item of the external state and do not transmit this data item along object state migration. In other words, select only those data items for migration that must be available at the destination together with their corresponding classes. A new migration strategy is necessary, where the programmer can define, in a fine-grained way, which data items and/or classes should be transmitted. Such a strategy is implemented in the following class:

```
1  package org.taf.mdl.strategies;
2
3  import org.taf.kalong.IKalong;
4  import org.taf.mdl.MDLException;
5  import org.taf.mdl.MigrationStrategy;
6  import org.taf.util.ArrayUtils;
7
8  import java.io.Serializable;
9  import java.net.URL;
10 import java.util.Map;
11
12 public final class Generic extends AbstractMigrationStrategy
13 {
14     public static final String UNITS = "generic.units";
15     public static final String DATAITEMS = "generic.dataitems";
16
17     public Generic()
18     {
19     }
20
21     protected void defineUnits() throws MDLException
22     {
23         defineUnitForEachClass();
24     }
25
26     protected void migrateAgent() throws MDLException
27     {
28         if( options == null )
29         {
30             throw new MDLException( ERROR_NO_OPTIONS );
31         }
32
33         final int[] unitsToSend = (int[])options.get( UNITS );
34         final String[] dataItemsToSend = (String[])options.get( DATAITEMS );
35         migrate( destination, unitsToSend, dataItemsToSend, useClassCache );
36     }
37 }
```

In the options map of the migration properties, two entries specify which data item and code units should migrate (lines 14–15). The entry with key `UNITS` must

contain an array of integer values, which are unit identifiers. The entry with key `DATAITEMS` must contain an array of Strings, which are the names of the data items to transmit. This migration strategy defines a single code unit for each class. In line 35 the migration process is started with the user-defined selection of code units and data items.

10.2.2. Loading Classes and Data Items in Advance

As already described in the last chapters, Kalong provides the possibility to fetch data items of the external state that are currently defined at the agent's home or mirror agency. We did not mention, that the same technique is possible for classes resp. code units too.

Until now, we always stressed that code classes that are not available at the current agency, can be downloaded from the agent's home, code server, or mirror agency automatically on demand. The MDL component provides a new technique, so that the programmer can start the process of code downloading manually, as soon as it is clear that a specific class will be used in future. This technique, called *code prefetching*, provides the advantage that code can be downloaded in parallel with agent execution, a feature that may improve the performance of mobile agents considerably.

Consider the following source code:

```
1  public class SampleAgent implements Runnable
2  {
3    public void run()
4    {
5      try
6      {
7        IMigrationContext migrationContext = Context.getContext( "migration" );
8
9        // ...
10
11       if( /* ... */ )
12       {
13           // assume that it is known now that class A is used in future
14           migrationContext.loadClassNonBlocking( "A" );
15
16           // ...
17
18           A a = new A();
19
20           // ...
21
22       }
```

```
23      } catch( Exception e )
24      {
25        e.printStackTrace();
26      }
27    }
28  }
```

We assume that in line 14 it is already known that class A will be used in future. In this line, the process of asynchronous class loading is started. The method invocation returns immediately and a new thread has been started that processes class loading in parallel. Later, when agent execution reaches line 18, the thread might have terminated already, so that there is no delay for loading class A at this point.

Data items can be loaded asynchronously too, using *loadDataItemNonBlocking* instead of method *loadDataItem* that was introduced in the last chapters.

10.3. Outlook to More Sophisticated Migration Strategies

In this section we will present some ideas for more sophisticated migration strategies. Although it might be sufficient for most application domains to delegate the decision about the migration strategy to the agent programmer, it is undoubtedly more convenient to let Kalong determine the migration strategy itself. We call such a migration strategy that decides in an autonomous fashion on the next migration strategy, an *automated migration strategy*. In some situations, for example, if the agent roams the Internet without a fixed itinerary but decides at each agency to which host the next migration should be directed to, there is no alternative to an automated migration strategy, if migration should be optimized, as the user has no knowledge about the route that the agent will take.

As we have already shown in Section 3.2 on page 26 the performance of agent migration depends on several factors, as for example network quality, execution probability of each code unit, the size of each code unit, etc. We now have to face two problems. First, all these parameters must be determined and we will present some techniques that we have developed to gather information about the network quality later. Currently we are working towards a technique to analyze an agent's code to determine the execution probability of each code class by static or dynamic code analysis. Second, we have to find algorithms to decide, on the basis of these parameters, how the next migrations should be processed.

Let's start with the second step, because it turns out to be simpler. Our goal is to decide which classes should be pushed from the sender to the destination agency and which classes should be loaded from the agent's home server later. We first disregard all other Kalong features, as for example code servers and mirror servers,

which complicate this decision further. As each class transmission is independent to any other class, we must simply compare the migration time for each single class.

Let the mobile agent consist of u code units, where each code unit comprises of exactly a single class. The size of code unit k equals $B_{\mathrm{c}}^{k}, k = 1, \ldots, u$. The network is modeled using delay δ and throughput τ, each are assumed to be available for all pairs of network nodes. The cost to transmit code unit k from the sender agency L_{i} to the destination agency $L_{\mathrm{i+1}}$ equals

$$T_{push}^{k} = \frac{B_{\mathrm{c}}^{k}}{\tau(L_{\mathrm{i}}, L_{\mathrm{i+1}})}$$

We do not factor in the agent's data and state size $B_{\mathrm{d}} + B_{\mathrm{s}}$ and the network delay $\delta(L_{\mathrm{i}}, L_{\mathrm{i+1}})$, as these costs arise in any case. The alternative of pushing the code unit is not to transmit it from the sender agency, but to impose on the destination agency to load missing code units on demand from the agent's home server L_0. We have to consider the execution probability $P_{L_{\mathrm{i+1}}}^{k}$ that code unit k is really needed at destination $L_{\mathrm{i+1}}$, so the cost amounts to

$$T_{pull}^{k} = P_{L_{\mathrm{i+1}}}^{k} \left(\delta(L_{\mathrm{i+1}}, L_0) + \frac{(B_{\mathrm{c}}^{k} + B_{\mathrm{r}})}{\tau(L_{\mathrm{i+1}}, L_0)} \right)$$

The cost to request a code unit equals B_{r}. Here, we have to include the delay to open a network connection, because each code unit is loaded using a new connection. We are aware of the fact that both equations base on the simple mathematical model of network load and transmission time developed in Section 3.2 on page 26, which has to be improved and refined to serve as a real forecast instrument.

For each code unit k we compare both costs and choose the technique that has lower cost. Figure 10.2 shows the source code of a method that implements this decision process. Parameter *code* contains the size of all code units, and *exProb* the execution probability for all code units on agency $L_{\mathrm{i+1}}$. The network parameters are given by *throughput1* for the connection between L_{i} and $L_{\mathrm{i+1}}$ and *throughput2* for the connection between $L_{\mathrm{i+1}}$ and L_0. Network delay is given only for the home connection, and parameter *request* contains the cost for a code request.

For example, an agent consists of two code units, where $B_{\mathrm{c}}^{1} = 20$ KB and $B_{\mathrm{c}}^{2} = 10$ KB. The execution probabilities are $P_{L_{\mathrm{i+1}}}^{1} = 0.8$ resp. $P_{L_{\mathrm{i+1}}}^{2} = 0.2$. The network connection to the destination agency has throughput $\tau(L_{\mathrm{i}}, L_{\mathrm{i+1}}) = 800$ Kb/sec and the one between destination agency and home server has throughput $\tau(L_{\mathrm{i+1}}, L_0) = 240$ Kb/sec. Delay equals 15 ms and the cost of a code unit request is 100 byte. It turns out that it does cost 200 ms to push the first code unit, whereas it takes 548 ms to load it later. Therefore, the first code unit is pushed to the destination agency. For the second code unit, cost for pushing equals 100 ms, whereas loading it later on demand does only cost 70 ms – so this code unit is not pushed. If we decrease

```
1  public Set findClasses( int[] code, float[] exProb, float throughput1,
2                          float throughput2, float delay2, int request )
3  {
4      float pushCase, pullCase;
5      Set result = new HashSet();
6
7      for( int i=0; i<code.length; i++ )
8      {
9          pushCase = ((float)code[i] / throughput1 );
10         pullCase = exProb[i] * ( delay2 + ((float)code[i] + (float)request ) /
               throughput2 );
11
12         if( pushCase < pullCase )
13         {
14             result.add( new Integer(i) );
15         }
16     }
17
18     return result;
19 }
```

Figure 10.2.: Method to determine the classes to push to the next destination.

throughput between destination agency and home agency, the result changes in a way that now all code units should be pushed, because it is more costly to load missing units later.

The decision process is more difficult, if not only the immediate next but more migrations should be planned. Not only the network parameters must be known for all agencies to be visited, but also the execution probability for all code units at all destinations must be known. At this point, we might regard this as a limitation of this kind of automated migration strategies, as it can be extremely expensive to bring together network information about all next agencies at the current agency.

In principle, even the decision to initialize a code server or mirror agency can be done using the same technique as described above. The question, whether a code server should be initialized at the current agency can be answered by comparing transmission time for the next migrations for the case that all code must be loaded from the home server resp. the code server. If migration time can be reduced, a code server should be initialized.

Currently, we are not able to determine all necessary information about network quality and execution probability. Therefore, we were not able to verify our technique in a real-world environment. However, we are currently working on extensions of Kalong, for which we now present the current state of implementation.

Network Analysis We have developed a tool to monitor network performance and integrated it in the Tracy mobile agent system. On each agency, this component gathers information about network quality by testing network connections to neighbor agencies. Two agencies are neighbors, if they are member of the same *domain* which is never more than a sub-network, as defined by the Tracy domain manager concept, see Section A.5 on page 277 for more information. Time measurements are done using *ping* messages sent periodically between two agencies, from which network throughput and latency are deduced. The monitor component provides an application programming interface so that other components or agents can use the results. In a second step, monitor components located at different agencies communicate to exchange network information using mobile agents. Using this technique it is also possible to use information about throughput and latency of a remote agency, which is necessary to implement such automated migration strategies as described above. Steffen Schreiber implemented this network monitoring tool as part of this diploma thesis [Schreiber, 2002].

Agent Profiling To determine the execution probability of classes in a Java-based mobile agent, we plan to use *profiling* techniques known from the area of compiler construction, which in general are used to predict the probability of executing specific code portions.

A first approach to determine a profile is to instrument the agent's source code and to count how often a basic block is executed. This type of profiling is called *dynamic*, because the agent must be executed to obtain profile information. The advantage of dynamic profiling is that it provides very accurate information. However, as the profile depends on the agent instance and not only on the type of agent, different input data, for example a different user task, might lead to a completely different profile. Therefore, such a profile is undoubtedly valuable, if a single agent is reused many times for comparable tasks, but it is questionable, whether dynamic profile information can be transferred to other instances, even if they are of the same agent type. Another drawback of dynamic profiling is that it increases not only the code by adding new statements for counting, but also the data, as the information must be part of the agent's state and carried through the network.

Another approach works *statically* and uses source code analysis to estimate profiles. It uses techniques as for example branch prediction or analysis of method invocation frequencies to forecast, how often specific pieces of code will be used. The advantage is that it is done before executing the agent and, therefore, will neither increase the code nor the agent's data. Static profiling will never be as accurate as dynamic profiling, but it works for all agents of the same type and can, therefore, be used several times.

The result of profiling is in both cases information, about the execution probability of classes.

Class Splitting As an extension of the last technique we consider the idea of *class splitting* to reduce network traffic for mobile agents. The problem with agent profiling, as presented above, is a code granularity that so far is always the complete class. It can be assumed that invocation frequency is not the same for all methods of the same class. Therefore, it makes sense to increase the granularity of agent profiling to provide information on the level of methods and to split a class into two or more new classes and distribute methods according to their execution probability. Groups of methods that are used with the same probability, because they call each other, should be member of the same class, while other methods with lower execution probability should form another class. If a method is called that is not implemented in the main class, then another class is loaded that contains the code. The effect of class splitting is that the resulting classes, and especially the main class, are smaller than the single original class, which will in turn reduce network transmission time.

Chris Fensch implemented a software component for class splitting as part of his diploma thesis [Fensch, 2001]. It provides a simple interface, where the user can define to split a class into n other classes and specify, which class should contain which methods. Classes are split on the level of Java byte code. The result of splitting code is completely transparent to the programmer. This set of new classes

can be used as if it were still only a single class, as all fragments are linked together.

Figure 10.3 shows an example of the class splitting technique. The left figure shows the original agent. Using profiling technique, we found that method *startAgent*, which is in Tracy only called once, when the agent is started, is never used on visited agencies and, therefore, should not be part of the main agent. The right figure shows the result of two classes *AnAgent*, which only contains a stub for method *startAgent* and class *Split*, which contains the code for this method. Method *startAgent* of class *AnAgent* creates an instance of class *Split*, if not already done, and forwards the method invocation to this object. The figure also shows how the splitted class can access private variables of the original agent using an auxiliary method that was introduced by the splitting algorithm.

The idea of class splitting was proposed in the area of Java applets already [Krintz et al., 1999]. We are currently working on experiments to verify the effect of class splitting for real-world agents.

```
1  package MyAgent;
2
3  class AnAgent extends MobileAgent
4  {
5    private Vector route;
6
7    // ...
8
9    public void startAgent()
10   {
11     // ...
12     route = new Vector();
13     // ...
14   }
15 }
```

(a) Original agent.

```
1  package MyAgents;
2
3  class AnAgent extens MobileAgent
4  {
5    private Vector route;
6    private transient Split I1;
7
8    public void startAgent()
9    {
10     if( I1 == null )
11     {
12       I1 = new Split( this );
13     }
14     I1.startAgent();
15   }
16
17   void access$123(Vector x)
18   {
19     route = x;
20   }
21 }
```

```
1  package MyAgents;
2
3  class Split
4  {
5    private AnAgent this$0;
6
7    Split(AnAgent x)
8    {
9      this$0 = x;
10   }
11
12   public void startAgent()
13   {
14     // ...
15     this$0.access$123(new Vector());
16     // ...
17   }
18 }
```

(b) The splitted agent consists of two classes.

Figure 10.3.: Example for the class splitting technique. We picture Java source code, although class splitting works on the level of Java byte code.

11. Practical Performance Evaluation

In this chapter we examine several measurements to demonstrate the performance of our new migration component Kalong. Our goal is to give some first impression on how fast (or slow) migration can be in Java based agent systems. Additionally, we will give some impression on how different parameters influence the migration performance, as for example code size, network quality, code compression, and security enhancements. Finally, we will show the effect of the new features of Kalong, which provide the possibility to send data items back to the agent's home agency, to load code from a code server instead from the home agency, or the effect of mirror agencies.

Our performance experiments must be seen as a first step towards a comprehensive performance analysis of the migration process of mobile agents in general. Due to some restrictions in the availability of enough network nodes and different network qualities, we had to limit our experiments in the following aspects:

- We only measure the Kalong migration component as part of a very simple mobile agent system, which is not equal to the Tracy system. We do not compare our results to other mobile agent systems, as some of these are not executable with the newest (and certainly fastest) Java virtual machine.
- There is no benchmark suite available, for example in form of several mobile agents, which perform specific migrations. Therefore, we developed our own mobile agents tailored to show the specific advantages of Kalong.
- We only measure the performance of mobile agents and do not compare it to the client-server case.
- We only measure migration times and not the performance of a whole mobile agent system. The agents that we used in the experiments do not produce load on each visited agency.
- We only measure the time for a single mobile agent. We have no experiences how the performance of Kalong will be for a higher number of agents that migrate in parallel.
- We only have a very small number of network nodes available, especially in the wide-area network. It is extremely interesting, how migration times increase in real-world applications.

11.1. Related Work

11.1.1. Performance Evaluation of Existing Mobile Agent Systems

As far as we know, only two systems have ever been explored concerning migration performance. First, Gray [1997b] proposes in his thesis on the AgentTCL system some performance evaluations. AgentTCL is based on the script language TCL. The AgentTCL system provides some basic functions for a flexible and secure mobile agent system. His results for migration times show high delays due to the very slow script interpreter and a migration protocol overhead. Second, the Tacoma system was evaluated by Johansen et al. [1997]. Tacoma is also not a Java-based mobile agent system. The authors give values for the migration time of one agent from one server to a remote one including time for serializing and deserializing, creating and initiating, as well as sending an acknowledge message. As far as we know, no Java based agent system has ever been explored concerning performance issues.

11.1.2. Performance Comparison of Mobile Agent Systems

Some work has been done to compare existing mobile agent systems. Dikaiakos and Samaras [2000] define some *micro-benchmarks* to assess a mobile agent system, e.g. one to capture the overhead of local agent creation, or one to capture the overhead of point-to-point messaging. Although the authors define a micro-benchmark (called ROAM) to capture the "agent-traveling overhead" [Dikaiakos and Samaras, 2000, p. 9], they do not provide any values for this rather interesting benchmark for *mobile* agent systems. Their paper also suffers of some fundamental methodological flaws, e.g. they do not publish any factor information: the reader does not know which hardware configuration, which type of network, or which software version was used. Silva et al. [2000] compare eight mobile agent systems using twelve experiments. Their results show the influence of several factors, e.g. the number of agent servers to visit on one tour, the influence of the agent's size and the influence of class caching, on the performance of mobile agents. In our opinion, different mobile agent systems cannot be compared without taking some fundamental design issues of each system into account. Unfortunately, the authors do not consider that each system has implemented different security strategies, different migration and transmission strategies, etc.

11.2. Methodology

11.2.1. Experiments and Measurements

In sum we conducted eight different experiments, where in each experiment the migration time for a specific mobile agent in a certain environment is measured. Each

experiment consists of several measurements, where in each measurement the same agent is started several times. Agents used in different measurements vary for example in code size or in the number of servers to be visited.

To conduct an experiment we developed a simple mobile agent system that is defined in package *org.taf.simpleagency*. The main function of this agency is to start agents, to measure migration time for each agent, to compute statistical information (mean value and confidence interval) for a measurement, and finally to generate a file that contains all the results of an experiment.

In each experiment we distinguish two roles for the involved computers. The computer on which all agents are started is called *master*, all other computers that are only visited by the agents are called *clients*. All experiments are described using a single XML file. For each experiment it contains a node that defines the master configuration and another node that defines the client configuration. The following excerpt shows the beginning of the description of experiment no. 1 (M01) for the master.

```
1  <node name="M01_master">
2    <map>
3      <entry key="network.protocol.class.1" value="org.taf.network.tcp.TCPEngine"/>
4      <entry key="network.protocol.port.1" value="5555"/>
5      <entry key="keystore.url" value="file:///home/mit/ips/braunpet/tracy2/migration/keystore"/>
6      <entry key="keystore.alias" value="ag1"/>
7      <entry key="keystore.password" value="*******"/>
8      <entry key="filename" value="M01_100l"/>
9
10     <entry key="agent.0.name" value="0"/>
11     <entry key="agent.0.url" value="file:///home/mit/ips/braunpet/tracy2/migration/examples"/>
12     <entry key="agent.0.class" value="agent.M01_0"/>
13     <entry key="agent.0.home" value="tcp://ipc047.inf.uni-jena.de:5555"/>
14     <entry key="agent.0.route" value="tcp://ipc033.inf.uni-jena.de:5566"/>
15     <entry key="agent.0.strat" value="pushnocache"/>
16     <entry key="agent.0.init" value=""/>
17     <entry key="agent.0.buildup" value="5"/>
18     <entry key="agent.0.rep" value="1000"/>
19
20     <entry key="agent.1.name" value="2500"/>
21     <entry key="agent.1.url" value="file:///home/mit/ips/braunpet/tracy2/migration/examples"/>
22     <entry key="agent.1.class" value="agent.M01_2500"/>
23     <entry key="agent.1.home" value="tcp://ipc047.inf.uni-jena.de:5555"/>
24     <entry key="agent.1.route" value="tcp://ipc033.inf.uni-jena.de:5566"/>
25     <entry key="agent.1.strat" value="pushnocache"/>
26     <entry key="agent.1.init" value=""/>
27     <entry key="agent.1.buildup" value="5"/>
28     <entry key="agent.1.rep" value="1000"/>
29
30   </map>
31 </node>
```

In lines 3-8 the master agency is configured, for example which network protocol should be started on which port. In lines 10-18 the first measurement is described.

Most important is line 12 where the name of the class is defined, line 14 where the itinerary for this agent is defined, line 15 where the migration strategy to be used is defined, and line 18 where the number of repetitions is specified. Lines 20-28 contain the description of the second measurement. The following XML fragment shows the definition of a client computer.

```
1 <node name="M01_client">
2   <map>
3     <entry key="network.protocol.class.1" value="org.taf.network.tcp.TCPEngine"/>
4     <entry key="network.protocol.port.1" value="5566"/>
5     <entry key="keystore.url" value="file:///home/mit/ips/braunpet/tracy2/migration/keystore"/>
6     <entry key="keystore.alias" value="ag2"/>
7     <entry key="keystore.password" value="*******"/>
8   </map>
9 </node>
```

For each experiment the Java virtual machine must be restarted. When the agency is started it is parameterized with the name of the experiment to start (for example `M01_master`). It then sequentially starts all the measurements. As already said, the only information we are interested in is the time an agent needs for a migration.

To measure the time for a single migration of a mobile agent, we have to consider the period of time from the initiation of the migration process (`go`-statement) to the point of time where the agent is restarted at the destination server. Due to the lack of a global time in a distributed system, we cannot simply compare time stamps originating from different computer systems. Therefore, we always consider at least two migrations: the first one to the destination server and the second one back to the origin – we call this a *ping-pong migration*. Therefore, printed times are never those for a single migration but always for a complete round-trip, which consists in most cases of only two computers and in some cases of up to 7 computers. As a consequence, the measured migration times do not only consist of the pure network transmission time, but also the time for serializing the agent at the sender agency and deserializing it at the receiver agency for each migration. Additionally we consider time to link agent's code, which involves verifying and preparing class code. The process of serializing an agent takes in all our measurements less than 2 ms and the process of deserializing the agent's state and the linking agent's code takes on average between 1 and 5 ms and is linear with respect to state size resp. class size.

Each agent migration is repeated between 200 and 1 000 times and we only report mean values and the 95% significance interval. The top 5% of the values were dropped, because we want to disregard times messed through the Java garbage collector task.[1] To illustrate our results we always used line-charts, although in some experiments

[1] The Java garbage collector is started whenever there is not enough memory to create new objects. The process to free memory takes between 300 and 900 ms in our experiments.

box-charts would have been the correct diagramming technique, because intermediate values cannot be interpolated. However, in our opinion, line-charts make the results we want to show more obvious to the reader.

11.2.2. Programming Agents for the Measurements

The common behavior of agents used in the experiments is defined in class *BaseAgent* in package *examples.agent*. There it is defined that an agent executes the itinerary given in the configuration file and finally migrates back to its home agency.

In general there exists a single agent class for each measurement. This class extends class *BaseAgent* and defines special functions as necessary in the concrete measurement, for example to send data items back to the agent's home agency.

In some cases it is necessary to artificially increase the size of the agent's code, for example to show how migration time depends on code size. We use static String objects for this purpose, which become part of the agent's code and are not part of the agent's object state.

11.2.3. Test Environment

For the measurements we used seven computers placed at the University of Jena, one computer placed at the University of Weimar (Germany)[2], one computer placed at the Fraunhofer Society Darmstadt (Germany)[3], and one at the University of Irvine (California, USA). More information about the used computers can be found in Table 11.1. All computers use the latest version of the Java virtual machine (build 1.4.1_01-b01). The Java virtual machine was initialized to use an initial heap size of 80 MB and a maximal heap size of 200 MB. The stack size is set to 512 KB. All computers were fully dedicated during the experiments and all computer systems have been used under the same conditions.

For most measurements we used the local area network in our department at the University of Jena, which is an Fast-Ethernet network with a bandwidth of 100 Mb/sec where computers are connected via a single router. Some measurements were done using a fully dedicated Ethernet network with a bandwidth of 100 Mb/sec resp. 10 Mb/sec connected via a switch.

Measurements of migration times to the computers in Weimar, Darmstadt, and Irvine were done using our standard Internet connection, which is a 155 Mb/sec uplink to the German GigaBit Research Network (G-Win), which itself has a theoretical bandwidth of 2.5 Gb/sec. The university of Weimar is also connected to G-Win using also a 155 Mb/sec. The quality of the network connections at Darmstadt and Irvine could not be determined.

[2]The city of Weimar is located about 20 km away from Jena.
[3]The city of Darmstadt is located about 300 km away from Jena.

Name	Location	Processor	MHz	RAM	OS
ipc047	Jena	Athlon	900	512	Linux 2.4.18
ipc026	Jena	Athlon	1400	512	Linux 2.4.18
ipc030	Jena	Athlon	900	256	Linux 2.4.18
ipc031	Jena	Athlon	900	256	Linux 2.4.18
ipc032	Jena	Athlon	900	256	Linux 2.4.18
ipc033	Jena	Athlon	800	256	Linux 2.4.18
ipc051	Jena	Athlon XP 1600+	1400	256	Linux 2.4.18
gonzo	Weimar	Pentium 3	800	1024	Linux 2.4.0
semoaext	Darmstadt	Pentium 2	450	512	SunOS 5.8
waylander	Irvine (USA)	Pentium 4	1700	896	Linux 2.4.18

Table 11.1.: Some parameters of the computer systems used in our experiments.

11.3. Results of the Basic Experiments

11.3.1. Transmission Time with regard to Code Size and Network Quality

In the first experiment we examined the time for a ping-pong migration of a single agent with different sizes in different networks. The agent is created on ipc047 and has to migrate back and forth to a single other agency. We compare the migration time for the following code sizes: 1685, 4185, 6685, 11 685, 22 685, and 51 685 byte. The agent's state is negligible in this experiment, as it consists of less then 100 byte. The agent is transmitted using the PushToNext strategy without enabling the code cache. All migrations were repeated 1000 times.

Figure 11.1 shows the migration times for all high-bandwidth connections. The destination agency was started on ipc033. The graph also shows the result of a measurement, where the sender agency as well as the receiver agency were located on the same computer (ipc047) and the agent migrates using the local loop without using the network.

The best migration performance was achieved using the 100 Mb/sec network via a switch, where the smallest agent (1685 byte) only needs 23 ms for a single migration. The migration time only increases slightly up to 34 ms for the largest agent (51 685 byte). Migration using the 100 Mb/sec network via a router is only few milliseconds slower: 25 ms for the smallest and 35.5 ms for the largest agent. The measurement using the internal network loop of the operating system was surprisingly slower than both measurements using a 100 Mb/sec network. Here, a single migration has costs of 28 ms for the smallest and 38 ms for the largest agent. The only reason we have found so far is that this increase is due to the higher compu-

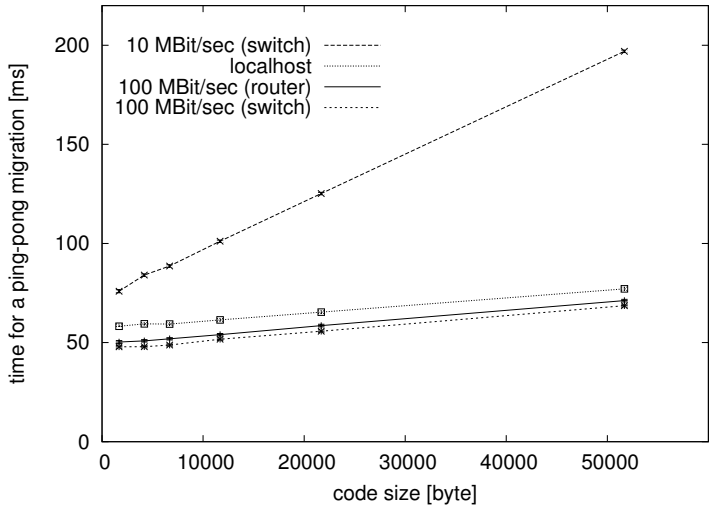


Figure 11.1.: Time for a ping-pong migration between computers ipc047 and ipc033 using different high-bandwidth networks. The *localhost* measurement was done on computer ipc047.

tational load of executing two agencies on a single computer in parallel. Migrating agents using a 10 Mb/sec network is noticeably slower than the previous network types. The smallest agent needs about 37 ms for a single migration and the largest agent needs about 98 ms.

As can be seen from the graph, migration time is linear with the code size of the agent. As can be seen from the 95% confidence intervals in Figure 11.1, measured migration times are not significantly different when transmitting small agents (less than 11 685 byte) using fast networks. This gives hints for the construction of an optimization strategy: it is not always worth to reduce a 10 KB agent to a 5 KB one, because the difference cannot be measured in general in fast networks. Using the 10 Mb/sec network, all results are significantly different.

Comparing our results to the theoretical possible migration times, we found that migration is about 8 times as high as possible using a 100 Mb/sec network. In a 10 Mb/sec network measured migration times are only twice as high as possible. In other words, we have achieved a network throughput of about 13 Mb/sec in the 100 Mb/sec network and a throughput of almost 5 Mb/sec in the 10 Mb/sec network in our measurements. The reason for this quite slow values can be found in a fixed overhead of the Java programming language resp. the Java virtual machine, which is known to have slow performance for network operations as compared to native code implementations.

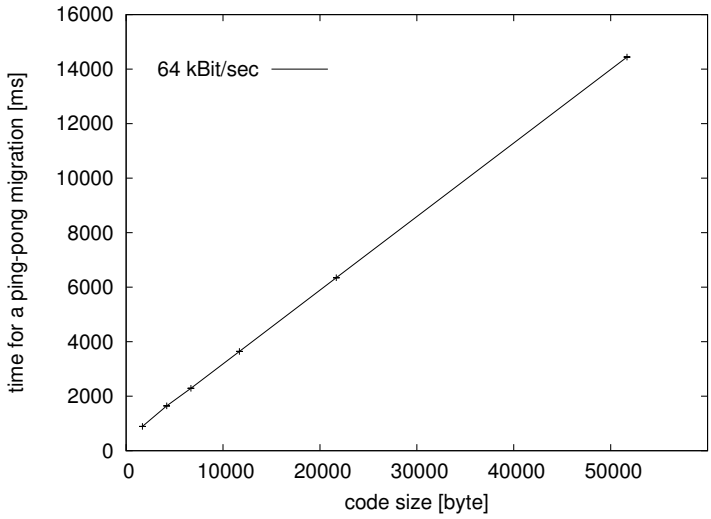
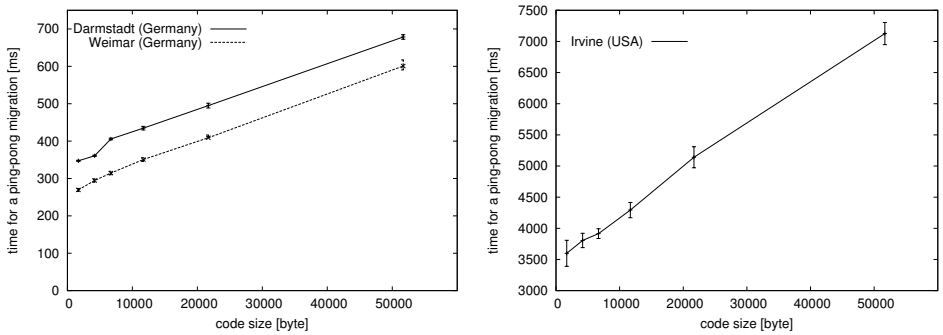


Figure 11.2.: Time for a ping-pong migration between computers ipc047 and ipc033 using a ISDN network connection (64 kb/sec).

Figure 11.2 shows the migration times in a network with bandwidth of 64 Kb/sec, which is the quality of a dial-up ISDN connection. This type of network is simulated using the traffic shaper technique of the Linux operating system, which artificially decreases throughput of a network device. The destination agency was started on computer ipc033. Although migration times are very high, we achieved a throughput that is only slightly below the theoretical optimum. For example, a single migration of the 51 685 byte agent has cost of 7220 ms, which results in a throughput of about 56 Kb/sec. It is questionable, whether this high throughput can be achieved in a real network environment too, but we were unfortunately unable to measure it.

Finally, Figure 11.3 shows the results for a ping-pong migration using a wide-area network. A migration of the smallest agent to the University of Weimar (gonzo) has a cost of about 135 ms and the largest agent has a cost of about 300 ms. A migration directed to the Fraunhofer Society Darmstadt (semoaext) is only about 13% slower than the migration to Weimar. The reason for this is probably that the agent is transmitted to gonzo over a 155 Mb/sec network connection, whereas it is transferred using a Giga-Bit network to Darmstadt. The migration to computer waylander at the University of Irvine has highest cost, as expected. The time for a single migration is between 1800 ms for the smallest and 3562 ms for the largest agent.



(a) Migration between computer ipc047 and computer gonzo (Weimar) resp. computer semoaext (Darmstadt).

(b) Migration between computer ipc047 and computer waylander (Irvine).

Figure 11.3.: Time for a ping-pong migration in different wide-area networks.

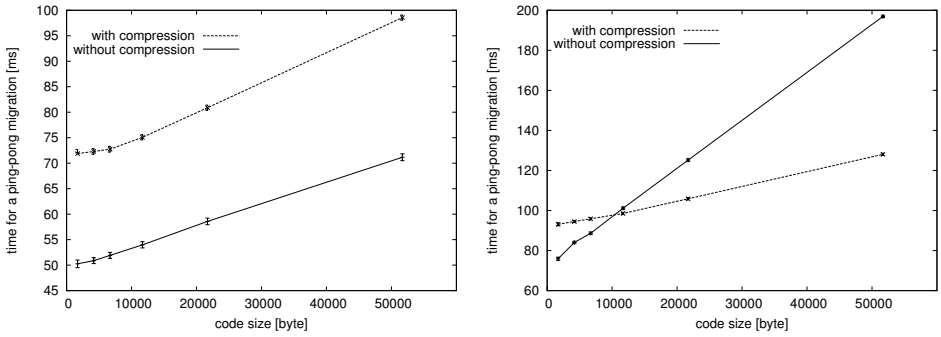
11.3.2. Transmission Time with regard to Data Compression

In the second experiment we examined the effect of data compression to reduce network load and transmission time. We used the same agents as in the experiment before and we measured the time for a single ping-pong migration between agencies at ipc047 and ipc033. If the agent is sent in a compressed form, all SATP messages are compressed using the technique described in Section 9.1.2 on page 175. The agent is transmitted using the PushToNext migration strategy without activating the code cache. All measurements were repeated 1000 times.

We compare the effect of data compression in two network environments, i.e. a 100 Mb/sec network via a router (Figure 11.4(a)) and the 10 Mb/sec network via a switch (Figure 11.4(b)). As can be seen from the graphs, in a high-bandwidth network, data compression has a negative effect on the migration performance for all code sizes. A migration is on average 40% slower than without data compression. This behavior can be explained by two reasons:

- Compressing small amounts of data sometimes increases the size of a message. For example, a SATP ADB message consists in our experiment on average of 79 byte in the uncompressed and 93 byte in the compressed form.

- Although a class of length 11 685 byte is for example reduced to 2176 byte, migration time is higher as it takes 12 ms to achieve this compression.



(a) Using a 100 Mb/sec network via a router.

(b) Using a 10 Mb/sec using a hub.

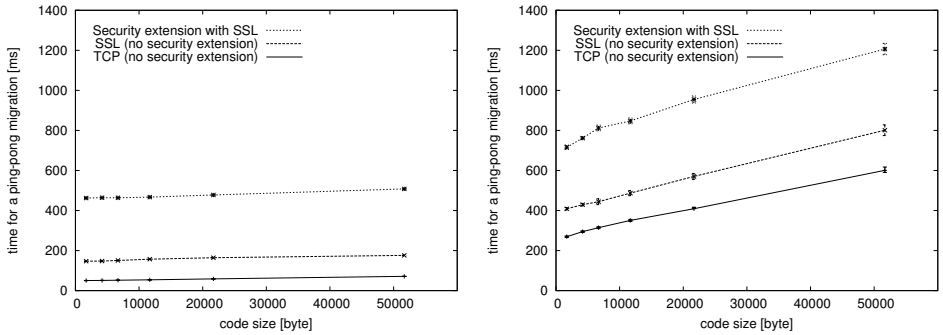
Figure 11.4.: Time for a ping-pong migration with regard to compression.

The consequence is that, although less data has to be transmitted, the overall migration time is higher as the time to determine the compression exceeds the time saved by the smaller network load. Therefore, in the high-bandwidth network, migration times with enabling data compression will almost always be higher than without compression. Only for very large agents, this technique might ever lead to a reduction of migration time. Such agents were out of the scope of our measurements and seem to be non-realistic.

In contrast, using the same computers in the low-bandwidth network, data compression has a positive effect for agents larger than approximately 10 000 byte. For smaller agents, we see the same effect as in the high-bandwidth network: the time to compute the compressed SATP message is higher than the benefit of transmitting smaller messages. For larger agents, we can now observe the expected effect. Sending the agent with compressed SATP messages leads to smaller migration times. Now, the time saved due to the smaller messages sent over the network exceeds the time to compress the messages.

As a consequence, we can conclude that the break-even point, where both curves intersect, depends on the network type. The lower the bandwidth of the underlying network is, the smaller the agent may be to make the effort for data compression pay off. We can assume that in wide-area networks, data compression is worth even for the smallest agent used in our experiments.

To improve the effect of data compression further, we see two main aspects. First, the process of compressing data can be done *before* the agent migrates. In the current



(a) Using a 100 Mb/sec network via a router.

(b) Using a wide-area network to gonzo.

Figure 11.5.: Time for a ping-pong migration with regard to different transmission protocols and security extensions.

implementation, data compression is done during the migration process. However, for the static parts of the agent, especially the agent's code, compression could be done earlier, for example while the agent is executed. On remote agencies, compressed code units should not be dropped and recomputed during the next migration, but saved and reused. As the agent's code is with high probability the largest part of the agent, this should improve the migration time dramatically. Second, we could use sophisticated compression algorithms for Java byte code, as for example those described by Pugh [1999] and Bradley et al. [1998], which will reduce the size for Java classes more than the *gzip* algorithm that we have used in our implementation. Both optimizations will be part of further investigation.

11.3.3. Transmission Time with regard to Security

In the next experiment we examined the cost of techniques that improve the security of a migrating agent. The goal was to show how migration times changes when agents migrate using the SSL network transmission protocol instead of TCP, and the security extension we described in Section 9.1.4 on page 181. The secure SSL network transmission protocol was configured to use server authentication and to not reuse sessions. The security extension of MDL includes digital signatures for SATP headers, state, and code messages. We used the same agents with the same code sizes as in the previous experiments. The agents are transmitted using the PushToNext strategy without enabling the code cache. All migrations were repeated 1000 times.

We conducted the experiment in two network environments. The first graph (Figure 11.5(a)) shows the migration times between ipc047 and ipc033 connected by a 100 Mb/sec network (router), whereas the second graph (Figure 11.5(b) shows migration times between ipc047 and gonzo at the University of Weimar.

The first graph shows that SSL transmission is between 2.5 times and 3 times more costly than using TCP, for example 73.5 ms for the smallest and 88 ms for the largest agent. The difference between both transmission types is approximately 52 ms in the fast network. The reason for the higher cost is the expensive handshaking protocol between sender and receiver to authenticate the server and to exchange encryption keys, which in sum increases the network load. This cost does not depend on the agent's size, which explains that migration time slows down in a higher extent for small agents. In addition, all data sent over the network must be encrypted resp. decrypted, which in fact increases the network load only slightly, but slows down migration times because of the high computational effort.

If we further activate, in addition to SSL transmission, all the security extensions described in Section 9.1.4 on page 181, i.e. signing of the SATP header, state, and all class files and inspecting classes at the destination to filter out malicious code, we can see in Figure 11.5(a) that migration times increase further. The migration cost for the smallest agent is now 231 ms and for the largest agent 253 ms, which is approximately between 7 and 10 times higher than migrating the agent without the security extension and with TCP. The total difference between both types of migration is approximately 215 ms. The reason for this could be at first sight a higher network load, because for each SATP message a digest must be sent over the network too. However, as we use the MD5 algorithms, each digest is only 16 byte long and during an agent migration in our experiments, only about 3 to 5 digest are transmitted, which should not have any effect on the migration time. Therefore, the higher migration times must be the result of the time to compute message digests and we found out that it takes for example 150 ms to determine the digest for a class of length 11 685 byte.

The most interesting result of these two measurements is that all security extensions increase network load only slightly but increase migration times considerably due to the high computational effort for signing and encryption. As can be seen from the graphs, the difference between the SSL curve and the *Security extension* curve is constantly about 300 ms for two migrations and does not depend on the network type. Notwithstanding, if we compare the SSL curve with the TCP curve, it can be noticed that the difference between both is higher in the wide-area network than in the local-area network, because SSL increase the network load of the whole connection and is, therefore, dependent on the network type.

As a summary, we can state that the migration overhead caused by security techniques can be expected to be increasingly smaller (measured as a percentage), if the bandwidth of the underlying network is getting lower. As in a low-bandwidth net-

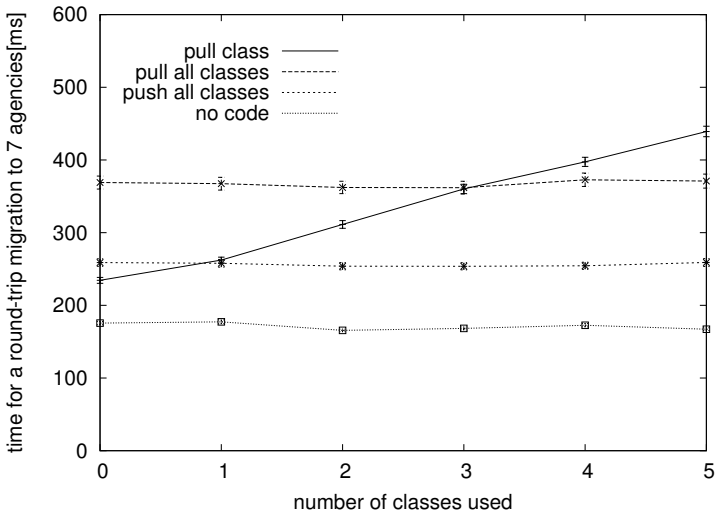


Figure 11.6.: Time for a migration to 7 agencies using different migration strategies.

work, migration times are higher, the constant overhead for security will not be of great weight. For example, we can expect migration times to increase by less than 10% when migrating between Jena and Irvine.

This result has a high practical advantage. Improving the security of mobile agents can only be achieved with high effort, which results in a slowdown of migration times. As the quality of the security techniques implemented for these experiments must be classified as basic, we can imagine that more sophisticated techniques will increase migration times further more. In specific application domains or in networks, where these security techniques are not necessary, Kalong's possibility to *switch off* this extension can be of high benefit. On the other side, in wide-area networks, where security of agent migration might be required, security has only a small impact on the whole migration time, which is anyhow high.

11.4. Effect of Migration Strategies

In this section we evaluate the influence of different migration strategies on migration performance. We conducted this experiment in a 100 Mb/sec network with all computers located in Jena. The agents were transmitted using the TCP transmission protocol. As measurements were repeated 1000 times.

The goal of this experiment is only to show that the difference between known migration strategies can be measured in a real-world network. This effect was forecast-

ed in our mathematical model in Section 6.2.2. It is not our goal to determine any migration strategy to be faster than others, because, as we have already discussed in Section 6.2.2, it depends on the concrete application scenario which migration strategies should be chosen – and this decision has to consider several parameters, for example the agent's code size, the network type, the probability for code execution, etc.

To perform our experiment, we simulate a typical application from the information retrieval domain. The agent visits several network nodes, where each platform has a database with documents of different types, for example simple text file, structured text files in XML or HTML, and images. Each document is characterized by a set of keywords. The agent has to visit each platform. First, it filters all documents according to a given set of keywords. The result is a set of *interesting* documents. Second, all these documents are examined in detail, which results in the set of all *significant* documents from which the agent takes a copy before migrating to the next platform. To examine an interesting document, a specific class file for the given document type is necessary on the current platform. Therefore, an agent consists of one class file for the agent itself, which contains code to perform the first step and all auxiliary tasks, like communication and route managing. Additionally, there are five other class files, each for one document type, which contain special code for the second step. If the agent finds a document of a specific type, the corresponding class file must be downloaded dynamically, if it is not already available on the current platform.

The experimental setup consists of a cluster of seven agent systems on computer systems ipc047, ipc026, ipc033, ipc051, ipc030, ipc031, and ipc032. On each platform we can change the number of document types that the agent will find interesting. Doing this we can directly influence the number of classes that will be downloaded.

The agent class is of size 2012 byte, whereas all auxiliary class are each 10 000 byte. In Figure 11.6 the graph can be seen for various numbers of document types found interesting resp. the number of classes needed at runtime and various migration strategies. Note, that in our experiments the agent does not take any data with it when it migrates to the next server. Therefore, our results show only the time of migrating code and initial data (which is again less than 100 bytes).

It can be seen that strategies PushAllClasses, PullAllClasses, and the strategy where no code is transmitted (because it is assumed that code is already available at all destinations, for example due to activating the code cache) are not dependent on the number of classes to load. This is clear, as the two first migration strategies always transmit all code without regard to their necessity. Obviously, transmitting no code is faster than all other migration strategies as the network loader is smallest in this case. If the agent must load all classes (PullAllClasses) it is on average 1.4 times slower than pushing the code to the next destination, because pulling all classes needs an additional network transmission on each agency.

Strategy PullPerClass is linear dependent on the number of classes to be downloaded dynamically, as only those classes are loaded that are needed for agent execution. It is faster in case of no interesting documents than all other methods (except transmitting no code at all), because only the agent class itself must be transmitted in this case. If only one additional class file must be loaded (of size 10 000 byte) strategy PullPerClass is as fast as the PushAllClasses strategy, which transmits 50 000 byte code at this time. When increasing the number of document types, the PullPerClass strategy is at last more than 70% slower than the PushAllClasses strategy and about 20% slower than the PullAllClasses strategy. This performance difference only results from the fact that code must be downloaded dynamically.

This experiment confirms the results of our mathematical model. The different amount of data sent over the network in each migration strategy can be measured in form of different migration times. The number of classes where curve PushAllClasses and PullPerClass intersect depends on the class size and the network type. For larger classes, this point can be expected to be higher, as the difference between sending all classes and only some classes becomes higher. In networks with lower bandwidth, it is not only more expensive to transmit data but also to open a network connection. The difference between all migration strategies can be expected to be higher. For example, it can be expected that the PullPerClass migration strategy is slower than the PushAllClasses strategy even for a small number of classes. However, as already explained, it was not the goal of this experiment to quantitatively compare migration strategies to find the *fastest*, as this is not possible in general. The results show that it is worth to reasons about different migration strategies and to choose a suitable one with regard to the application, to the network environment, the agent's size and many other parameters. Kalong's possibilities to program such migration strategies dynamically and to react in a very flexible and fine-grained way to these parameters, is a necessary and worth concept.

11.5. Effect of Caching

In this experiment we want to analyze the effect of the Kalong's code cache. As already described in Section 6.3.4 on page 105, the Kalong protocol can check, whether agent's code is already available at the destination agency before sending the code. In this experiment we compare migration times of a single agent to five agencies. The agent is started at gonzo (Weimar) and then migrates to four agencies in Jena (ipc047, ipc026, ipc033, ipc051) and returns back to gonzo. The agent uses the PushToNext migration strategy, once with and once without enabling the code cache. All measurements were repeated 200 times.

The solid line in Figure 11.7 shows the migration time for different code sizes in the case that the agent does not enable the code cache. It shows that migration time

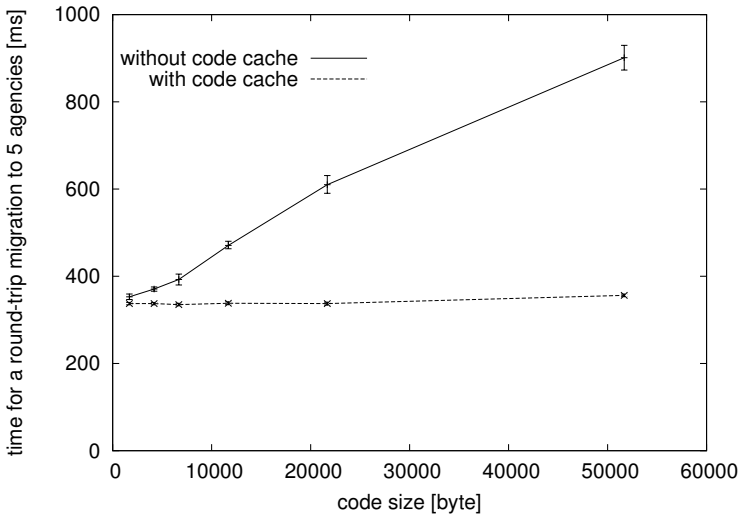


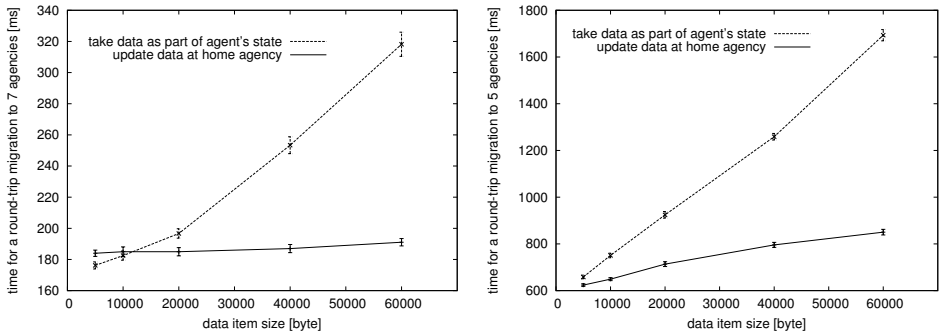
Figure 11.7.: Time for a migration to 5 agencies in a wide-area network with regard to the code cache.

depends on the code size, as expected. The dashed line in Figure 11.7 shows the migration time, if the agent activates the code cache and all agent's code is already available at the destination agency. It can be seen that migration times are no longer dependent on the code size as no code is transmitted in this case at all. The time for the migration is the result of transferring the agent's state and data items.

The effect of the code cache depends directly on code size, of course. For small code (1685 byte) the difference is only 15 ms for the whole round-trip. For the largest code (51685 byte) the complete migration time without using the cache is 2.5 times higher as with enabling the code cache.

Of course, the code cache can only have a positive effect, if at least a single class of the agent is already available at the destination agency. Figure 11.7 can also be interpreted as that the solid line shows the time for the first migration, whereas the dashed line shows the migration time for all following migrations for the same agent. An interesting question that we have not examined so far it, how the code cache increases migration performance, if a huge amount of agents of the same type, i.e. using the same classes, migrates to the same agencies.

Another interesting question concerns the overhead of the cache protocol. Unfortunately, we were not able to measure this overhead, but we can reason about the increase on network load that is caused by it. If we assume that an agent consists of five classes and each class name is 20 characters long, then the ADB message comprises of $5 \times (20 + 16)$ bytes, as the digest for each class is 16 byte long. The



(a) Using a 100 Mb/sec network.

(b) Using a wide-area network.

Figure 11.8.: Time for a migration of a single agent to 7 resp. 5 agencies in different networks. The agent creates a data item with given size and takes it as part of its state or sends it back to its home agency.

answer message of type ADBReply consists of only a single byte for each class. In sum, this are 185 byte that must be exchanged between sender and receiver agency in order to prevent the transmission of an agent's code.

11.6. Effect of Data Uploading

In the next experiment we want to examine the performance benefit of sending data items back to the agent's home server instead of taking it as part of the agent's state to all other agencies. We compare the migration time for a complete round-trip to 7 resp. 5 agencies for different data items in a local 100 Mb/sec network and a wide-area network. The agent migrates using the PushToNext migration strategy without using the code cache. The size of the agent is 2443 byte. The measurements were repeated 1000 times.

The first graph (11.8(a)) shows the result for the local network. The agent starts at ipc047 and first migrates to ipc026. There, the agent gets a new data item of a given size (5000, 10 000, 20 000, or 60 000 byte). Then, the agent migrates to five other agencies (ipc033, ipc051, ipc030, ipc031, and ipc032) before returning back to its home agency. The dashed line shows the result for the case that the agent takes the new data item as part of its state to all other agencies. The migration time depends on the size of the data item as expected. The solid line shows the result for the case that the agent sends the new data item back to its home agency before it leaves

ipc026. Here, the migration time does not depend on the data item size anymore as the agent does not carry the data item. The migration time does only depend on the code size and the constant size of the state. It can be seen in Figure 11.8(a) that for small data items it is slower to send data items back. In theses cases, the time to open an additional network connection to the home agency and to send a small data item is higher than the time that is needed to carry this data to all other agencies. For data items larger than about 11 000 byte, sending the data item back is faster. For example, in case of a 60 000 byte data item, sending the data item home is 1.6 times faster than taking it along.

The second graph (11.8(b)) shows the result for the wide-area network, where the agent is started at ipc047 (Jena) and then migrates to four other agencies in Weimar and Jena (gonzo, ipc051, gonzo, ipc051) before migrating back home. It can be seen that it is now for all data items faster to send the data item back than to carry it along. The solid line again shows the time for carrying the data item to all agencies, which depends on the size of the data item. In contrast to the first graph, sending the data item back is now also dependent on the data size, which can be explained with the time to send the data item back to the home server from Weimar to Jena using a low-bandwidth network connection. However, in the wide-area network we have a higher speed-up: sending the data item home is twice as fast as carrying it as part of the state. It is obvious that the performance benefit depends on the number of agencies to which the data item is not carried anymore and the bandwidth of the underlying network. If the number of servers is higher or the network slower, the performance speed-up is higher.

11.7. Effect of Code Servers

In this experiment we are interested in the performance gain that could be achieved by using code servers for dynamic code loading instead of home servers. An agent is able to initialize a code server dynamically during runtime at any agency that it is currently visiting. The effect is that some or all code units remain at the code server, even if the agent migrates to another agency. In future endeavors, the agent can download classes from this code server, for example if it is placed nearer to the current agency than the home agency.

We compare the time for a complete round-trip to four agencies vs. the size of the classes that must be loaded. The agent base class has 2176 byte and needs a single other class of size 2500, 5000, 10 000, 20 000, or 50 000 byte. Both classes are loaded dynamically during runtime. All measurements were repeated 200 times.

Our scenario consists of four agencies on gonzo, ipc026, ipc047, and ipc051. The agent is started on gonzo (Weimar) and migrates using the PullAllClasses migration strategy. In the first case, the code is loaded from the agent's home server, and in

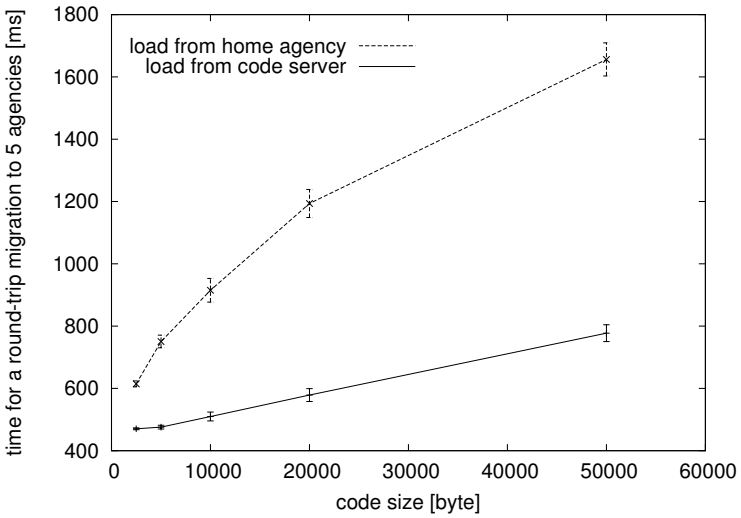


Figure 11.9.: Time for a migration to 4 agencies using the Pull strategy with regard to different locations of the code server.

the other case, all code is loaded from a code server that the agent initialized on agency ipc026. Migration times include in the second case the time to release the code server at the end.

Figure 11.9 shows the result of our measurement. The dashed line shows the migration times for an agent that always loads classes from its home agency. The time depends on the code size and increases steadily with the size of the code that must be downloaded over the wide-area network. If only a small class must be loaded, the agent needs about 676 ms for the complete tour, and when loading the largest code, it needs about 1453 ms. The solid line shows the results for the case that the agent loads classes from a code server that is located in the local network (on agency ipc026). Migration time depends on the code size, of course, but the increase is not so steep. Even for the smallest agent, this type of migration only needs 494 ms, which is an improvement of 17% as compared to the first migration type. For the largest agent, the improvement is even higher. The agent needs 717 ms, which is an improvement of more than 50%.

The experiment shows that it is worth to use code servers to improve the performance of loading code. In the case that a code server could be placed at a node that is accessible by a faster network than the home server, a code server makes sense. Using code servers has no drawback, as it is without any cost to activate a code server and it has only minor costs to release a code server after the agent has terminated. In our measurements, sending the SATP message to release the code server had costs

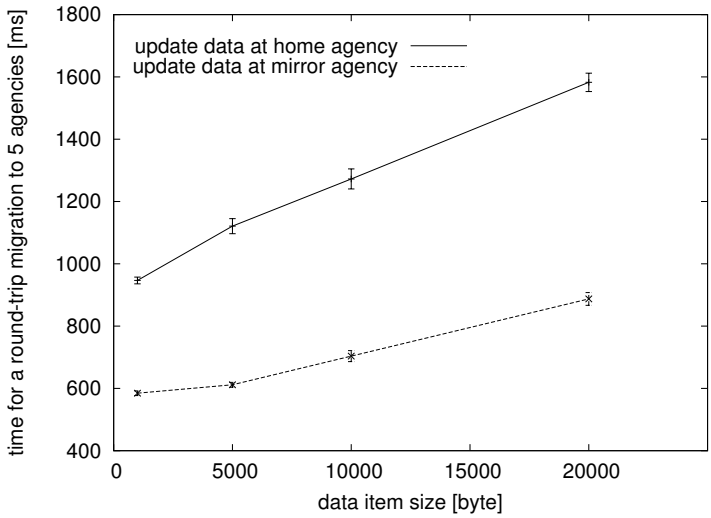


Figure 11.10.: Time for a migration to 5 agencies, where the agent uploads data items either on its home server or on a near mirror server.

of less than 10 ms.

11.8. Effect of Mirrors

The last experiment considers the effect of a mirror server to reduce costs for loading and updating data items. The scenario in this experiment consists of five agencies on computers gonzo, ipc026, ipc047, ipc033, and ipc051. The agent consists of 3095 byte of code and migrates using the PushToNext migration strategy without enabling the code cache. All measurements were repeated 200 times.

The agent is started at gonzo and then migrates to ipc026, where it creates a data item with 1000, 5000, 10 000, or 20 000 byte. In the first case, the data item is sent to the agent's home server, while in the other case, the agent initializes a mirror server at ipc026. The agent then migrates to the other agencies, where it loads the data item from the home resp. mirror server, modifies it, and uploads it again. Finally, if a mirror agency exists, the data item is loaded from the mirror to the home agency. Therefore, a data item is transmitted seven times between an agency and the mirror server resp. the home agency.

The solid line in Figure 11.10 shows the migration time for the first case, when data items are updated at the home agency. The migration time is linear dependent to the data size and grows from 946 ms for the smallest data item up to 1582 ms for the largest data item. The dashed line shows the time for the second case, where

data items are uploaded at the mirror agency. The complete round-trip has now only costs of 584 ms for the smallest data item, which is a speed-up of about 38% and costs of only 887 ms for the largest data item, which is a speed-up of about 44%. The reason for the speed-up is obviously the fact that the mirror server was accessible over a high-bandwidth connection and six of the seven data transmission used this type of network. Without the mirror server, all data transmission were done using the low-bandwidth connection between Jena and Weimar. We can conclude that it makes sense to activate a mirror agency for data uploading and downloading, if this mirror agency can be accessed using a faster network type than the home agency. The benefit of the mirror is the greater, the higher the network load is for data transmissions between an agency and the mirror.

12. Conclusions

Mobile agents provide a new and fascinating approach to the design and architecture of distributed systems. They must be seen as a supplement to other, more traditional design paradigms, as for example the client-server paradigm. Mobile agents have many advantages as compared to the client-server approach. In this thesis, we focused on the *performance argument* that states that mobile agents are able to reduce network load and processing time by shipping code to the data instead of downloading data to the code.

The starting point of our research was the problem that the performance argument only holds under specific assumptions, while in other cases, today's mobile agents might cause higher network load as compared to the client-server approach. Therefore, the software designer currently has the problem to decide which paradigm should be used. The only solution published in literature so far suggests to estimate the network load in both paradigms for a specific application or application domain, based on a mathematical model, and choose the approach that produces lower network load. We argued that this kind of decision is unsafe, as a change of the parameters of the mathematical model might overset the whole decision.

The main hypothesis of this work was that it is imperative to analyze the drawbacks of today's mobile agents in order to find the reasons for their poor performance in some situations.

In the first step we, therefore, analyzed the network load of today's mobile agents as compared to client-server techniques in several application scenarios. We found that the simple migration technique of today's mobile agents, which does not differ from techniques used in mobile object systems, is the main reason of their poor performance. Using mathematical models, we were also able to proof that in fact mobile agents have inherent drawbacks which in certain situations prevent them from ever being faster than client-server approaches.

In the second step, we then analyzed the migration process of mobile agents in detail and discussed design issues and the possibility of optimizations for agent migration. We proposed several improvements of existing migration techniques in order to consider the specific requirements of mobile agent systems. In detail, we proposed that mobile agents should not be transmitted as a single transmission unit, but code and data items should be able to migrate independently. We suggested an *adaptive technique for code and data migration*, where the agent can choose which pieces of code and which data items should be transferred to which server. Additionally, we

also proposed two new types of agencies, namely *code servers* and *mirrors*, the agent can dynamically initialize in order to decrease execution time. We found that none of the existing mobile agent systems is able to implement any of this improvements and we, therefore, suggested and implemented our new Kalong mobility model.

In the third step, we specified this mobility model and a migration protocol, named SATP. The most practical result of this thesis is a software component, named Kalong, which provides services to migrate a mobile agent in a very flexible and fine-grained way. Kalong can be seen as a virtual machine for agent migration, as it provides a minimal and generally applicable set of commands to conduct the whole migration process. As Kalong does not rely on any specific requirements of the surrounding mobile agent system, we believe that it is possible to adapt Kalong to almost any agent system. Kalong is extendable by the user and we presented several examples to show that basic security problems of mobile agents can be solved within Kalong.

In the last step, we evaluated Kalong's performance and especially the impact of Kalong's new features. We were able to show that the performance of mobile agents greatly benefits from these new functions.

Clearly, further work is necessary to refine the results of this thesis. The most important and most promising issue is, of course, the area of automated migration strategies. We already implemented a software component to measure network load and we are on the way to implement a technique for static and dynamic profiling of mobile agents. Together, both components are the base for an automated strategy, which itself can select the code units and data items to migrate in order to achieve lowest network load. Another important issue is the refinement of the network performance model that we unfortunately were not able to validate yet. The next step with the Kalong software component is to couple it to other mobile agent systems, as for example Semoa or Grasshopper. Finally, Kalong must be evaluated against client-server based techniques in order to examine, whether mobile agents with sophisticated migration strategies can outperform traditional client-server approaches.

Further investigations have to be done to validate and finalize the Kalong approach. However, we hope that the work described in this thesis already contributes to the understanding of the migration process of mobile agents and opens new gates to further thoughts.

Bibliography

Numbers behind the reference point to pages on which the reference is cited.

Anurag Acharya, Mudumbai Ranganathan, and Joel Saltz. Sumatra: A language for resource-aware mobile programs. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet (MOS'96), Linz (Austria), July 1996*, volume 1222 of *Lecture Notes in Computer Science*, pages 111–130, Berlin, 1997. Springer-Verlag. 70, 74

Adobe Systems, Inc. *PostScript® Language Reference Manual*. Addison-Wesley, Reading, MA, 3rd edition, 1999. ISBN 0-201-37922-8. 17

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Amsterdam, 2000. ISBN 0-201-10088-6. 88

Wolfram Amme, Niall Dalton, Michael Franz, and Jeffery von Ronne. SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form. *ACM SIGPLAN Notices*, 36(5):137–147, May 2001. 89

Ken Arnold and James Gosling. *The Java™Programming Language*. The Java Series. Addison-Wesley, Reading, MA, 3rd edition, 2000. ISBN 0-201-70433-1. 15, 51

Marco Avvenuti and Alessio Vecchio. MobileRMI: a toolkit for enhancing Java Remote Method Invocation with mobility. In Ciaran Bryce and Chrislain Razafimahefa, editors, *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages, Sophia Antipolis (France), June 2000*, 2000. Paper is only available online `http://cui.unige.ch/~ecoopws`. 53

Mario Baldi, Silvano Gai, and Gian Pietro Picco. Exploiting code mobility in decentralized and flexible network management. 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 27–38, Berlin, 1997. Springer-Verlag. 24

Mario Baldi and Gian Pietro Picco. Evaluating the tradeoffs of mobile code design paradigms in network management applications. In R. Kemmerer and K. Futatsugi,

editors, *Proceedings of the 20th International Conference on Software Engineering (ICSE'98), Kyoto (Japan), April 1998*, pages 146–155, Los Alamitos, CA, 1998. IEEE Computer Society Press. 24

Michel Barbeau. Modeling and comparison of bandwidth usage of three migration strategies of mobile agents. 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 197–210, Teaneck, NJ, 1999. World Scientific Pub. 90

Joachim Baumann. Agents: A Triptychon of Problems. In *Proceedings of the 1st ECOOP Workshop on Mobile Object Systems: Objects and Agents: Love at First Sight or Shotgun Wedding?, Aarhus (Denmark), August 1995*, Aarhus, Denmark, 1995. 71

Joachim Baumann, Fritz Hohl, Kurt Rothermel, Markus Schwehm, and Markus Straßer. Mole 3.0: A Middleware for Java-Based Mobile Software Agents. In Nigel Davie, Kerry Raymond, and Jochen Seitz, editors, *Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 355–372, Berlin, 1998. Springer-Verlag. 19, 25

C. Bäumer, M. Breugst, S. Choy, and T. 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, Teaneck, NJ, 1999. World Scientific Pub. 19

Werner Van Belle and Theo D'Hondt. Agent mobility and reification of computational state. In Ciaran Bryce and Chrislain Razafimahefa, editors, *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages, Sophia Antipolis (France), June 2000*, 2000. Paper is only available online `http://cui.unige.ch/~ecoopws`. 71

Lorenzo Bettini and Rocco De Nicola. Translating strong mobility into weak mobility. In Gian Pietro Picco, editor, *Mobile Agents, Proceedings of the 5th International Conference (MA 2001), Atlanta (USA), December 2001*, volume 2240 of *Lecture Notes in Computer Science*, pages 182–197, Berlin, 2001. Springer-Verlag. 70

Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984. 10, 17

J. K. Boggs. IBM Remote Job Entry Facility: Generalized Subsystem Rmote Job Entry Facility. Technical Report 752, IBM Technical Disclosure Bulletin, August 1973. 17

Jan Bosch and Stuart Mitchell, editors. *Object-Oriented Technology: ECOOP'97 Workshop Reader*, volume 1357 of *Lecture Notes in Computer Science*, Berlin, 1998. Springer-Verlag. 257, 258

Q. Bradley, R. Horspool, and J. Vitek. Compression of Java Bytecode. In *Proc. of CASCON'98*, 1998. 91, 237

Jeffrey Bradshaw, editor. *Software Agents*. The MIT Press, Menlo Park, CA, 1996. ISBN 0-262-52234-9. 13, 18, 265

Peter Braun. Feingranulare und dynamische Migration bei mobilen Agenten. In Clemens H. Cap, Werner Erhard, and Wolfgang Koch, editors, *15. GI/ITG Fachtagung Architektur von Rechensystemen (ARCS'99), Jena (Germany), October 1999*, pages 225–228, Berlin, 1999a. VDE Verlag. 267

Peter Braun. Über die Migration bei mobilen Agenten. Technical Report Math-/Inf/99/13, Friedrich-Schiller-Universität Jena, Institut für Informatik, 1999b. 267

Peter Braun, Jan Eismann, and Wilhelm Rossak. Managing Tracy Agent Server Networks. Technical Report 01/12, Friedrich-Schiller-Universität Jena, Institut für Informatik, 2001a. 267, 278

Peter Braun, Jan Eismann, Wilhelm Rossak, Richard Kowalczyk, Bogdan Franczyk, and Andreas Speck. Integrating Mobile and Intelligent Agents in E-Marketplaces. In *Proceedings of the 5th International Conference on Business Information Systems (BIS-2002), Poznań (Poland), April 2002*, Lecture Notes in Computer Science, Berlin, 2002. Springer-Verlag. 268

Peter Braun, Christian Erfurth, and Wilhelm Rossak. An Introduction to the Tracy Mobile Agent System. Technical Report Math/Inf/00/24, Friedrich-Schiller-Universität Jena, Institut für Informatik, September 2000a. 267, 279

Peter Braun, Christian Erfurth, and Wilhelm Rossak. Experiences with State-of-the-Art Migration Strategies. In David Kotz and Friedemann Mattern, editors, *Agent Systems, Mobile Agents, and Applications, Proceedings of the Second International Workshop on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA 2000), Zurich (Switzerland), September 2000*, volume 1882 of *Lecture Notes in Computer Science*, Berlin, 2000b. Springer-Verlag. Poster Proceedings. 74, 93

Peter Braun, Christian Erfurth, and Wilhelm Rossak. Performance Evaluation of Various Migration Strategies for Mobile Agents. In Ulrich Killat and Winfried Lamersdorf, editors, *Fachtagung Kommunikation in verteilten Systemen (KiVS 2001), Hamburg (Germany), February 2001*, Informatik aktuell, pages 315–324. Springer-Verlag, 2001b. 74, 93

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture*. John Wiley and Sons, New York, NY, 1996. ISBN 0-471-95869-7. 149, 279

G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agent technology: Current trends and perspectives. In *Congresso annuale AICA '98, Napoli (Italy), November 1998*, 1998. The paper is only available online at `http://polaris.ing.unimo.it/MOON/papers/`. 81

G. Cabri, L. Leonardi, and F. Zambonelli. Weak and strong mobility in mobile agent applications. In *Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java (PA JAVA 2000), Manchester (UK), April 2000*, 2000. The paper is only available online at `http://polaris.ing.unimo.it/MOON/papers/`. 71

Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with a mobile code paradigm. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97), Seattle (USA), April 1997*, pages 22–32, New York, 1997. ACM Press. 4, 24

David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea? In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet (MOS'96), Linz (Austria), July 1996*, volume 1222 of *Lecture Notes in Computer Science*, pages 25–45. Springer-Verlag, Berlin, 1997. 3

Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing mobile code languages. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet (MOS'96), Linz (Austria), July 1996*, volume 1222 of *Lecture Notes in Computer Science*, pages 93–110, Berlin, 1997a. Springer-Verlag. 51

Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. A characterization of mobility and state distribution in mobile code languages. In Max Mühlhäuser, editor, *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems: Agents on the Move, Linz (Austria), July 1996*, pages 10–19, Heidelberg, 1997b. dpunkt Verlag. 51

Marios D. Dikaiakos and George Samaras. Qualtitative performance analysis of mobile agent systems: A hierachical approach. Technical report, University of Cyprus, Department of Computer Science, June 2000. 228

Fred Douglis and John K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Approach. *Software—Practice and Experience*, 21(7):1–27, July 1991. 18

F. Brent Dubach, Robert M. Rutherford, and Charles M. Shub. Process-originated migration in a heterogeneous environment. In *Seventeenth Annual ACM Computer Science Conference, Louisville, USA, February 1989*, pages 98–102, New York, 1989. ACM Press. 87

Christian Erfurth. Eine Plattform zur Migration von Komponenten in einem verteilten System. Diplomarbeit, Friedrich-Schiller-Universität Jena, Institut für Informatik, December 1999. 267

Christian Erfurth, Peter Braun, and Wilhelm 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, 2001a. 90

Christian Erfurth, Peter Braun, and Wilhelm Rossak. Some thoughts on migration intelligence for mobile agents. Technical Report 01/09, Friedrich-Schiller-Universität Jena, April 2001b. 90

Joseph R. Falcone. A programmable interface language for heterogeneous distributed sytems. *IEEE Transactions on Computer Systems*, 5(4):330–351, July 1987. 17

William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Authentication and state appraisal. In *Proceedings of the Fourth European Symposium on Research in Computer Security*, volume 1146 of *Lecture Notes in Computer Science*, pages 118–130, Rome, Italy, September 1996a. Springer-Verlag. 177

William M. Farmer, Joshua D. Guttman, and Vipin Swarup. Security for mobile agents: Issues and requirements. In *Proceedings of the 19th National Information Systems Security Conference*, pages 591–597, Baltimore, Md., October 1996b. 177, 180

Christian Fensch. Class Splitting as a Method to Reduce Network Traffic in a Mobile Agent System. Diplomarbeit, Friedrich-Schiller-Universität Jena, Institut für Informatik, 2001. 223

Leonard N. Foner. Entertainming Agents - A Sociological Case Study. In W. Lewis Johnson, editor, *Proceedings of the First International Conference on Autonomous Agents, Marina del Rey (USA), Februar 1997*, pages 122–129, New York, 1997. ACM Press. 12

Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5), May 1998. 81

Munehiro Fukuda, Lubomir F. Bic, Michael B. Dillencourt, and Fehmina Merchant. Intra- and inter-object coordination with MESSENGERS. In *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 179–196, Cesena, Italy, April 1996. Springer-Verlag. 17

Stefan Fünfrocken. Transparent migration of Java-based mobile agents. 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 26–37, Berlin, 1999. Springer-Verlag. 70, 90

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994. ISBN 0-201-63361-2. 66, 166

Sven Geisenhainer. Entwurf und Implementierung einer Anwendung zur Visualisierung und Steuerung des Agentensystems Tracy. Diplomarbeit, Friedrich-Schiller-Universität Jena, Institut für Informatik, July 2000. 267

Robert S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop, Monterey (USA), July 1996*, pages 9–23, Berkeley, 1996. USENIX Association. 70

Robert S. Gray. *Agent Tcl: A flexible and secure mobile-agent system*. PhD thesis, Dartmouth College, Computer Science, 1997a. Available as technical report PCS-TR98-327. 19

Robert S. Gray. *Agent Tcl: A flexible and secure mobile-agent system*. PhD thesis, Department of Computer Science, Dartmouth College, June 1997b. 228

Robert S. Gray, George Cybenko, David Kotz, Ronald A. Peterson, and Daniela Rus. D'Agents: Applications and performance of a mobile-agent system. *Software—Practice and Experience*, 32(6):543–573, May 2002. 19

Robert S. Gray, David Kotz, Ronald A. Peterson, Joyce Barton, Daria A. Chacón, Peter Gerken, Martin O. Hofmann, Jeffrey Bradshaw, Maggie R. Breedy, Renia Jeffers, and Niranjan Suri. Mobile-agent versus client/server performance: Scalability in an information-retrieval task. In Gian Pietro Picco, editor, *Mobile Agents, Proceedings of the 5th International Conference (MA 2001), Atlanta (USA), December 2001*, volume 2240 of *Lecture Notes in Computer Science*, pages 229–243, Berlin, 2001. Springer-Verlag. 38

Thomas Gschwind. Comparing Object Oriented Mobile Agent Systems. In Ciaran Bryce and Chrislain Razafimahefa, editors, *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages, Sophia Antipolis (France), June 2000*, 2000. Paper is only available online `http://cui.unige.ch/~ecoopws`. 19

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, Los Alamitos, CA, 1998. IEEE Computer Society Press. 61, 81

C. G. Harrison, D. M. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Research Report RC 19887, IBM Research Division, 1995. 3

Carl E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977. 12

Fritz Hohl, Peter Klar, and Joachim Baumann. Efficient code migration for modular mobile agents. 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. The paper is not published in the workshop reader (Bosch and Mitchell [1998]) and is perhaps only available online. 92, 109

IEEE IC-Online. The Future of Software Agents, Internet Computing Online Virtual Roundtable with Mani Chandy, Danny Lange, Pattie Maes, John Ousterhout, Jeff Rosenschein, Sankar Virdhagriswaran, James E. White. `http://www.computer.org/internet/v1n4/round.htm`, July/August 1997. 16

IKV. *Grasshopper Programmer's Guide, Release 2.2.* IKV++ GmbH, Berlin, March 2001a. 19

IKV. *Grasshopper User's Guide, Release 2.2.* IKV++ GmbH, Berlin, March 2001b. 19

Torsten Illmann, Tilman Krüger, Frank Kargl, and Michael Weber. Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture. In Gian Pietro Picco, editor, *Mobile Agents, Proceedings of the 5th International Conference (MA 2001), Atlanta (USA), December 2001*, volume 2240 of *Lecture Notes in Computer Science*, pages 198–212, Berlin, 2001. Springer-Verlag. 70

Ashraf Iqbal, Joachim Baumann, and Markus Straßer. Efficient algorithms to find optimal agent migration strategies. Technical Report 1998/05, Universität Stuttgart, Fakultät für Informatik, April 1998. 26

Leila Ismail and Daniel Hagimont. A performance evaluation of the mobile agent paradigm. *ACM SIGPLAN Notices*, 34(10):306–313, 1999. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99), Denver (USA), Nov. 1999. 39

Ravi Jain, Farooq Anjum, and Amjad Umar. A comparison of mobile agent and client-server paradigms for information retrieval tasks in virtual enterprises. In *Proceedings of the Academia/Industry Working Conference on Research Challenges (AIWORC'00), Buffalo, NY (USA), April 2000*, Los Alamitos, CA, 2000. IEEE Computer Society Press. 38

Wayne A. Jansen. Countermeasures for mobile agent security. *Computer Communications: Special Issue on Advances in Research and Application of Network Security*, November 2000. 177

Dag Johansen, Nils P. Sudmann, and Robbert van Renesse. Performance issues in TACOMA. 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. The paper is not published in the workshop reader (Bosch and Mitchell [1998]) and is perhaps only available online. 228

Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating system support for mobile agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems (HotOS-V), Orcas Island (USA), May 1995*, pages 42–45, Los Alamitos, CA, 1995. IEEE Computer Society Press. 19, 47

Günter Karjoth, Danny B. Lange, and Mitsuru Oshima. A security model for agents. *IEEE Internet Computing*, 1(4), July/August 1997. 177

Neeran M. Karnik. *Security in Mobile Agent Systems*. PhD thesis, Univeristy of Minnesota, Department of Computer Science, 1998. 177, 180

Neeran M. Karnik and Anand R. Tripathi. Design Issues in Mobile Agent Programming Systems. *IEEE Concurrency*, 6(6):52–61, 1998. 61, 81

Neeran M. Karnik and Anand R. Tripathi. Security in the Ajanta Mobile Agent Programming System. *Software—Practice and Experience*, 31(4):301–329, April 2001. 19, 177

Joseph Kiniry and Daniel Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4):21–30, July/August 1997. 19

Frederick Knabe. Performance-oriented implementation strategies for a mobile agent language. In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet (MOS'96), Linz (Austria), July 1996*, volume 1222 of *Lecture Notes in Computer Science*, pages 229–244. Springer-Verlag, Berlin, 1997a. 72, 87, 88

Frederick C. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, Paittsburgh, Pa., December 1995. 51

Frederick C. Knabe. An overview of mobile agent programming. In Mads Dam, editor, *Proceedings of the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, Stockholm (Sweden), Juni 1996*, volume 1192 of *Lecture Notes in Computer Science*, Berlin, 1997b. Springer-Verlag. Invited paper. 51

Pål Knudsen. Comparing two distributed computing paradigms – a performance case study. Master's thesis, University of Tromsø, August 1995. 39

R. Koblick. Concordia. *Communications of the ACM*, 42(3):96–97, March 1999. 19

David Kotz and Robert S. Gray. Mobile Agents and the Future of the Internet. *ACM Operating Systems Review*, 33(3):7–13, July 1999. 42

Richard Kowalczyk, Peter Braun, Jan Eismann, Bogdan Franczyk, Wilhelm Rossak, and Andreas Speck. 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, Los Alamitos, CA, 2002. IEEE Computer Society Press. 149, 268

Chandra Krintz, Brad Calder, and Urs Hölzle. Reducing Transfer Delay Using Java Class File Splitting and Prefetching. In Loren Meissner, editor, *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99), November 1999*, pages 276–291, New York, 1999. ACM Press. 92, 224

Danny B. Lange and Mitsuru Ishima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, Reading, MA, 1998. ISBN 0-201-32582-9. xiv, 19, 51, 178, 180

Sashi Lazar and Deepinder Sidhu. Discovery: A Mobile Agent Framework for Distributed Applications. Technical report, Maryland Center for Telecommunications Research, Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County, 1998. 68

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 2nd edition, 1999. ISBN 0-201-43294-3. 15, 52, 59

Dejan S. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, Danny Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. 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, Berlin, 1999. Springer-Verlag. 19

Ingo Müller. Integrating Mobile Agent Technology to an E-Marketplace Solution – The InterMarket Marketplace. Diplomarbeit, Friedrich-Schiller-Universität Jena, Institut für Informatik, June 2002. 268

B. J. Nelson. *Remote procedure call*. PhD thesis, Carnegie-Mellon University, Pittsburgh, Pa., 1981. 10, 17

ObjectSpace. *Voyager Core Package Technical Overview: The Agent ORB for Java*, December 1997. 19, 68

ObjectSpace. *Voyager Core Package Version 2.0: Technical Overview*, 1998. 19, 72

A. Outtagarts, M. Kadoch, and S. Soulhi. Client-Server and Mobile Agent: Performances Comparative Study in the Management of MIBs. 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 69–81, Teaneck, NJ, 1999. World Scientific Pub. 39

Stavros Papastavrou, George Samaras, and Evaggelia Pitoura. Mobile agents for WWW distributed database access. In *Proceedings of the 15th International Conference on Data Engineering, Sydney (Australia), March 1999*, pages 228–237, Los Alamitos, CA, 1999. IEEE Computer Society Press. 39

Holger Peine. An introduction to mobile agent programming and the Ara system. Technical Report ZRI-Report 1/97, Department of Computer Science, University of Kaiserslautern, Germany, 1997. 72

Holger Peine and Torsten Stolpmann. The architecture of the Ara platform for mobile agents. 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 50–61, Berlin, Germany, 1997. Springer-Verlag. 70

Heiko Peter. Entwurf und Implementierung eines temporalen Vorwärtsplaners. Diplomarbeit, Friedrich-Schiller-Universität Jena, Institut für Informatik, December 2002. 268

Michael Philippsen and Matthias Zenger. JavaParty – Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997. 89

Gian Pietro Picco. *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility.* PhD thesis, Politecnico di Torino, Italy, February 1998. 9, 24, 26, 61

Gian Pietro Picco. μCODE: A Lightweight and Flexible Mobile Code Toolkit. 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 160–171, Berlin, 1999. Springer-Verlag. 108

Ulrich Pinsdorf and Volker Roth. Mobile agent interoperability patterns and practice. In *Proceedings of the 9th Annual Conference and Workshop on the Engineering of Computer-based Systems (ECBS-2002), Lund (Sweden), April 2002*, pages 238–244, Los Alamitos, CA, 2002. IEEE Computer Society Press. 47, 150

William Pugh. Compressing Java class files. *ACM SIGPLAN Notices*, 34(5):247–258, May 1999. 91, 237

Antonio Puliafito, Salvatore Riccobene, and Marco Scarpa. Which Protocol Should I Use? An Analytical Comparison of the Client-Server, Remote Evaluation and Mobile Agents Protocols. 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*, Los Alamitos, CA, 1999. IEEE Computer Society Press. 38

Antonio Puliafito, Salvatore Riccobene, and Marco Scarpa. Which paradigm should I use? An analytical comparison of the client-server, remote evaluation and mobile agent paradigms. *Concurrency and Computation: Practice and Experience*, 13(1): 71–94, 2001. 38

R. L. Rivest. The MD5 Message-Digest Algorithm. RFC 1321, April 1992. 106

Volker Roth and Vania Conan. Encrypting Java Archives and its application to mobile agent security. In Frank Dignum and Carles Sierra, editors, *Agent Mediated Electronic Commerce: A European Perspective*, volume 1991 of *Lecture Notes in Artifical Intelligence*, pages 232–244. Springer-Verlag, Berlin, 2001. 177, 181

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, Los Alamitos, CA, 2001. IEEE Computer Society Press. 19, 51, 177

Marcelo Gonçalves Rubinstein and Otto Carlos Muniz Bandeira Duarte. Evaluating tradeoffs of mobile agents in network management. *Networking and Information Systems Journal*, 2(2):237–252, 1999. 41

Jeff Rulifson. RFC 5: The Decode-Encode Language, June 1969. `http://www.faqs.org/rfcs/rfc5.html`. 16

G. Samaras, M. D. Dikaiakos, C. Spyrou, and A. Liverdos. Mobile Agent Platforms for Web-Databases: A Qualitative and Quantitative Assessment. 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 50–64, Los Alamitos, CA, 1999. IEEE Computer Society Press. 39

Ichiro Satoh. Adaptive Protocols for Agent Migration. In *Proceedings of 21st IEEE International Conference on Distributed Computing Systems (ICDCS'2001), Mesa (USA), April 2001*, pages 711–714, Los Alamitos, CA, 2001. IEEE Computer Society Press. 110

Ichiro Satoh. Dynamic Configuration of Agent Migration Protocols for the Internet. In *Proceedings of the 2002 International Symposium on Applications and the Internet (SAINT'2002), Nara City (Japan), January/February 2002*, pages 119–126, Los Alamitos, CA, 2002. IEEE Computer Society Press. 110

Steffen Schreiber. Beschreibung und Analyse von dynamische Netzen für Agentensysteme. Diplomarbeit, Friedrich-Schiller-Universität Jena, Institut für Informatik, July 2002. 222, 268

Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In P. Ciancarini and A. L. Wolf, editors, *Proceedings of the Third International Conference on Coordination Models and Languages (Coordination'99), Amsterdam (The Netherlands), April 1999*, volume 1594 of *Lecture Notes in Computer Science*, pages 211–226, Berlin, 1999. Springer-Verlag. 70

Charles M. Shub. Native code process-originated migration in a heterogeneous environment. In *Eighteenth Annual ACM Computer Science Conference, Washingtion DC, USA, February 1990*, pages 266–270, New York, 1990. ACM Press. 87

Louis Moura Silva, Guilherme Soares, Paulo Martins, Victor Batista, and Luís Santos. Comparing the performance of mobile agent systems. *Journal of Computer Communications, Special Issue on Mobile Software Agents for Telecommunications*, 23(8):769–778, April 2000. 228

Guilherme Soares and Louis Moura Silva. Optimizing the migration of mobile agents. In Ahmed Karmouch and Roger Impey, editors, *Mobile Agents for Telecommunication Applications, Proceedings of the First International Workshop (MATA 1999), Ottawa (Canada), October 1999*, Teaneck, NJ, 1999. World Scientific Pub. 89, 91

Tammo Spalink, John H. Hartman, and Garth A. Gibson. A mobile agent's effects on file service. *IEEE Concurrency*, 8(2):62–69, April/June 2000. 39

Constantinos Spyrou, George Samaras, Paraskevas Evripidou, and Evaggelia Pitoura. Wireless Computational Models: Mobile Agents to the Rescue. In *Proceedings of the 10th International Workshop on Database & Expert Systems Applications (DEXA-1999), Florence (Italy), September 1999*, pages 127–133, Los Alamitos, CA, 2000. IEEE Computer Society Press. 39

James W. Stamos. *Remote Evaluation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Ma., January 1986. Also available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-354. 10

Bjarne Steensgaard and Eric Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, USA, December 1995*, pages 68–78, New York, 1995. ACM Press. 87

Luk Stoops, Tom Mens, and Theo D'Hondt. Fine-grained interlaced code loading for mobile systems. In *8th ECOOP Workshop on Mobile Object Systems: Agent Applications and New Frontiers, Malaga (Spain), June 2002*, 2002. Paper is only available online `http://cui.unige.ch/~ecoopws`. 92

Markus Straßer, Joachim Baumann, and Fritz Hohl. Mole – a Java based mobile agent system. In Max Mühlhäuser, editor, *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems: Agents on the Move, Linz (Austria), July 1996*, pages 28–35, Heidelberg, 1997. dpunkt Verlag. 19, 68, 73

Markus Straßer and Markus Schwehm. A performance model for mobile agent systems. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas (USA)*, volume 2, pages 1132–1140, Athens, GA, 1997. CSREA Press. 4, 25

Java Object Serialization Specification. Sun Microsystems Inc., July 1999. `http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serial-title.doc.html`. 57

Java Remote Method Invocation: Distributed Computing for Java. Sun Microsystems Inc., July 2002. `http://java.sun.com/marketing/collateral/javarmi.html`. 10, 76, 276

Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers, and Timothy S. Mitrovich. An Overview of the NOMADS Mobile Agent Systems. In Ciaran Bryce and Chrislain Razafimahefa, editors, *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages, Sophia Antipolis (France), June 2000*, 2000. Paper is only available online `http://cui.unige.ch/~ecoopws`. 70

Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 2nd edition, 2001. ISBN 0-130-31358-0. 48, 279

Éric Tanter, Michael Vernaillen, and José Piquer. Towards transparent adaption of migration policies. In *8th ECOOP Workshop on Mobile Object Systems: Agent Applications and New Frontiers, Malaga (Spain), June 2002*, 2002. Paper is only available online `http://cui.unige.ch/~ecoopws`. 110

Wolfgang Theilmann and Kurt Rothermel. Disseminating mobile agents for distributed information filtering. 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 152–161, Los Alamitos, CA, 1999. IEEE Computer Society Press. 39

Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997. 51

Giovanni Vigna, editor. *Mobile Agents and Securtiy*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1998a. 177

Giovanni Vigna. *Mobile Code Technologies, Paradigms, and Applications*. PhD thesis, Politecnico di Milano, February 1998b. 9, 24

Tim Walsh, Paddy Nixon, and Simon Dobson. As strong as possible mobility: An architecture for stateful object migration on the Internet. In Ciaran Bryce and Chrislain Razafimahefa, editors, *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages, Sophia Antipolis (France), June 2000*, 2000. Paper is only available online `http://cui.unige.ch/~ecoopws`. 70, 71

Xiaojin Wang, Jason Hallstrom, and Gerald Baumgartner. Reliability Through Strong Mobility. In *7th ECOOP Workshop on Mobile Object Systems: Development of Robust and High Confidence Agent Applications, Budapest (Hungary), June 2001*, 2001. Paper is only available online `http://cui.unige.ch/~ecoopws`. 70

James E. White. Mobile agents. In Bradshaw [1996], pages 437–472. ISBN 0-262-52234-9. 70

James E. White, C. S. Helgeson, and D. A. Steedman. System and method for distributed computation based upon the movement, execution, and interaction of processes in a network. US Patent 5.603.031, 11 February 1997. 18

David Wong, Noemi Paciorek, and Dana Moore. Java-based Mobile Agents. *Communications of the ACM*, 42(3):92–105, March 1998. 19

J. Zander and R. Forchheimer. Softnet – an approach to high level packet communication. In *Proc. of AMRAD Conference, San Francisco (USA), 1983*, 1983. 17

Michael Zapf and Kurt Geihs. What Type Is It? A Type System For Mobile Agents. In *Proceedings of Second International Symposium* From Agent Theory to Agent Implementation *(AT2AI-2) at the 15th European Meeting on Cybernetics and Systems Research (EMCSR 2000), Vienna (Austria), April 2000*, 2000. 48

A. The Mobile Agent System Tracy2

A.1. Introduction and History of Tracy

This appendix introduces the Java-based mobile agent system Tracy that has been developed by our team at FSU Jena during the last four years. In the following sections, we will give a brief overview of the main features of Tracy. We decided to move this description to an appendix as the development of Tracy was not part of this dissertation project.

When our project group started research in the area of mobile agents in 1998, no system available offered enough support for the main research issue we were interested in: migration of mobile agents. Our first research hypothesis [Braun, 1999a,b], was that the migration process of mobile agents should be optimized in order to increase mobile agents' performance as compared to the client-server-approach. All systems available at the end of year 1998 only provided a single and most often very simple migration technique. None of these systems could be adapted to the extent that would have been necessary to implement our new ideas concerning efficient and high-performance migration. Therefore, we decided to start the development of a new mobile agent system from scratch. It was not our goal to develop the n-th mobile agent system that is specialized to a specific research issue, but we wanted to build a system that could be used for real-world application development. Therefore, we made a considerable effort to provide services for agent communication, security of mobile agents, an easy-to-use graphical user interface and several other features.

The first implementation of Tracy was done by the author in the first half of the year 1999. The focus was to acquire first experiences with the implementation of the migration process of mobile agents. Christian Erfurth supplemented this first prototype within his diploma thesis [Erfurth, 1999] by services for agent communication and anonymous information exchange using a blackboard in the second half of 1999. Sven Geisenhainer developed a graphical user interface for Tracy within his diploma thesis [Geisenhainer, 2000] and several other students contributed parts of Tracy, for example Chris Fensch (SSL transmission and data compression), and Steffen Grumbach (technique to monitor internal processes of Tracy). With the completion of our first technical report about Tracy in September 2000 [Braun et al., 2000a], Tracy was published on the Web for public download.

In year 2000, Jan Eismann supplemented our team and he implemented the powerful concept of the Tracy domain manager [Braun et al., 2001a]. Several other new

features were implemented since then, as for example a multi-user concept and several security features. Steffen Schreiber implemented a technique to collect network performance information [Schreiber, 2002], which will be used to implement sophisticated migration strategies later. Heiko Peter implemented techniques for decision planning for mobile agents [Peter, 2002]. In 2002, Sven Geisenhainer ported Tracy so that it can be used as an Enterprise Java Bean (EJB) component as part of an Application Server. Jan Eismann ported Tracy to be executable on a PDA (Compaq iPAQ) and proofed that mobile agents can also migrate using the Bluetooth protocol.

The Tracy team had the honor to present their system at the CeBIT fair in Hannover, Germany in 2001 and 2002. For this occasion, we developed several demo applications, for example a prototype of a multi-user calendar application using mobile agents as information carrier, a tool for planning business trips, where mobile agents collect information using Web Services using the SOAP protocol, and a prototype for an agent-based application where agents monitor stock quotes and inform their owner about important changes. Tracy was evaluated to be used as major component of an electronic marketplace for mobile agents [Braun et al., 2002; Kowalczyk et al., 2002; Müller, 2002].

Since the beginning in 1999, in sum more than a dozen people have been member of the Tracy project. Several students made their diploma theses within the Tracy project. Since 2001, the Tracy team is member of the AgentLink European Network of Excellence in Agent Research[1] and an active member of the special interest group on mobile agents (sigma)[2].

A.2. Tracy Infrastructure

The Tracy *infrastructure* consists of several platforms, on each running a Tracy *agent server*, which creates the environment for running several kinds of agents and offers services for receiving and sending mobile agents over the network. Each agent server is independent from the other, even though they are able to communicate with each other. The architecture of the Tracy *system* is, therefore, the many times repeated architecture of the singular agent server, see Figure A.1.

The agent server sits on top of a Java virtual machine (JVM), which is itself based on top of an operating system. On top of a Tracy agent server there can execute one or many applications that use it to host application-specific agents. It is not necessary that there is a permanent connection between application and agent server. An application can start an agent server temporarily to launch agents that immediately migrate to another platform. In the same way, it is possible that an application only connects occasionally to a running agent server, e.g. to check, whether new agents

[1] See `http://www.agentlink.org` for more information.
[2] See `http://www.semoa.org/sigma` for more information.

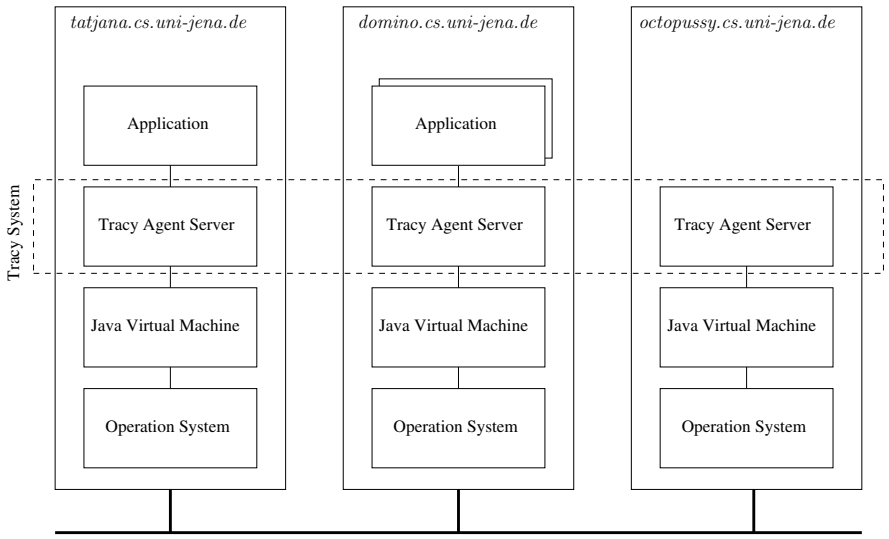


Figure A.1.: Example for a Tracy infrastructure consisting of three platforms.

have arrived. If there is no application connected to the agent server, then the agent server is able to offer services on its own. An agent server has a unique name which consists of the host name and the logical agent server name, e.g. *tatjana.cs.uni-jena.de/fortknox*.

It is possible to start multiple agent servers on a single platform by either starting multiple agent servers on one Java virtual machine, or starting several virtual machines each executing one agent server. To distinguish several agent servers on one platform, each server accepts incoming communication requests on a unique port. A Tracy agent server can be configured while launching to accept communication on arbitrary port numbers. However, in contrast to other agent systems, e.g. Aglets, in Tracy the programmer needs no knowledge about these communication port numbers. A Tracy *local name service* (LNS) associates all logical agent server names of one host with information about communication port numbers.

A.3. The Tracy Agent Model

A.3.1. Foundations

Our agent model consists of three types of agents. First, a *system agent* is a stationary agent that offers services related to the operating system on which the server runs, e.g. file services, printing services, etc. Second, a *gateway agent*, which is also station-

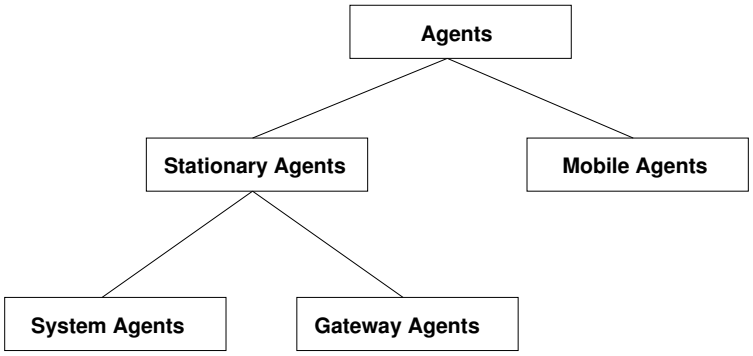


Figure A.2.: Classification of software agents in Tracy.

ary, is responsible for the communication to software components outside the Tracy architecture, e.g. legacy software, data base management systems, or even other mobile agent systems. The third class of agents are *mobile agents* that are characterized by the ability to migrate to other platforms, but have none of the above mentioned permissions. Mobile agents are strictly controlled by the agent server, so that it is impossible for mobile agents to access the underlying operating system or external software components on other than its home platform.

In Tracy each agent has a globally, i.e. within the Tracy system, unique *name* as identifier and a *home platform* on which it was created. Both do not change after the agent was initialized. The creation can be initiated either from outside the Tracy agent server, e.g. using the TracyAPI, or the graphical user interface or from inside the server, e.g. by another agent. The execution of an agent can be suspended, i.e. stopped temporarily, and resumed, i.e. started again. An agent can be asked to quit itself, or can be killed by a user (not by other agents), e.g. if a malicious agent consumes system resources. Mobile agents can migrate to other Tracy agent servers, see Section A.3.4. Agents can communicate with each other either by using asynchronous messages, or by leaving information on a blackboard, see Section A.3.3.

One of the main differences to other mobile agent systems is the fact that Tracy does not support any kind of remote communication, i.e. you cannot send messages to an agent on another agent server. This restriction comes from our interpretation of mobile agents: an agent must move to the destination platform, if it wants to communicate to other agents over there.

A.3.2. Accessing the Host via System and Gateway Agents

The distinction between mobile and stationary agents is mainly a result of security problems. Because of the ability to migrate, mobile agents are considered to be

an insecure part of the agent system. So we cannot grant unrestricted access to the host. System and gateway agents are stationary agents which are not able to migrate. Therefore, these agents are considered to be secure and they may access the host. If mobile agents need to access the host, we use stationary agents as a dynamic interface and as a security wall. They act as so called system agents. If mobile agents must be able to use local applications to solve their tasks, they may access them using gateway agents. This kind of stationary agent acts again as a dynamic interface and a security wall.

However, security is not the only aspect to be considered: gateway agents connect local applications to the agent server. The application *speaks* with the agent server via a gateway agent and other agents may *speak* with the application via a gateway agent, as well. Thus, gateway agents act also access filters for the associated application. In Figure A.3 the various types of available agents are illustrated. Stationary agents expand the possibilities of the agent server. Local applications can be connected with the agent server and so the features of the applications can be joined with the possibilities of the agent server. In addition, the possibilities of the agent server can be expanded by system agents, e.g. a specific (virtual) infrastructure could be created.

A.3.3. Communication between Agents

Mobile agents are a design paradigm for distributed computing, in which a mobile agent migrates to another platform to fulfill a user-defined task. This task can be done better at the remote platform or can be done only at the remote platform. The agent needs to connect to operating system services, to a data base, or to another local running application at the (remote) platform. In the last section we have discussed that these services are integrated in the agent server via stationary agents. In addition, the agent server can provide more services by simply running more stationary agents. Thus, the mobile agent must be able to communicate with system and gateway agents to do its job and to get access to any information.

In Tracy, agents may communicate with each other directly. This is done using asynchronous messages and not via direct method calls between agent objects. The latter effects an agent in a direct way which is a contradiction to the concept of agent autonomy. Thus, in Tracy, messages which will be exchanged between agents are like mails. All of the three types of agents can exchange these mail-messages. Every agent has a mailbox in which new mails are stored. The agent can decide on its own how to handle these mails. It can decide to accept mails or not by closing its mailbox (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.

The reader could have the impression that we provide also remote communication. However, Tracy does not support any kind of remote communication, i.e. an agent

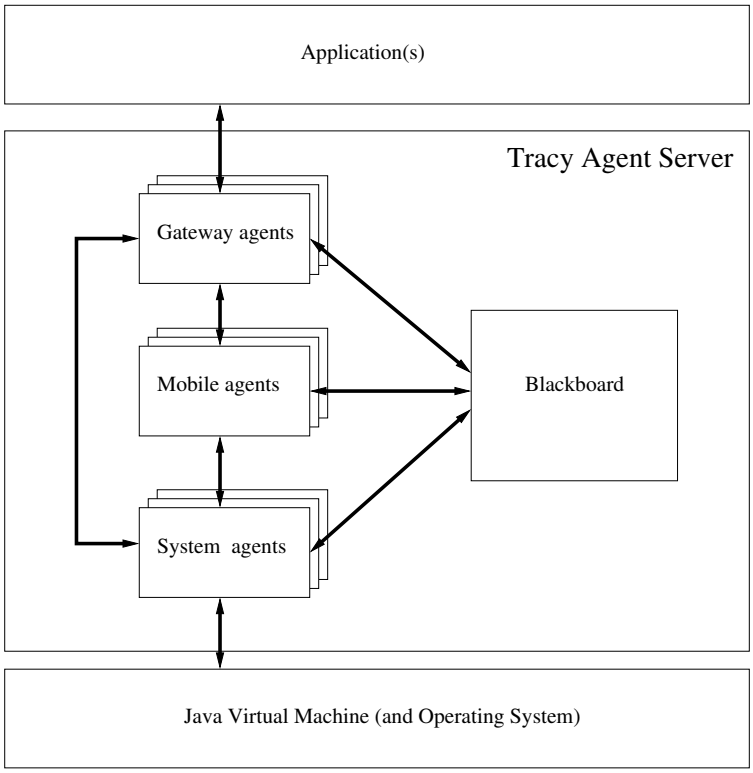


Figure A.3.: Architecture of a Tracy server from the programmer's point of view.

cannot send messages to another agent residing on a different agent server. Even if both agent servers were to reside on the same host sending mails between them would not be possible. This restriction is a result from our interpretation of mobile agents: an agent must move to the destination platform, if it wants to communicate to other agents over there. Remember that local applications can send and receive messages via gateway agents only. Another feature is that the user can send messages to agents by using the TracyGUI.

Agents can communicate with each other not only by using asynchronous messages. Communication is also possible by leaving information on a blackboard via an interface integrated in the Agent Manager (see Figure A.3). In this second kind of communication, which is an indirect one, the agent puts some information on the blackboard like an announcement in a newspaper. Other agents can read this announced information from the blackboard via a symbolic name. We think the blackboard is another basic approach to provide information deposited by an agent, by the agent server, or by an application. For example, the agent server deposits in-

formation on the blackboard about services provided by the server. The blackboard is a mean to provide persistent data, too.

The blackboard itself is organized like a file system – there are directories and files. A directory is a container for files and other directories, where as files can only contain data of some specific types, like plain text, XML, HTML, graphic files, etc. Each entity might have an owner who defines grants to other agents to read or write this item.

Besides reading and writing blackboard entities, directories and files can be observed by agents. An agent will be informed about any change of the observed entity by a message. An agent can become active when a specific blackboard entity has changed its value. Other events are adding and deleting blackboard entities. Remember that local applications can only write and read information from the blackboard via gateway agents.

Based on these two kinds of communication, by messages and with the use of the blackboard, we have a loose coupling between agents, between agents and the agent server, and between agents and associated applications. The user (via the graphical user interface), or an associated application can use messages to communicate with agents and so agents can also receive commands. There are also broadcast messages which can be sent by the GUI for administrative purposes and by the agent server for system state propagation, e.g. send a system failure using a broadcast message.

A.3.4. Agent Migration

When a mobile agent wants to migrate to another platform, the underlying mobile agent system (MAS) is responsible for marshaling of code and state information that must be transmitted to the destination platform. Normally, the state consists of the program counter, the value of all variables, and the call stack. The MAS on the destination platform has to unmarshal this package and start the agent. As we have seen in Chapter 5 current mobile agent systems offer various ways to migrate mobile agents.

The Tracy mobility model bases on the Kalong resp. the MDL component presented in Chapter 7 on page 113. It gives the programmer more influence on the migration strategy, and even the transmission strategy the agent should take. In particular, it allows the programmer *and* the agent to modify the migration strategy according to local circumstances. A sophisticated migration strategy based on our model would be in a position to transmit all agent's code when migrating outside a fire wall, and transmit only necessary code when the home platform can be reached easily.

In our model, only the mobile agent itself can initiate the migration process by invoking one of the `go`-commands, which we will explain later. To provide some kind of agent server initiated migration, we include the following concept: each agent has the ability to react to incoming messages, dispatched from other agents or the agent

server itself. A mobile agent may receive a message with the suggestion to leave the platform, e.g. in case of system failure. It depends on the agent's programmer, whether he cares about those messages or not. Only the agent server is allowed to send such migration invitations.

From the programmer's viewpoint we currently offer a weak form of mobility, in which an agent can start a migration by a `go`-command that is parameterized by the name of the destination platform and the name of the method that should be invoked next. A mobile agent can use one of the following commands to start the migration process:

1. `go( destination, method )`, as described above
2. `go_back( method )`, initiates a migration back to the last platform the agent came from, and
3. `go_home( method )`, initiates a migration back to its home platform.

As already said, the mobile agent can also influence the migration strategy, and the transmission strategy that should be used for the next transfer. Both can be declared by optional parameters of the `go`-commands, e.g. `go( destination, method, "pull-per-unit")` to choose the pull-per-unit migration strategy. A mobile agent can define a *default migration strategy* and a *default transmission strategy* that will be applied for each migration. If the agent could not be transmitted successfully to the destination platform, the agent is reactivated on the residing platform and a pre-defined method with name `migrationFailed` is invoked.

A.3.5. Agent Security

Agent security was of major interest for the implementation of the Tracy agent server and many security related services are already provided. However, Tracy is not a security-centric mobile agent system, as for example Semoa, but we are on the way to improve Tracy to solve some open problems.

Security is implemented as part of several components of a Tracy server and we will only give a brief overview on the problems we have faced so far.

First of all, agents are protected against each other on the level of Java language. In Tracy, each agent has its own class loader and its own thread group, and Tracy prevents agents to have references to each other as far as possible. To protect the agent server and the underlying host against malicious agents, we use the Java sandbox technique. Permissions are granted on the base of the agent's owner information and the last visited agency, and can be modified dynamically.

Services that are provided to protect agents code and data were already described in Section 9.1.3. The MDL migration component offers several extensions on top of

Kalong to protect data items so that they are only readable at selected agent servers or not modifiable at all. An agent's immutable state and its code is digitally signed with the agent owner's private key. Each agency must digitally sign the complete agent before sending it to the next destination. The migration component inspects agent code with regard to probably malicious code sequences. On the level of network transmission, Tracy offers to send all agent information using the SSL protocol.

A.4. The Architecture of a Tracy System

We will now have a look at the overall architecture of an agent-based system. An agent-based system is a host that uses a Tracy server as a substantial component to offer services to external agents, or consumes services of other agent servers using their own mobile agents. Different types of agent-based systems are distinguished by the way the agent server is connected to other applications on that host. In Figure A.4 we see five different configuration types for an agent-based system. The dashed-line box stands for a host on which a Tracy agent server is running. We omit to depict the Java virtual machine and the Operating System. All hosts are connected using a network.

The first example (*fortknox*) shows a host running only a Tracy agent server. In this case, the agent server must be able to offer services on its own, either by system agents that were started along with the agent server, or by a blackboard that was initialized from a connected data base. The complexity of services that such an agent server can offer depends on the available system agents.

An agent server may run stand-alone for its whole life-time, but can also be temporarily connected to a graphical user interface, as shown in the second example (*palmyra*). The graphical user interface to control an agent server is named TracyGUI. It can be started along with the agent server, but might also be closed in the mean time and relaunched again later. Using TracyGUI an agent server can be completely monitored and controlled: All agents' activities can be viewed, new agents can be started, existing agents can be killed, and the blackboard can be modified.

The graphical user interface always tries to depict an almost real image of the underlying agent server. As a consequence, each arriving agent, each console message that an agent wants to print, and each blackboard modification is visible to the user *almost* in real-time. Unfortunately, it is a very expensive task to keep a graphical user interface up to date, and it slows down the whole agent server noticeable. As a consequence, a graphical user interface should only be executed when the user wants to control or view the agent server explicitly.

The remaining three examples in Figure A.4 show hosts on which an application uses an agent server to offer services to foreign agents, or use services of other agent servers using their own mobile agents. It is the privilege of applications to start

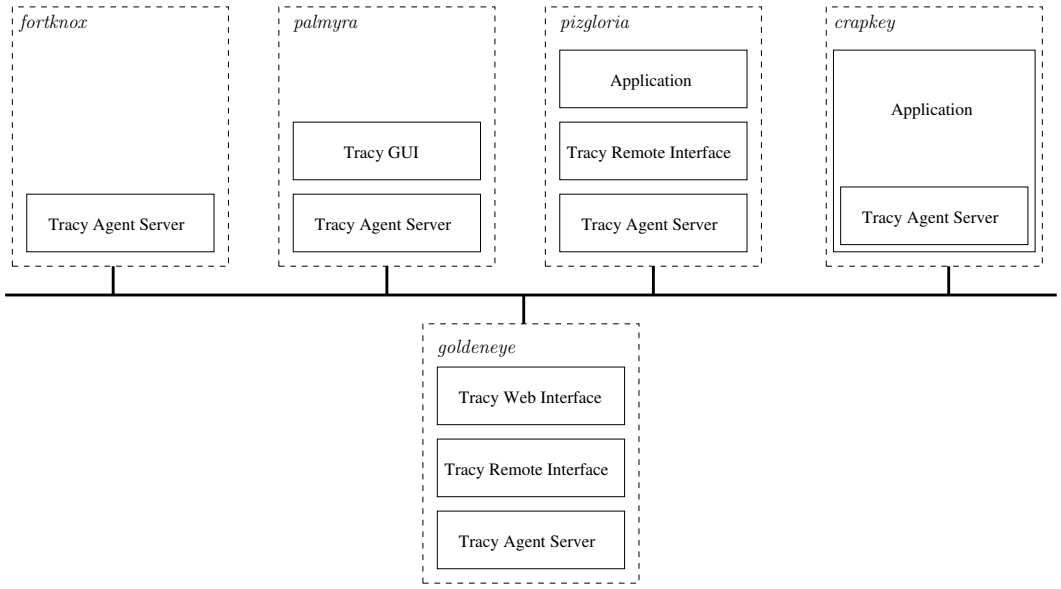


Figure A.4.: Five configuration types of an agent-based system.

gateway agents (see Section A.3) that are able to offer services within the agent server and direct requests to the connected application.

In the first case (*pizgloria*), there is an application written in the Java programming language that accesses a Tracy agent server using the Java Remote Method Invocation (RMI) technique [Sun, 2002]. In this case, application and Tracy agent server are loosely coupled – the only connection is established by method calls using the Tracy Remote Interface. This interface defines methods to control and monitor an agent server to other Java-based applications. In the simplest case the application resides on the same host as the agent server. However, it is also possible that application and agent server reside on different hosts (not pictured in Figure A.4). To protect an agent server against applications, basic security checks are already implemented.

In the second case (*crapkey*), the coupling between these components is very strong. In this case the Tracy agent server is an embedded software component within an Java-based application. The application uses the TracyAPI to control the agent server. Additionally, Tracy can also be used as Enterprise Java Bean within an application server.

The fifth example (*goldeneye*) in Figure A.4 shows a type of configuration where on top of the Tracy Remote Interface there is a component, called Tracy Web Server, that offers a Web interface for a Tracy agent server to a user. By using this Web interface a user can do some control and monitoring actions but has no full access, as

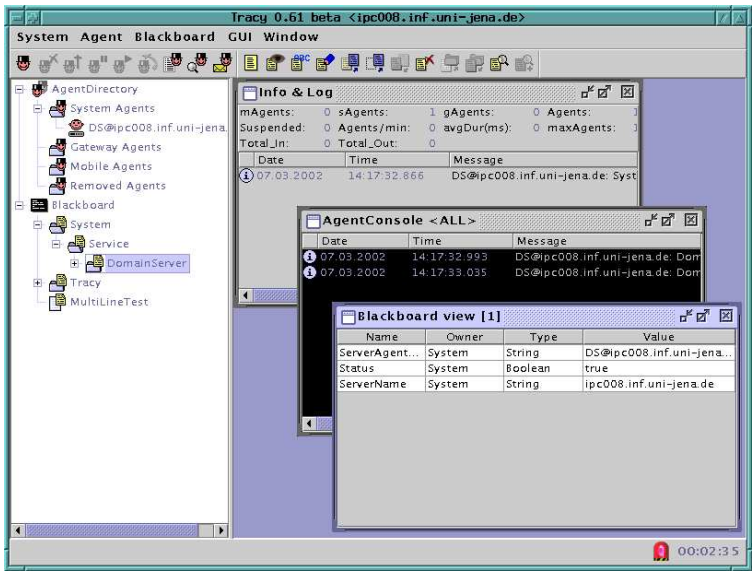


Figure A.5.: The Tracy graphical user interface.

opposed to the graphical user interface described above. The user can start agents, view the agent's control messages, and kill agents. However, the user has no rights to view or modify the agent server's blackboard. The Tracy Web Interface is able to handle multiple users in parallel and offers each of them an own *private agent space*. Of course, it is not allowed to control or even view agents that are owned by other users.

A.5. Managing Tracy Networks

The Tracy Domain Manager Service is an approach to construct and evolve a network of mobile agent servers. It can be seen as a service that is indispensable for mobile agents to move through the network automatically. Without such a service the programmer of a mobile agent must code the agent's itinerary into its business logic.

The basic concept we employ is that of a *logical agent server network*. We define a logical network as an undirected graph in which vertices represent agent servers and an edge exists between a pair of vertices if there is the possibility to transmit mobile agents between the corresponding servers. Not all agent servers must be able to exchange mobile agents due to different transmission protocols, firewalls or private subnetworks that are only reachable via a gateway server. A logical network is a necessary prerequisite for a mobile agent to move through the network automatically.

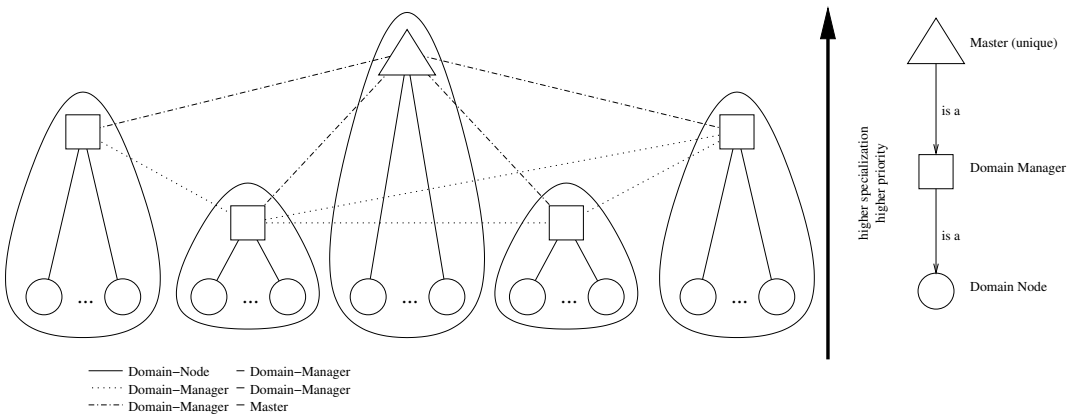


Figure A.6.: Topology of our logical agent server network. An edge between a pair of vertices indicates that the corresponding agent servers know each other.

On each server it can ask a stationary agent, or a service, for the neighboring agent servers and decide to which it will migrate to next. Without such a network service the agent's programmer has the obligation to code the agent's itinerary into its business logic. While this is sufficient in some applications and in small networks, it is not reasonable to define an agent's route in a world wide network, for example. In such an environment mobile agents must be able to define their itinerary on their own. They must be in a position to react to unreliable network connections and unreliable agent servers and, therefore, possibly modify their itinerary on the fly.

Our approach has a two-level structure, where agent servers within a subnetwork are combined into a domain, which is limited to subnetworks. All agent servers within a single domain enlist at a central server, which is called *domain manager*, compare Figure A.6. Domains can be connected to each other so that mobile agents can also reach agent servers in other domains. Connecting and disconnecting of agent servers to the network works fully automatic and dynamic. Our approach is multi-agent based, i.e. several stationary and mobile agents communicate to each other to build and evolve the logical network. It does not depend on any specific mobile agent system. Although we have implemented our approach on top of our mobile agent system Tracy it is designed to be portable to any other mobile agent system with minimal effort.

Main characteristics of our approach are its robustness in failure situations and its high performance, which is shown by results of a first evaluation [Braun et al., 2001a]. For example we can guarantee that at any time there exists a domain manager for each domain. If a domain manager crashes (because its host agent server crashes) all

remaining agent servers vote for becoming the new domain manager. If the original domain manager is relaunched, it can reclaim this role.

Currently, in all other mobile agent systems, it is only possible to manage a single *stand-alone* agent server by using some kind of console or graphical user interface. In a logical agent server network, the administrator can obtain an immediate view of all agent servers. In the Tracy system there is already an approach to use this information in combination with its graphical user interface which can be dynamically connected to other running agent servers to administrate them, e.g. to start and stop agents or just for checking the status of the server.

A.6. Tracy's Software Architecture

Our first implementation of a Tracy server was a three layer architecture. The upper layer contained the core agent server functionality and all services as for example agent communication and the blackboard as described in Section A.3 on page 269. The intermediate layer was responsible for the agent migration process, and the lowest layer implements several network transmission protocols [Braun et al., 2000a].

The main problem with this architecture is that all services for agents are static part of the upper layer. It exists a basic class *Agent* that contains methods to access all these services, e.g. to communicate to other agents, to access the central blackboard service, or to initialize the migration process in case of mobile agents. This has the consequence that it means considerable effort to add new services to this architecture, as it is necessary to adapt class *Agent* for each new service. The inflexible structure of services prevents dynamic installation of new services during run-time. Another problem with this first architecture is that the implementation of the migration model is hard-wired in the same way and it is not possible to provide more than one migration model in parallel in a single agent server.

Therefore, we decided to re-design the Tracy server architecture to make it more flexible, compare Figure A.7. The architecture of a Tracy2 server is comparable to a micro-kernel architecture [Buschmann et al., 1996] as used in several operating systems [Tanenbaum, 2001]. This micro-kernel provides only few basic services indispensable for an agent-based system. This micro-kernel only requires functions of a Java virtual machine in version 1.2, which make it very easy to port it to limited mobile devices.

The main function of the micro-kernel is to provide a thread pool to execute agents and control their life-cycle. A thread-pool is a data structure that manages several Java thread objects and assigns an already running thread to an agent that is going to be executed. This technique significantly increases the overall performance of agent execution. The thread-pool is secure in so far as agents have no chance to access and tamper with thread objects of other agents currently executed. From the

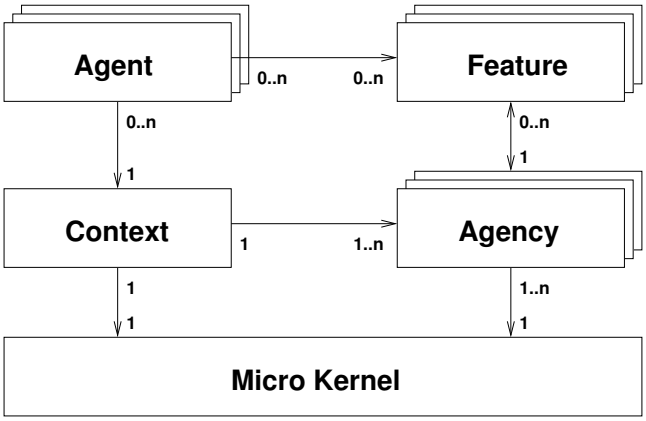


Figure A.7.: Software architecture of Tracy2.

micro-kernel's view, the only requirement for an agent is that it implements the basic Java interface *Runnable*. The agent's life-cycle consists of only two states: *Running* and *Waiting*. The latter state means for an agent that it is not currently executed (especially it does not have an associated thread) and currently waiting to become active again, for example by receiving a message.

On top of the micro-kernel there is an *agency* that manages all agents that are currently residing at this platform using an *agent directory* independent, whether they are currently running or waiting. The agency provides basic functions for agents to inform about their environment, for example to find other agents on this platform. The agency offers functions to start and stop agents and uses the micro-kernel for these tasks. Each agent is registered with the agency when it is started. It makes no differences, whether the agent is started locally or has migrated to this agent server. The agent directory entry exists for the whole life-time on the local platform, even when the agent is waiting. It is deleted when the agent leaves the server by migration or when the agent is killed.

Additionally, an agency has the task to manage the so-called *features*, which is our concept to provide services within Tracy2. For each service, as for example inter-agent communication, blackboard, migration, etc., there exists a single feature component. Features can be added dynamically to an running agent server, they can be stopped (for example in case of an error) and restarted later. Features must be registered with the agency under a user-defined name, which equals the name by which an agent (or other features) can access this feature later.

The consequence of this very flexible concept is that an agent (which still is any object of type *Runnable*) must use a new technique to access features. As there does

not exist a basic class for agents that provides methods to access each feature, we use the concept of so-called *context* objects. To communicate to a single feature, an agent must request its context object by calling a static method of class *Context* using the name of the feature. The following example shows an agent that in line 11 requests its context object for the message features (whose name is *msg*). In line 16 a message is sent to an agent whose name is *Goldfinger*. The example does not show how to receive messages from other agents.

```
1  import org.taf.tracy2.Context;
2  import org.taf.tracy2.features.message.*;
3
4  public class SampleAgent implements Runnable
5  {
6      protected IAgentMessageContext amc;
7
8      public void run()
9      {
10         if( amc == null ) {
11             amc = (IAgentMessageContext)Context.getContext( "msg" );
12             amc.openMailBox();
13         }
14
15         Message message = new Message( "Goldfinger", "offer", "gold_bar" );
16         amc.sendMail( recipientName, message );
17     }
18 }
```

The migration feature uses the Kalong migration component as described in Section 9.2.2 on page 193.

This basic architecture only provides a single agency. As an extension it is also possible to construct a multi-agency agent server, where two or more separated agencies exist in parallel. Each agency has its own set of features and all agents and all features of a single agency are strictly separated from those at other agencies. However, all agencies still run within the same Java virtual machine and on top of the same micro-kernel, which makes it possible to execute two different agencies on a single computer system in parallel. The motivation of a multi-agency server is to separate different agent-based applications in a very strict sense.

From the programmer's point of view, the new Tracy2 design is compatible to the first version of Tracy, which makes it very easy to migrate to the new version of Tracy. No agents must be modified, but they can be reused without any changes. However, they must be compiled again, because basic classes have changed. Of course, these agents cannot benefit from new services that are implemented for the Tracy2 agent

server.

Lebenslauf

1976–1980	Besuch der Grundschule in Solingen.
1980–1990	Besuch des Gymnasiums (August-Dicke-Schule) in Solingen.
Mai 1990	Erwerb der allgemeinen Hochschulreife.
1990–1995	Studium des Faches Informatik mit Nebenfach Betriebswirtschaftslehre an der Universität Dortmund. Abschluss mit dem Vordiplom Informatik (Prädikat „gut“).
1995–1998	Studium des Faches Informatik mit Nebenfach Wirtschaftswissenschaften an der Friedrich-Schiller-Universität Jena.
März/April 1997	Studienaufenthalt am Insitut National de Recherche en Informatique et en Automatique (INRIA), Rocquencourt (Frankreich).
Februar 1998	Diplomierung im Fach Informatik mit Nebenfach Wirtschaftswissenschaften an der Friedrich-Schiller-Universität Jena (Prädikat „sehr gut“).
1998–2003	Wissenschaftlicher Mitarbeiter bei Prof. Dr. Wilhelm R. Rossak am Lehrstuhl für Softwaretechnik der Friedrich-Schiller-Universität Jena.

Jena, 03. Feburar 2003

Index