An Introduction to the Tracy Mobile Agent System

TECHNICAL REPORT
No. Math/Inf/00/24

Peter Braun, Christian Erfurth, Wilhelm Rossak
Friedrich Schiller University Jena
Computer Science Department
Ernst-Abbe-Platz 1-4
D-07743 Jena, Germany
Tel. +49 (0) 36 41 / 94 63 31
Fax. +49 (0) 36 41 / 94 63 02
E-Mail: P.Braun@computer.org

Friedrich Schiller University Jena
Computer Science Department
Contents

Abstract i

Preface iii

1 Introduction 1

I Tracy Programming Concepts 5

2 The Tracy Agent Model 7
  2.1 Tracy Infrastructure ..................................... 7
  2.2 Agent Model ............................................... 8
  2.3 Accessing the Host System via Service and Gateway Agents .......... 9
  2.4 Communication between Agents ........................... 9
  2.5 Agent Migration ........................................... 11
    2.5.1 Strong and Weak Mobility ........................... 11
    2.5.2 Migration Strategy .................................. 12
    2.5.3 Transmission Strategy ............................... 14
  2.6 Agent Mobility Models ................................... 14
    2.6.1 Introduction ....................................... 14
    2.6.2 Example: Aglets Mobility Model ....................... 14
    2.6.3 The Tracy Mobility Model ........................... 15
  2.7 Security ................................................ 17

3 Programming Tracy Agents 19
## II  Inside Tracy – Design and Implementation

### 7  Architecture of a Tracy Agent Server

- **7.1 Agent System Layer** .................................... 61
- **7.2 Package Manager Layer** ................................ 63
- **7.3 Net Layer** .................................................... 63
- **7.4 Monitor** ..................................................... 64

### 8  Implementation Details

- **8.1 Layer Architecture and Connectivity** ..................... 65
- **8.2 Agent System Layer** .................................... 67
  - **8.2.1 Using the Agent System Layer API** ..................... 67
  - **8.2.2 Being an Agent System Layer Listener** ................. 70
  - **8.2.3 Message Queue Disposer** ................................ 71
  - **8.2.4 Blackboard** ........................................... 71
  - **8.2.5 Agent Execution** ..................................... 71
- **8.3 Package Manager Layer** ................................ 72
  - **8.3.1 Using the Package Manager Layer API** ................. 72
  - **8.3.2 Being a Package Manager Layer Listener** ............... 74
  - **8.3.3 Agent Package Manager** ................................ 74
  - **8.3.4 Agent Migration** ...................................... 80
  - **8.3.5 Migration Definition Language** ......................... 81
  - **8.3.6 Implementing Migration Strategies** .................... 84
  - **8.3.7 Receiving Net Layer Events** ............................ 86
- **8.4 Net Layer** .................................................... 86
  - **8.4.1 Using the Net Layer API** ............................... 87
  - **8.4.2 Being a Net Layer Listener** ............................ 88
  - **8.4.3 Simple Agent Transfer Protocol Layer** ................. 89
  - **8.4.4 Connection Layer** ..................................... 90
  - **8.4.5 Implementing new Transmission Strategies** ............ 91
- **8.5 Monitors for Viewing and Debugging** ...................... 92
  - **8.5.1 Programming Interface** ................................ 92
CONTENTS

8.5.2 Design ........................................... 93
8.5.3 Programming Client Monitors ....................... 94

9 Conclusions and Future Work 97

List of Figures 99

References 101
Abstract

This report describes Tracy, a mobile agent system that is currently under development at Friedrich Schiller University Jena. Tracy is a general-purpose agent system written in the Java 2 programming language. Thus, Tracy defines all basic features, i.e. starting and controlling of agents, migration of mobile agents, and communication between agents on the basis of asynchronous messages and by using blackboards. Tracy also contains first plain security mechanisms.

The scope of Tracy is twofold. On the one hand, it is a mobile agent system that can be used for real-world application development on the basis of mobile agents. A Tracy agent server can be used in multiple environments, e.g. as a single stand-alone server, or as an embedded component within an Java-based application. A graphical user interface makes it convenient for the user to start and control agents.

On the other hand, Tracy was developed to lay the foundation for our main agent-related research topic, which is the aspect of agent mobility and the challenge of migration optimization. The main difference of Tracy as compared to other mobile agent systems is that it provides a migration model that offers a flexible alternative to a pure agent model and a pure network transmission model. From the agent’s view it means that the agent server offers a multitude of different migration strategies the agent can choose from. From the researcher’s point of view it means that the complete process of migrating an agent is accessible to the programmer and can be adapted, so that new migration techniques can be easily implemented.

This report gives an introduction to both aspects of Tracy. In the first part, we show how to use and configure Tracy agent servers and we give a comprehensive introduction in all aspects related to agent development. In the second part, we concentrate on implementation aspects of Tracy. After we gave an overview of the software architecture of Tracy we guide the readers in adapting Tracy to their requirements. Especially, we concentrate on implementing new migration strategies.
Preface

This technical report contains the first documentation of a mobile agent system named Tracy that is under development at the Computer Science Department of Friedrich Schiller University Jena (FSU). We will show the basic principles of our mobile agent system, describe how to program agents, and describe how to build up a logical network of Tracy agent servers. We give a general idea of design and implementation aspects and guide the reader to implement own migration strategies. Tracy is a system which we are still correcting and improving. Therefore, this report can only capture an interim state of Tracy. Several developments are already scheduled for an upcoming version and we added outlooks to these improvements where necessary.

The aim of this report is not to introduce the reader to the field of mobile agents in general. Therefore, to benefit from this report most, the reader should already be familiar with the foundations of this rather new research topic. Unfortunately, there is no single book to which we could point the reader to obtain all necessary basics. We endeavored to add helpful links to overview papers in the first chapter of this report. The second requirement that the reader should meet is fluency in the Java programming language. Our mobile agent system, as almost all systems developed since 1997, is written in Java. However, Tracy seems to be one of the first systems that are developed under Java 2 and is also compatible to the newest release of the Java Virtual Machine, that is version 1.3. We expect the reader to have knowledge about the fundamentals of Java and some higher features like object serialization, remote method invocation, and network programming.

We welcome your feedback about this report and, of course, about Tracy, especially if you find errors or omissions. We installed a mailing list as a means of communication between Tracy developers. The address is mobility@informatik.uni-jena.de. If you want to register, please send an email to mobility-request@informatik.uni-jena.de with the word subscribe in the mail body and the subject line left blank. If you want to contact the authors of Tracy directly, please send an email to P.Braun@computer.org. We have also installed a Web site from which you can obtain recent information. The address is tracy.informatik.uni-jena.de.

At last, we would like to thank all members of the Tracy team at FSU Jena, without whom Tracy would not have been implemented in this way, yet. They include, in alphabetical order, Sven Geisenhainer, who implemented the graphical user interface within his diploma thesis, Steffen Grumbach, who implemented all monitor functionality, and Steffen Schlötzer, whose tool ByCAL for Java bytecode analyzing and transformation was very useful.
Chapter 1

Introduction

Mobile agent systems are a rapidly growing area of research. A mobile agent is a program that can migrate from host to host in a network of heterogeneous computer systems and fulfill a task specified by its owner. It works autonomously and communicates with other agents and host systems. During the self-initiated migration, the agent carries all its code and the complete execution state with it. Mobile agent systems build the environment in which mobile agents can exist. This technical report describes a new mobile agent system, named Tracy which is under development at Friedrich Schiller University Jena.

The research area of mobile agents has branched off research on software agents. A software agent is a software entity which continuously performs tasks given by a user within a particular restricted environment. Research on software agents was started in the area of Artificial Intelligence, already in the 1970s. We should emphasize that the term agent is used in very different ways. Actually, several papers just deal with the problem to define the word agent at all [8, 9]. The enumeration of all characteristics that a program should fulfill to become an agent is not in our intention, here. However, the most important properties seem to be autonomy, i.e. the capability to act proactive and goal directed, and adaptation, i.e. being able to learn and improve its behavior. A comprehensive overview of such characteristics is for example given by Bradshaw [4, pp. 8ff]. Good overview papers for software agents from the view of artificial intelligence are given by Jennings and Wooldridge [41, 42]. Another paper that can serve as an introduction to wider fields of software agents is the work of Green and Somers [15]. The authors give a good overview to the various areas of agent research, and they also mention mobile agents, already. Additionally, we would point the interested reader to two books that might serve as comprehensive introductions to all fields of software agent research and their possible application domains. The first book, edited by Bradshaw [5], contains a collection of papers concerning the areas of agent-supported user interfaces, agents for learning and intelligent assistance, and, at last, a summary of agent communication, collaboration, and mobility. Second, the book by Chorafas [6] also gives an introduction to many agent-related topics. In addition, this book contains several links to real-world projects. It describes possible application areas for software agents. A part of its own is dedicated to application areas in business, like for example information filtering agents.

Since about five years, there is a separate research branch working on mobile agents. The first mobile agent system was Telescript [39] developed by General Magic. In this domain of mobile software agents, i.e. software agents that are able to move somehow, all aspects related to code mobility are analyzed. 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 also have a rather pragmatic notion of the term mobile agent. To our understanding, it is any kind of software entity that is able to initiate a migration on its own within a network of heterogeneous computer systems. In
the last years, research and development of mobile agents are very fast growing research areas. However, it is still in its infancy. Most projects conducted so far had the goal to proof the possibility to implement mobile code, which is the basis of mobile agents, in different programming languages and show that it is worth using mobile agents for particular application domains. Most research was done in the area of mobile code security.

Some of the mobile agent systems developed in the last years are Aglets [20] by IBM, Voyager [24,25] by ObjectSpace, and Concordia [23] by Mitsubishi. Systems developed for university-based research are for example Mole [33], Discovery [21], D’Agents [14], Ara [26], and MAP [29]. See Kiniry and Zimmermann [17], and Wong et al. [40] for a comprehensive review of Java-based mobile agent systems. Research in the area of mobile agents looked at languages that are suitable for mobile agent programming [7,19], and languages for agent communication [2]. Very much effort was put into security issues [37], control issues [3,31], and design issues [16]. Several prototypes of real-world applications in the area of information retrieval, management of distributed systems, and mobile computing are in development [28,30]. Oddly enough, none of the above mentioned mobile agent systems and no research group has focused on the inherent topic of mobile agents, that is migration. Today, each mobile agent system has developed its own understanding and notion of terms like code mobility and state migration. Almost no work has been done to define these terms and classify concepts concerning mobile agent migration.

The main research topic related to mobile agents at FSU Jena are mobility aspects. We work on a wide area of mobility-related problems, ranging from reordering an agent’s route to faster fulfill a user given task, to optimizing the migration strategy, i.e. how code and data are transmitted via the network. We also work on better transmission techniques, e.g. using extensive code compression. One of our main ideas is that mobile agents must be able to influence the migration process to be able to adapt to changing requirements, e.g. network parameters. To support our research, none of the existing prototypes were useful, because in these systems the process of code migration is not open to the programmer. In most systems which are implemented in Java code migration is implemented using standard Java techniques. Therefore, we developed our own mobile agent system, named Tracy, in which we can change and configure almost all migration related aspects. However, Tracy is a general-purpose mobile agent system, i.e. it serves as a foundation of both research and application development in the field of mobile agents. Our agent system gives all necessary tools needed to develop applications using mobile agents. Agents can be created, controlled, and suspended. They can communicate with each other by asynchronous messages and use a blackboard. Especially, Tracy defines the environment to migrate agents between platforms within a logical Tracy network.

In this report we give an introduction to programming aspects of our agent system, and using it for research on migration related topics. Therefore, we divided this report in two parts. In the first part, we describe Tracy for those who want to use it for their own development of mobile agents. As already said, Tracy is a general-purpose mobile agent system and can therefore be used for any kind of application development and research on mobile agent topics. After we gave an overview of our agent model in Ch. 2 on page 7, we will show how to program software agents and especially mobile agents in Tracy. In Ch. 4 on page 39 we give a concise overview of the general architecture of an agent-based system. In this chapter we show various environments in which Tracy can be used. In the following chapter we will describe how a single agent server can be managed, especially by a graphical user interface we developed on top of Tracy. The first part of this report concludes by a detailed description of the Tracy application programming interface. Using TracyAPI makes it possible to use a Tracy agent server as a black-box component within an application. The TracyAPI gives full control over all necessary services of an agent server. We show how to control and maintain it.

The second part of this report gives detailed information on how we implemented Tracy. After a
brief introduction in the general software architecture of a Tracy agent server in Ch. 7 on page 61, we describe in detail each of the three layers of the Tracy architecture in Ch. 8 on page 65. After this chapter the reader will be able to modify the agent system according to his/her wishes. Again, we emphasize the migration aspect. We introduce how to implement new migration strategies and transmission strategies. Finally, Ch. 9 on page 97 gives an outlook to further development.
Part I

Tracy Programming Concepts
Chapter 2

The Tracy Agent Model

In this chapter we introduce the mobile agent system Tracy. We show concepts and models, and give definitions of important terms, but postpone a detailed introduction to programming agents by source code examples to Ch. 3 on page 19. First, we show basics of a Tracy infrastructure and present our model of Tracy agents. Then we describe our communication model that consists of two types of messages. After that, we present a comprehensive description of terms related to agent mobility and propose our code mobility model. We compare it concisely to the Aglets model. Last, we summarize our security model which can be only understand as first step to protect an agent server against malicious agents, but is by no means a sufficient solution to the very important problem of malicious agent servers.

2.1 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 agent server, see Fig. 2.1 on the following page.

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 be 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 have arrived. If there is no application connected to the agent server, then the agent server must be able to offer services on its own. A platform has a unique name, normally the host name, e.g. dagobert.uni-jena.de.

It is possible to start multiple agent servers on one platform by either starting multiple agent servers on one Java Virtual Machine, or starting several Virtual Machines each executing one agent server. In the current version of Tracy both scenarios lead to the following problem. Due to the lack of an own name, an agent server cannot be addressed individually. To migrate a mobile agent to a destination agent server, the name of the platform hosting the agent server must be known. To distinguish several agent servers on one platform, additional information, e.g. about the receiving TCP port, must be applied. Although this is state-of-the-art technique also in other
agent systems, it is not convenient at all. In the next version of Tracy it will be possible to start multiple agent servers, each executed on one Virtual Machine, on one platform and distinguish them via a name, especially during migration. All necessary additional information about TCP ports etc. will be transparent to the user.

2.2 Agent Model

Our agent model consists of three types of agents (cf. Fig. 2.2 on the next page). 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 stationary, 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. Contrary, 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, cf. Sec. 2.7 on page 17.

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 once. The creation can be initiated either from outside the Tracy agent server, e.g. using the TracyAPI (cf. Ch. 6 on page 49) or the graphical user interface (cf. Sec. 5.2 on page 44), 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 Sec. 2.5 on page 11, where we present our mobility model in detail. Agents can communicate with each other either by using asynchronous messages, or by leaving information on a blackboard, see Sec. 2.4.

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.
2.3 Accessing the Host System via Service and Gateway Agents

As already noted in the last section, we divide three types of agents. Why this classification? First of all there are security reasons. Because of the ability to migrate, mobile agents are considered as 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 as secure and they may access the host. But sometimes mobile agents need to access the host system, too. Now, our decision to classify agents helps us to solve this problem. We use the stationary agents as a dynamic interface and as a security wall to allow mobile agents the access to the host system. This is the concept of our system agents. And what is the concept of our gateway agents? A similar problem as the described problem while accessing the host system exists for associated applications, i.e., for legacy software or database management systems. Mobile agents should be able to use local applications to solve their tasks. Such applications can be accessed by gateway agents. So, this kind of stationary agents is a dynamic interface and a security wall too. But this is not the only aspect. Gateway agents connect local applications to the agent server. The application “speaks” with the agent server via a gateway agent and other agents can “speak” with the application via a gateway agent, as well.

We have seen that there are not only security reasons for defining different agent types. There are problem specific reasons, too. In Fig. 2.3 on the following page the various types of agents are illustrated with the connections to the operating system and associated applications. These types of 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., any logical infrastructure can be created with the help of system agents.

2.4 Communication between Agents

Mobile agents are a design paradigm for distributed computing, in which a mobile agent migrates to another platform to fulfill an 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 database, 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 mail-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 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 (see Sec. 5.2 on page 44).

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 Fig. 2.3). 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 information on the blackboard about services provided by the server. The blackboard is a mean to provide persistent data, too (we will implement blackboard persistence in a later version of Tracy).

The blackboard itself is organized like a file system – there are directories and files. A directory
Section 2.5. Agent Migration

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.

By 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.

2.5 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. Current mobile agent systems offer various ways to migrate mobile agents. In this section we classify the migration techniques according to the following aspects:

1. How does the MAS interpret the term state?
2. What is the migration strategy, i.e. which pieces of code and data are transferred and when are they transferred?
3. What is the transmission strategy, i.e. what is the protocol used to transmit an agent to the destination platform?

2.5.1 Strong and Weak Mobility

Existing mobile agent systems can be distinguished by the type of mobility they offer to the programmer. Each type can be characterized by the interpretation of the term state.

The weakest form of mobility only transmits the instance variables 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 in MOLE [33] and Discovery [21]. In a stronger form of mobility the MAS 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 [24]. The drawback of these two forms of mobility is that the programmer has additional effort to implement state marshaling and unmarshaling of local variables. The migration command can only be the last instruction within a method, so that changing the platform induces invoking a new method.

Mobile agent systems that offer the highest level of agent mobility cannot only marshal all instance variables, but also all local variables of the current method (procedure), together with the
program counter and the call stack. On the destination platform the agent is initialized and started at the first instruction after the \texttt{go}-command. First mobile agent systems, like Telescript [39] or AgentTCL [13] 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 migration to a MAS, if full access to the underlying programming language, the compiler, and the runtime-system is available. A new command \texttt{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 MAS. To implement strong migration 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 it. Modifying the JVM is difficult to carry out (although it is said to be done in Ara [27] and Sumatra [1]), and strategically imprudent. A customer can only use the resulting MAS if he uses the modified JVM, not to mention problems of licensing the JVM source code. The other way to offer strong migration to the programmer is agent source code transformation. Fünfrocken [10] transforms the agent’s source code by a pre-processor that inserts code to save and recover the execution state. Another attempt is made by Sekiguchi et al. [32]. The drawback of both methods is a longer source code and a not neglectable performance decrease. Strong migration 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 mobile agent server 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.

### 2.5.2 Migration Strategy

In this section we will have a look at the way code and data can be transmitted to the destination platform. All migration techniques we will describe now determine all data that must be transmitted to the destination site automatically. In most mobile agent systems based on the Java programming language, \textit{object serialization} [34] is used for this task.

Concerning code transmission, we can distinguish four strategies, compare Fig. 2.4 with regard to transmission type, site location and code granularity. Some systems offer a migration strategy that we call \textit{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, see Fig. 2.5 (a). Some systems do not transmit the whole code but filter 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 (MAS). This migration strategy is used for example in Voyager 2.0 [25], Ara [26], and Extended Facile [18]. We could perceive by intuition that this strategy is fast, because only one transmission is necessary for the complete agent. This strategy also corresponds to one of the main characteristics of mobile agents, that is autonomy. The agent needs no connection to the home platform, from which it was started. The drawback of this method 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 file 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 (Fig. 2.5 (b)), whereas the second strategy loads all class files as one package immediately if one class file must be loaded (Fig. 2.5 (c)). The pull strategy is used for example in MOLE [33]. 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 one 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, see Fig. 2.5 (d)). 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.

Most mobile agent systems offer only one of these strategies. In contrast, for example in Sumatra [1], the programmer can combine several of these strategies manually. We do not know any system in which the decision on what migration strategy should be used is made by the agent system automatically.

2.5.3 Transmission Strategy

The transmission strategy defines the way an agent is actually transmitted to the destination platform. Most Java based mobile agent systems use remote method invocation (RMI) [35] for this task. The destination agent server defines a RMI server that defines a method to accept a mobile agent. Some systems use other protocols, like FTP, HTTP, or SMTP. Other systems define their own transmission protocol on top of the TCP/IP protocol.

2.6 Agent Mobility Models

2.6.1 Introduction

In this section we define the term mobility model and propose the Tracy agent mobility model by which our mobile agent system distinguishes itself from all other agent systems. After we have become acquainted with different migration strategies in the last section, we will now have a look at how these strategies are implemented.

An agent mobility model defines abstractions and the behavior of an agent server when mobile agents migrate to another platform. The mobility model decides for example on the code and data granularity, i.e. of what size are the pieces of code and data that can be transmitted between agent servers. As we have already seen in the last section, current mobile agent systems can only handle class or archive granularity for code, and the serialized agent for data. Another issue that we have observed is whether the code is pushed to the destination platform in an active way, or is pulled from the origin platform. Other aspects of the mobility model concern commands to initiate migration, whether the programmer can use strong or weak migration, etc. A specific migration strategy uses the building blocks of such a mobility model. It is clear that the more flexible a mobility model is, the more migration strategies can be implemented. For example, a model which is not designated to allow pushing code to the destination platform can obviously not be used to implement the push-all-to-next strategy.

2.6.2 Example: Aglets Mobility Model

Let’s take the mobile agent system Aglets [20] by IBM as an example here. In Aglets the programmer can influence the way of migrating agents’ code by assembling several class files to a Java archive (JAR) file. By this technique a mixture of both strategies push-all-to-next and pull-per-unit can be realized. Normally, an agent consists of several class files as usual in a Java program. If the agent migrates, only those classes are transmitted along with the serialized agent for which there exists a living object, i.e. an object that is referenced by a non-transient instance variable. Other class files must be fetched by the destination platform if they are needed. According to our
Section 2.6. Agent Mobility Models

migration strategy naming pattern introduced in the last section, we could denote this strategy as *push-living-to-next-pull-others*. Using a Java archive which includes the agent’s class file, leads to the transmission of the complete archive, i.e. all classes in that archive, when the agent wants to migrate. This may lead to transfer of classes that are currently not in use. The advantage of this method is, however, that it does not require fetching of additional classes, if it is guaranteed that the archive contains all necessary class files. This strategy is comparable to our push-all-to-next strategy.

This mobility model is characterized by the opportunity to choose between two different migration strategies. However, the decision must be made by the programmer and cannot be influenced or modified later during runtime, neither by the programmer, nor by the agent itself. It is not possible to adapt the migration process so that other migration strategies can be implemented. Lange and Oshima [20, p. 168] already pointed out the major argument pro transmission as a JAR file. If the agent migrates outside a fire wall, fetching required classes from the code base (within the fire wall) is rather impossible. In this case, the programmer is well-advised to gather all classes to a JAR file and allow complete code transmission. However, if the agent needs not to cross any other fire wall during life-time, always transmitting the whole code was a bad choice. The other drawback we see is the selection of class files in a strategy push-living-to-next-pull-others. The aim of reasoning about which classes should be transmitted to the destination platform is to find out those classes that will be used on the remote site with high probability. The fact that there exists a living variable referencing an object of a specific class does not imply high class usage probability, in general.

2.6.3 The Tracy Mobility Model

The Tracy mobility model that we will present now, gives the programmer more influence on the migration strategy, and even the transmission strategy the agent should take. All migration strategies mentioned above can be implemented using this model. 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. We plan to offer strong migration by source code transformation, comparable to Fünfkoken [10], later. A mobile agent can use one of the following commands to start the migration process:

1. \texttt{go(destination, method)}, as described above
2. \texttt{go\_back(method)}, initiates a migration back to the last platform the agent came from, and
3. \texttt{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, "PASSIVE" ) 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 origin platform and a pre-defined method with name migrationFailed is invoked.

Our migration model is built from the following abstractions: First of all, a data item can be a variable or any kind of data that the agent needs for execution. A data item is uniquely identified by a name. The state of a mobile agent consists of a set of data items (variables), and the name of the next method to invoke on the destination platform. Normally, a state does not include all of the agent’s variables, i.e. the agent’s state is less than the serialized agent. A mobile unit is a combination of some code and data items. Code is not defined in particular within our model, we leave that to the migration strategy. In our intention, the entire code of an agent is subdivided into many non-overlapping code components, each forming a mobile unit. The reason for transmitting data items in a state as well as in a mobile unit is that mobile units can be downloaded to a server on demand. Data that was not sent with the state (because it was not required for execution, so far) must be transmitted along with the downloaded code that will use it for the first time. Each mobile unit has a home platform, and a set of additional platforms from which it can be downloaded. At last, the agent unit definition block (AUDB) holds the description of all units that belong to one mobile agent. Especially, the AUDB contains the information from which agent servers available in the network a unit can be downloaded.

Now we will roughly describe the agent transfer process from the viewpoint of the origin platform. The whole agent transfer process is determined by the specific migration strategy. To program a migration strategy we introduced an script-like language MDL (Migration Definition Language), cf. Sec. 8.3.5 on page 81. Using this language it is possible to decide on the structure of mobile units and which units and data should be sent to which platform.

In our mobility model three types of agent servers are distinguished. First, the home platform is the agent server on which an agent is started, and to which the agent can return without carrying any code because the home platform does always keep all agent’s code and data. Second, there is a remote platform which is any agent server on which an agent temporarily resides at. At last, there is a mirror platform which is a remote platform on which some (or all) mobile units of an agent can be replicated. The advantage of a mirror platform is that mobile units might be downloaded faster from a mirror platform than from the home platform. Corresponding to our model of a home platform we distinguish the following three cases of migration:

1. migration is initiated on the home platform
2. migration is directed to the home platform
3. migration is initiated and directed to platforms not equal to the agent’s home platform

If the migration process is initiated on the agent’s home platform the agent (code and data) first must be divided into one state object and several mobile units. This subdivision into mobile units cannot be changed later during an agent’s life cycle, whereas the structure of a state object is newly defined for each migration. At this time, the state and some mobile units are annotated with the information to which destination platform they should be transferred. After the agent has left its home platform, mobile units and data items are not deleted, because code and data could be downloaded by another agent server and data items could be updated. If the agent must be transferred to its home platform, it can be assumed that all mobile units are already there, so only
the state must be transmitted. In the third case, the agent server has to decide which mobile units should be transmitted to the next platform. Data that is neither carried along with the state object, nor carried along with any mobile unit, must be updated on the home platform. After the agent has left the platform data items and all mobile units are deleted. In the third case of migration it might be useful to declare that the current platform shall be a mirror platform, now. That means that some units can be also downloaded from this platform in future.

After the definition of state and mobile units, the network transmission is started. State and mobile units, and the agent unit definition block (AUDB), will be sent to one or many agent servers. Note, that the AUDB must be available on an agent server before any mobile unit can be received. Migrating an agent could have the consequence of multiple network transmissions, e.g. one to update data on the home platform, and one to transmit some mobile units and state to the next agent server.

On the destination platform mobile units and state are received. The agent’s code is generated from the code component of the mobile units. The agent’s variables are initialized by the state object. During the agent’s execution it can happen that the control flow reaches a part of code that is not on this platform, yet. In this case, dynamic unit downloading is initiated by the agent server. A set of agent servers that possibly hold the required unit is deposited with the AUDB.

Currently, we have already implemented this model, except of the ability to transmit data items other than the whole serialized agent. Other techniques will be integrated in a next version of this migration model, e.g. a unit cache. A unit cache is placed on each agent server, so that mobile units could be shared between different agents of the same type. A unit cache can also save unit transmission, because an unit needs not to be transmitted to the destination platform if it is already located in the remote unit cache. However, on top of our migration model, we have already successfully implemented all migration strategies introduced in Sec. 2.5.2.

In our migration model the programmer resp. the agent can also influence the technique used to transmit code and data over the network. We have successfully implemented two different transmission strategies, that we call SATP/TCP and SATP/RMI. Several other techniques are currently under development. The SATP/TCP transmission strategy uses a Tracy specific transmission protocol named Simple Agent Transfer Protocol (SATP) on top of the TCP/IP network protocol. The SATP protocol is tailored to our mobility model and is able to transmit mobile units, data items, and the agent unit definition block. We describe this protocol in Sec. 8.4.3 on page 89. Strategy SATP/RMI also uses our SATP protocol, but transmits code and data by remote method invocations on the remote agent server. Other strategies that are currently under development will offer secure transmissions using Secure Socket Layer (SSL) and faster (but less reliable) network transmission on basis of UDP.

2.7 Security

Our main focus in Tracy lies on mobility aspects, so we did not spend much effort on security issues yet. In order to protect the agent server and the underlying host from malicious agents, we have activated the sandbox security model of Java 2 [36]. With that we can guarantee (as long as Java 2 can guarantee it), that no mobile agent has the permission to read or write to the local file system, open insecure network connections, etc. The Tracy security models is built upon the concept of a home platform. Agent servers can grant specific permissions to agents due to the home platform they were created on. It is not possible to grant permissions to a specific agent only.
Chapter 3

Programming Tracy Agents

In this chapter we give a comprehensive introduction to agent programming in Tracy. In the first section we introduce the class Agent which is the main class within the TracyAPI. We describe an agent’s life-cycle and give a first example how to add a new agent to a running agent server. The following sections are dedicated to communication (Sec. 3.2) and migration topics (Sec. 3.3). Section 3.4 is to inform the reader about the connection between agents and agent servers. We explain by which methods the agent server controls agents. The next section is dedicated to some conventions that should be noted when programming agents in Tracy and in the last section we present examples for a mobile agent and a system agent.

3.1 Introduction

The class Agent in the package de.unijena.tracy.agent is the main class within the TracyAPI. It is an abstract class that serves as base for the three classes of agents we have already mentioned in Sec. 2.3, which are MobileAgent, SystemAgent, and GatewayAgent. Class Agent must not be instantiated by the programmer. The Agent class defines methods and instance variables to control an agent’s life-cycle, get and set internal data structures, and receive messages. Some of these methods are useful to the programmer, e.g. methods to inform about the current agent server, whereas some methods are only useful for an agent server to control the agent. The Agent class also defines some methods that are supposed to be overridden by subclasses.

First, we will explain the life-cycle of an agent, see Fig. 3.1 on the following page. At the beginning the state of an agent is Undefined. Before the state changes to Running necessary things will be defined within the agent, e.g. the interface to the agent manager will be set. The agent execution always begins by the method call invokeMethod() which invokes an agent’s method, e.g. the method startAgent() will be invoked if the agent is started the first time. If the thread of execution of this first method comes back to the invokeMethod() the state changes to Waiting until the agent receives a message. The agent can change by itself the state Running to Waiting if it calls the method waiting(). As in the first case, the state changes to Running if the agent receives a message. During execution the agent can get a suspend signal which interrupts agent execution. After a resume signal the execution will be continued. The state of the agent changes to Suspended during this time. The agent can end his execution by calling the die()-method. This call changes the state to Dying and the agent will be removed. This scheme holds for stationary agents and mobile agents. In the case of a mobile agent there is an additional state Dispatching while the agent is migrating to another platform.
Let's now see how an agent can be defined. The first point you must have in mind is that you do never instantiate any agent by yourself. You only define a new agent class by subclassing one of the three agent classes and override some methods. Agent instantiation in any case is done by the agent server, i.e. an agent cannot live without an agent server and you cannot add an existing agent object to an agent server afterwards. Some initialization must be done by the agent server and without that the agent is not properly defined. To give a practical example, we subclass MobileAgent, although we have not defined 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, package de.unijena.tracy.agent must be imported, which includes all basic definitions.

```java
import de.unijena.tracy.agent.*;

public class MyFirstTracyAgent extends MobileAgent {
    public MyFirstTracyAgent() {
        // do some initialization
    }

    public void defineAgent() {
    }

    public void startAgent() {
        // do something
    }
}
```

After compiling this class you can start this agent either by the graphical user interface (see Sec. 5.2 on page 44), or the TracyAPI (see Ch. 6 on page 49). As soon as the agent server instantiates this agent, the class constructor is called. After the object is instantiated, the agent is started by the agent server by invoking method startAgent(). This method is defined abstract in class Agent, so you must override it in your definition. The same holds for method defineAgent() that is called by the agent server when the agent should be configured, e.g. by a graphical user interface.
As already said, after an agent’s instantiation the method `startAgent()` is called by the agent server. The agent server assigns an own thread of execution. The agent is running in parallel to other agents probably existing in the agent server. If this method terminates, the agent terminates and is deleted from the agent server as in the example above. To prevent an agent from dying you can take one of the following two measures: First, the agent can migrate to another platform on which a new method is called. Second, you can set the agent to the state `Waiting` by invoking method `waiting()`. So, for example:

```java
public void startAgent() {
    printMessage("Hello World");
    waiting();
}
```

Here, the agent prints a message on the agent servers console and falls in state `Waiting` afterwards. In this situation the agent still exists in the agent server, but it is passive now, because it does not have an execution thread any more. The only way to reactivate an agent is by sending a message to it (see Sec. 3.2 on the next page).

Of course, there is also a method by which an agent can terminate itself explicitly, that is

```java
protected final void die()
Called by the agent to kill itself.
```

This method is called by an agent itself to indicate that its life-time is over. The underlying agent server is allowed to delete the agent. Now, let’s run through all messages that an agent can use to inform itself about its environment.

```java
public final String getHomeAddress()
Returns the agent’s home platform address.

public final String getAgentName()
Returns the agent’s name.
```

Both methods are defined `public` because they are also used by the agent server.

```java
protected final String getHostName()
Returns the host name of the underlying computer system, e.g. dagobert.uni-jena.de
```

There are four methods to print messages on the agent server’s console. It depends on the agent server whether those messages are printed on the terminal output directly, or sent to a graphical user interface.

```java
protected final void printMessage( String message )
Prints a normal message.

protected void printQuestion( String message )
Prints a user-directed question (marked with a special icon in the user interface).

protected void printWarning( String message )
Prints a user-directed warning (marked with a special icon).

protected void printError( String message )
Prints an error message (marked with a special icon).
```
3.2 Communication

There are two possible kinds of communication (cf. Sec. 2.4 on page 9): communication via asynchronous messages, and communication with the help of the blackboard. Because of that we will divide this section into two chapters – communication through messages and communication using the blackboard.

3.2.1 Communication through Messages

First of all we will look at the structure of a message. Let us deduce the parts of a message from a mail. We need a recipient (addressee), a sender and, of course, a content. A subject would be fine, too, so a message could be of a certain type. What does the recipient need to know? The recipient needs to know the type, content, and the name of the sender. The name of the recipient is only needed to deliver the message, we see thus a message consists of a sender, has a type, and has a content. The “post office” will deliver the message to the recipient. All these parts of a message and the name of the recipient are simple strings.

Now let us look at the “post office”. Every agent will be registered at the post office and gets its own mailbox. The registration is done before the agent is started, and the mailbox will be removed if the agent ends its execution or, in the case of a mobile agent, if the agent migrates to a remote server. We will translate the terms: the mailbox is a message queue where new messages will be spooled to, and the post office is a message queue disposer which manages the queues, spools messages, informs the agent about new messages, and delivers messages to the agents. By the way, only agents can receive messages, but agents, the agent server and the GUI can send messages. If there is a message for an associated application a gateway agent must be the recipient, and if an associated application wants to send a message, the gateway agent must do this job. System agents can also be used to accept messages for the agent server.

The type or subject of a message is used to tell an agent the kind of message, e.g. as a result of a message with the subject “go” an agent might migrate to a remote server, maybe the name of the remote server is specified in the content of the message or in the program code.

Let us look at the sequence of sending and receiving messages in detail. First, the sender must create a new object of class Message and fill in all required fields. After that the message can be sent with the method

```java
protected final void sendMessage(String recipient, Message newMessage)
Send a message to an agent, of whom its name is known.
```

which is defined in class Agent. There is another method

```java
protected final void sendMessage(String recipient, String type, String content)
Send a message to an agent. The message-object is created from type and content parameters.
```

which creates a new message object automatically. The message and the name of the recipient will be transferred to the message queue disposer which tries to spool the message to the queue of the recipient. If a message was queued successfully the recipient, i.e. the agent, will be informed that a new message has arrived. For that purpose there is a flag newMail defined in class Agent. Now the agent may initiate message delivery using the following method

```
protected final void sendMessage(String recipient, Message newMessage)
Send a message to an agent, of whom its name is known.
```
public final void checkForMail()
Checks for new messages and reads all spooled messages.

which gets all spooled messages, one after the other. This method is defined public because
it can be called by the agent server when the agent itself is in state Waiting. Processing one
message is done by calling the following method

protected abstract void handleLastMail(Message msg)
Analyzes one message and defines actions for message handling.

This method has to be implemented by the programmer and is used to analyze the message as
well as to define the actions of the agent. Please note that the programmer is responsible to check
whether new mail has arrived. Thus, you should call method checkForMail() frequently. If
the agent is in state Waiting it need not to check for new mails periodically, because the agent
server initiates message delivery itself. Here is a small example to illustrate mail processing

1 public void handleLastMail(Message inMsg)
2 throws AgentExecutionException {
3
4 if (inMsg != null) {
5 if (inMsg.getTyp().compareTo("run") == 0) {
6 if (inMsg.getContent().compareTo("" ) != 0)
7 pattern = inMsg.getContent();
8 printMessage("I will search for: " + pattern);
9 go("remote.host.org", "searchRemote");
10 }
11 else
12 printError("Unknown message type: " + inMsg.getTyp());
13 }
14 }

Each agent can manage its own message queue by the following methods

protected final void getQueueState()
Checks the state of the message queue (opened or closed for receiving messages).

protected final void openQueue()
Opens the message queue. After that the agent can receive messages (messages can be spooled
in the queue of the agent).

protected final void closeQueue()
Closes the message queue. After that the agent cannot receive messages (messages cannot be
spooled in the queue of the agent).
3.2.2 Communication using the Blackboard

The blackboard can be used for indirect communication or for storing data as introduced in Sec. 2.4 on page 9. The blackboard is organized like a file system with directories and files, and the entities have read and write permissions. Files have a structured content (or value). This content of a file entity might be plain text, a HTML document, binary data or a simple boolean flag, etc.

An agent can put such entities on the blackboard or can get one from the blackboard. To put a new entity on the blackboard the agent uses the method defined in class Agent

```java
protected void putEntity(String anEntity, BlackboardContent aValue)
writes an information to the blackboard.
```

Class `BlackboardContent` is in package `de.unijena.agentsystem.comm`. The agent directs the request to the agent server. The agent server will forward the method call to the blackboard which checks write permissions. The name of the entity has to contain the complete path, e.g. `/complete/path/to/entity`. If the path does not exist it will be created automatically. By the way, this is the only way for agents to create directories. If the named entity exists on the blackboard the content will be changed to the new one if the permissions (read/write) are set respectively. If the entity does not exist, the entity will be created. The content of the entity has to be a type or a subtype of `BlackboardContent` which is a container for the entity content and has several access methods. There are subtypes of this class, e.g. `BBXMLContent` for XML documents, or simple boolean flags in `BBBooleanContent` for simple boolean flags. Here is a small example for an agent that puts a String (class `BBStringContent` on the blackboard:

```java
1 public void startAgent() throws AgentExecutionException {
2     // put agent's name on the blackboard
3     try {
4         putEntity("/Sample/Agents/AgentName",
5               new BBStringContent("ContentName", name));
6     } catch( BlackboardException be ) {
7         printError("Failure: " + be.getMessage());
8         // end execution
9         die();
10    } // do something other
11 }
```

In our example you see that a blackboard exception could be thrown, which indicates that the agent has had no permissions to write the entity or there exists a directory with the same name already.

There can also be system entities on the blackboard. These files or directories are owned by the system. Mobile or gateway agents cannot put or change such entities on the blackboard. System agents can create, change and delete the system entities. Only system agents have the following methods

```java
protected void putSystemEntity(String entityName, BlackboardContent aValue)
writes a system entity on the blackboard. If the entity exists already the content will be changed.
```

```java
protected void deleteSystemEntity(String anEntity)
deletes a system information from the blackboard.
```
To access the content of an blackboard’s entity there is the following method which returns the content of the entity.

```java
protected BlackboardContent getEntity(String anEntity)
Reads information from the blackboard.
```

Please note that this method is also used by system agents to read system entities. Let us have a look at an example again. It reads the content of the file which we have written on the blackboard in our last example above:

```java
public void startAgent() throws AgentExecutionException {
    // container for the agents name read from the blackboard
    String readName;
    // read agent’s name from the blackboard
    try {
        BlackboardContent bc =
            getEntity("/Sample/Agents/AgentName");
        if ( (bc.getContentType()).compareTo("String") ) {
            readName = (String)((BBStringContent)bc).getContent();
        } else {
            // check other possible types
        }
    } catch( BlackboardException be ) {
        printError("Failure: " + be.getMessage());
        // end execution
        die();
    }
    // do something other
}
```

The example is quite similar to the write-example above. Ones we have accessed the content `bc` of the entity we have to check the type of the content. After that we can access the contained object (which is a string in this case) by using the `getContent()` method. If the named entity does not exist or the agent has no permission a blackboard exception will be thrown.

To delete entities there is the following method

```java
public void deleteEntity(String anEntity)
Deletes an information from blackboard.
```

which removes the named entity from the blackboard if it is allowed to do.

There are some methods which can be used to browse the blackboard. The first method can be used to get a recursive list of entities starting at the named entity.

```java
public Vector getChildsOfFolder(String anEntity)
Returns a list of all entities of the blackboard starting with the specified entity.
```

The second method scans the blackboard for entities that match to a given pattern. There is a flag to scan case sensitive.

```java
public Vector searchEntity(String someKeywords, boolean caseSensi-
tive)
Searchs for entities on the blackboard which contain a specified keyword.
```

It is possible to observe an entity, that means every time when this entity changes, including a
entity deletion, the observer, in this case an agent, will be informed. For that purpose an agent
has to register with the entity that should be observed. See method

    protected void getEntityWhenChanged(String anEntity)
    Registers the agent as an observer within the entity on the blackboard.

Notification about changes are sent via messages to the agent. These messages have special sub-
jects which provide the kind of change that occurred. The message body contains further in-
formation. See the following short description of message types (subjects) defined in the class
Entity which is resided in the package de.unijena.tracy.agentsystem.comm that have
to be handled by the handleLastMail() method:

    Entity.LEAF_CHANGED
    File content of observed file has changed. Name of file with complete path contains the message
    body.

    Entity.LEAF_DELETED
    Observed file was deleted. Name of file with complete path contains the message body.

    Entity.LEAF_ADDED
    New file was added within observed directory. Name of file with complete path contains the message
    body.

    Entity.FOLDER_DELETED
    Observed directory was deleted. Name of directory with complete path contains the message body.

    Entity.FOLDER_ADDED
    New directory was added within observed directory. Name of directory with complete path contains
    the message body.

    Entity.ENTITY_RENAMED
    Name of an observed file or directory was changed. Name of entity with complete path contains the
    message body.

We modify the small message handle example from above to illustrate the handling of these spe-
cial messages:

```java
public void handleLastMail(Message inMsg)
    throws AgentExecutionException {
    if (inMsg != null) {
        if (inMsg.getTyp().compareTo(Entity.LEAF_CHANGED) == 0) {
            // define a reaction here
        } else if (inMsg.getTyp().compareTo(Entity.LEAF_DELETED) == 0) {
            // define a reaction here
        } else if (inMsg.getTyp().compareTo(Entity.LEAF_ADDED) == 0) {
            // define a reaction here
        } else if (inMsg.getTyp().compareTo(Entity.FOLDER_DELETED) == 0) {
            // define a reaction here
        } else if (inMsg.getTyp().compareTo(Entity.FOLDER_ADDED) == 0) {
            // define a reaction here
        } else if (inMsg.getTyp().compareTo(Entity.ENTITY_RENAMED) == 0) {
            // define a reaction here
        } else {
```
The observer can be removed with the following method

```java
protected void removeObserver(String anEntity)
    Removes the agent as an observer from the blackboard's entity.
```

### 3.3 Migration

In this section we will show how a mobile agent can start migration to a remote agent server. Actually, there are two ways to move a mobile agent. One is to use the `go` command to initiate migration with a default migration strategy and a default transmission strategy to a given destination platform. The other way is to use so-called migration properties to configure the migration process in detail. In the latter case the programmer can choose the migration strategy and the transmission strategy the agent should use. Additionally, using the concept of migration properties the programmer can define a migration route once. The next `go` command automatically chooses the next destination platform in the given itinerary. The `go` method is the tool to invoke agent migration. This command is defined in the abstract class `MobileAgent` and cannot be overridden. Only the agent itself can initiate the migration process.

```java
protected void go( String dest, String methodName )
    Migrates an agent to the destination platform dest and restart it by invoking method methodName.
```

The name of the destination platform is just the name of the remote server, like e.g. dagobert.uni-jena.de. It depends on the Tracy agent server configuration which migration strategy and which transmission strategy is chosen in this case. If no configuration file is given the push-all-to-next migration strategy is chosen, and the agent is transmitted using a TCP/IP network connection. Calling the `go` method stops agent execution immediately. Statements following the `go` invocation will never be executed, neither in the case of a migration error. A `go` command might be included in a `try ... catch` clause. In this case, neither the exception `AgentExecutionException`, nor super classes of this exception, must not be caught.

```java
try {
    ...
    System.out.println("Running on server pluto.uni-jena.de ...");
    go( "dagobert.uni-jena.de", "runOnDagobert" );
    // statements below will never be executed
    System.out.println("This message will never be seen.");
} catch( IOException e ) {
    System.err.println( e.getMessage() );
    e.printStackTrace();
}
```

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

```java
protected void removeObserver(String anEntity)
    Removes the agent as an observer from the blackboard's entity.
```
protected void go_home(String methodName)
Migrates an agent to its home platform and restarts it by invoking the method with the name
methodName.

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

When a mobile agent should migrate to another agent server, serialization of the agent takes place.
Serialization means that all instance variables of the agent, together with all recursively referenced
objects and their instance variables are traversed and put into a flat byte array. In the current
version of Tracy we use the standard object serialization technique that is already implemented in
Java [34]. Except of all transient references all variables and objects are serialized. In the
next version of Tracy it will be possible to only serialize a subset of variables and references
according to the migration strategy used. Due to the use of the standard serialization technique
it is necessary that all referenced objects must be either marked as transient, so that they are
not marshaled at all, or they must implement the interface java.io.Serializable. If a not
serializable object is found during serialization an exception will be thrown. Note, that class
variable are not part of the serialized object. Thus, the object will probably retrieve different
values of class variables at the destination platform.

If the migration process is successful, the mobile agent is removed from all local agent directories.
However, migration can fail for some reasons. The most frequent reason will be that the desti-
nation platform is temporarily not available. In this case the agent is restarted by the local agent
server by invoking the method migrationFailed. Within this method the agent can decide
whether it wants to migrate to another platform or stop working at this point.

The second way to migrate an agent is to use so-called migration properties which allow the pro-
gramer to configure the migration process in detail. Migration properties are encapsulated in
an object of the class MigrationProperties in the package de.unijena.tracy.agent. A
migration properties object is created using one of the following constructors:

public MigrationProperties(String dest, String method)
Creates an object and defines the next destination and the name of the next method to invoke.

public MigrationProperties(String dest, String method, String
migStrategy)
Creates an object and defines the next destination, the next method, and the name of the migration
strategy to use.

The migration strategy is named by a string, e.g. ”ACTIVE” for the push-all-to-next strategy,
”PASSIVE” for the pull-per-unit strategy, and ”PREFETCH” for the push-all-to-all strategy. The
latter strategy does only make any sense when a route (described below) is defined. You can
define the destination platform and the method to invoke after creating the migration properties
object, too.

public MigrationProperties(MigrationProperties migStrat)
Creates an object and a copy of all properties from the given migStrat object.

public MigrationProperties(String migStrategy)
Creates an object and define the name of the migration strategy to use.

In these two cases, the agent must define the destination platform later by the following methods
of class MigrationProperties.
public void setNextDestination( String destination )
Defines the name of the next destination platform.

The next method that should be invoked on the destination platform can be defined by:

public void setMethod( String methodName )
Defines the method to be invoked on the destination platform.

In the case that all migration related information are stored in a MigrationProperties object, the following new go command should be used.

protected void go( MigrationProperties mp )
Migrates an agent according to the given migration properties.

Look at the following example:

```java
void runOnPluto() {
  MigrationProperties mp = new MigrationProperties( "PASSIVE" );
  mp.setNextDestination( "dagobert.uni-jena.de" );
  mp.setMethod( "runOnDagobert" );
  go( mp );
}
```

It is also appropriate to use an object of the class MigrationProperties if the transmission strategy should be defined by the user.

```java
public void setTransmissionProperty( String name, String value )
Sets the transmission property with name name to value value.
```

For example, to define that for the next migration strategy RMI should be used you call mp.setTransmissionProperty( "strategy", "RMI" ); assuming that mp contains a reference to a valid object of class MigrationProperties. To define the transmission strategy is currently the only transmission property that can be set. In a next version of Tracy further transmission properties, e.g. data compression, will be available.

Sometimes, a mobile agent has to visit a collection of agent servers that is already fixed when the agent is started. Additionally, the agent has to perform almost the same tasks on each server. In this case, migration properties offer the opportunity to define a so-called route, which is a fixed set of agent server names to which the agent will migrate sequentially. On each server the mobile agent is started by invoking the same method.

```java
public void setRoute( String[] route )
Defines a migration route. The array of String objects contains the agent servers' names.
```

The order in which the agent servers are visited is defined by the order of agent server names within the array, i.e. first, it is visited server route[0], then route[1] and so on. In case that a route is defined the go( mp ) command always uses the next platform name as destination agent server.

```java
public class Runner extends MobileAgent {
  public void startAgent() throws AgentExecutionException {
```
In this example a new `MigrationProperties` object is defined, a route and a method are set. The agent migrates to all destinations defined in the route, on each agent server the `startOnPlatform` method is invoked. If no more agent server is left the agent migrates back to its home platform and dies. To ask a `MigrationProperties` object whether there are entries left, the following method can be used.

```java
public boolean routeHasMoreElements()
Returns true if and only if there are more unvisited agent servers in the given route.
```

Using an object of type `MigrationProperties` is also appropriate when only some migration properties should be set once, whereas others should vary from one migration to an other. For example, the migration strategy should be fixed, but destination name and next method should be defined in the `go` command. Within a mobile agent you can define so-called default migration properties by using the following method.

```java
protected void setDefaultMigrationProperties( MigrationProperties migProp )
Defines the given migration properties as default value.
```

If such a default migration properties object is defined, the mobile agent takes it into consideration at the next migration. However, migration properties set by the `go()` command always overwrite corresponding entries of the default migration properties object. See the following example:
Section 3.4. Methods used by the Agent Server to control an Agent

Here, the agent migrates to host pluto.uni-jena.de and is restarted with method runOnPluto, although the default migration properties define that method startOnPlatform shall be invoked. Notwithstanding, migration is done using the RMI transmission strategy.

When programming a mobile agent some general programming conventions must be followed. Each method must throw the exception AgentExecutionException in package de.uni.jena.tracy.agent, when a migration initiation might occur during its execution. That means not only this method itself, but also all other methods that might be invoked. The reason for this is that the migration request is propagated to the agent server by an exception of type WantToMigrate which is a subtype of AgentExecutionException.

3.4 Methods used by the Agent Server to control an Agent

One of the main characteristics of agents is their autonomy. However, sometimes an agent needs help of the agent server, e.g. to start other agents or to communicate with other agents. In some cases the agent server needs to control all agents currently residing on it, e.g. if an agent wastes resources or the agent server has to shut down. Therefore, some control-related methods are implemented in the class Agent.

At the beginning of an agent’s life cycle the agent has to be initialized. The following method is called if the agent is started for first time.

```
class Agent
{
  public final void init(String name, String home)
  {
    This method is called by an agent server to initialize a new agent with name (name) on its home platform (home).
  }
}
```

We meant to use Java methods to control an agent’s execution thread, but since Java 2 these methods are deprecated mostly. So we had to use a control flag within the agent. This flag has to be tested by the agent itself periodically. Forehanded we have integrated some check points but within the agent’s source code the programmer has to insert additional check calls if the agent should be able to react as fast as it can. For example, if the agent server fails, the agent will get a hint from the server but the agent has to check the control flag to recognize the failure and to decide how to handle it.

There are several methods to control an agent. These methods will usually be called by the agent manager, e.g. to suspend agents that waste resources or when the user wants to control agents via the our graphical user interface (see Sec. 5.2 on page 44). The following methods are implemented within the class Agent.
public final void suspendAgent()
Suspends an agent.

public final void suspendAgentWithCheck()
Suspend agent and check the control state. If no other agent thread is running (state of the agent is Waiting) this method should be invoked instead of suspendAgent().

public final void resumeAgent()
Resumes agent.

public final void stopAgent()
Stops agent.

public final void quitAgent()
Asks agent to quit itself. Behavior depends on the content of quitCmd.

public final void quitAgentWithCheck()
Asks agent to quit itself and check the control state. If no other agent thread is running (state of agent is WAITING) this method should be invoked instead of quitAgent(). Behavior depends on quitCmd.

The following methods are only defined within class MobileAgent because only this type of agent has the ability to migrate.

public final synchronized void migrateAgent()
Asks the agent to migrate. Behavior depends on migrateCmd, migrateMeth, and migrateDest.

public final void migrateAgentWithCheck()
Asks the agent to migrate and check the control flag. If no other agent thread is running (state of agent is Waiting) this method should be invoked instead of migrateAgent(). Behavior depends on migrateCmd, migrateMeth, and migrateDest.

As suggested above there are some variables which influence an agent’s behavior. The agent can be asked to quit or to migrate. In both cases the agent can simply ignore such a request or react on it. Within the agent’s source code there are predefined constants which can be used by the agent programmer to define the behavior in such cases:

IGNORE
behavior: ignore a quit or migrate request

QUIT
behavior: react on quit request

MIGRATE
behavior: react on migrate request (only mobile agents)

The behavior can be set by using the following methods:

protected final void setQuitCmd(int i)
This method sets the behavior of an agent for a quit request.

protected final int getQuitCmd()
This method returns the behavior an agent’s behavior for a quit request.
protected final void setMigrateCmd(int i, String meth)
This method sets the behavior of an agent for a migrate request. It also sets the name of the method
which should be called after a successful migration.

public final void setMigrateDest(String dest)
This method sets the destination for an agent if it migrates after receiving a migrate request. If not
set, the home address will be used.

protected final int getMigrateCmd()
This method retrieves the behavior of an agent for a migrate request.

These named methods store the behavior into the following fixed variables:

quitCmd
Holds the behavior of the agent for quit request.

migrateCmd
The behavior of the agent for migrate request.

migrateMeth
Stores the name of the method which is called after migration by a migrate request.

migrateDest
The destination the agent migrates to after receiving a migrate request.

We have designed a case-based behavior for quit and migrate requests, i.e. the agent ignores a
quit request in the case of the variable quitCmd contains the constant value IGNORE.

In some cases it can be interesting to know the state of the agent. Therefore, there is the following
method within the agents code:

public final int getState()
The method call returns the current state of the agent.

Once again the programming convention we want to give to the programmer is to integrate ad-
ditional checks of the control flag. This should be done in methods which need a little bit longer
to finish. Therefore the following method is implemented within the agent

protected final synchronized void checkControl()
Checks the control flag ctrFlag

3.5 Programming Conventions

The programmer has to follow some conventions caused by concepts of mobile agents and by
restrictions of the programming language.

A running agent should check for messages periodically. The agent will be informed about new
messages but the agent should decide on its own how to handle the message. Therefore, the
programmer has to estimate at which points during agent execution messages are handled.

The agent server is able to control the execution of agents. Because of the restrictions of Java 2
we use control flags. These flags must be checked periodically. This is done by a method call
(checkControl()) which has to be done by the main execution thread of the agent (this means
within the source code). It is important to do so because the agent server propagates critical
system states by this method, too. The method checkControl() is called every time one calls
the checkForMail() method which handles all spooled messages.

Avoid the use of sleep() calls within your agent source code. This can cause a never ending
execution of the agent. Use instead wait() calls or set the agent in the state Waiting.

There is a special exception AgentExecutionException which could be thrown by any agent
method. This exception is used to get out of the current agent execution, if the agent wants to
migrate or wants to die. Do not handle this exception within your agent source code.

### 3.6 Example

We conclude this chapter with a small example showing agent migration and communication.
The example consists of two agents. The first one, named WhoAgent is a mobile agent that can
be sent out to collect the names of all users that are currently working on the remote platforms. A
mobile agent does not have permission to access the underlying operating system to spawn a new
process executing the who command. Thus, we use another agent, named WhoSystemAgent to
collect these names. This agent is of type SystemAgent.

The communication between WhoAgent and WhoSystemAgent is done via messages. After the
mobile agent has arrived at a platform, it sends a message to the local WhoSystemAgent which
has the name Who@hostname (where hostname is the name of the local platform). The message
is of type “who”. The system agent is waiting for messages of this type, answering with a comma
separated list of all users’ names. The reply message is of type “reply-who”. The mobile agent
receives this reply messages and merges its list of users (that it has found on previous hosts
already) with the new list of users on the current platform. Then, the mobile agent migrates to
the next destination.

First, we show the source code of the mobile agent.

```java
package examples;
import de.unijena.tracy.agent. *
import de.unijena.tracy.agentsystem. *
import de.unijena.tracy.agentsystem.comm. *
import java.io. *

public class WhoAgent extends MobileAgent {
  private MigrationProperties mp = null;

  public void defineAgent() {}

  public void startAgent() throws AgentExecutionException {
    startOnPlatform();
  }

  public void startOnPlatform() throws AgentExecutionException {
    String hostname = getHostName();
    foundHosts = foundHosts.concat( hostname + "", "");
    try {
```
Now, we show the source code of the system agent. Note, that we omit any sophisticated exception handling.

```java
package examples;

sendMessage( "Who@"+hostname, "who", "" );
} catch( MessageQueueException e ) {
    System.out.println( e.getMessage() );
} waiting();

public void finish() throws AgentExecutionException {
    printMessage( "Found the following users: " + allLogins );
    die();
}

protected void systemFailure() {}

protected void handleLastMail( Message inMsg ) throws AgentExecutionException {
    if( inMsg.getTyp().equals("reply-who") ) {
        allLogins = allLogins.concat( inMsg.getContent() );
        /*
         * sort login names
         */
        java.util.Set setOfLogins = new java.util.TreeSet();
        java.util.StringTokenizer t =
            new java.util.StringTokenizer( allLogins, ";" );
        while( t.hasMoreTokens() == true )
            setOfLogins.add( t.nextToken() );
        allLogins = new String();
        java.util.Iterator iterLogins = setOfLogins.iterator();
        while( iterLogins.hasNext() == true ) {
            allLogins = allLogins.concat( (String)iterLogins.next() + ";" );
        }
        /*
         * go to the next platform
         */
        if( route.length > index )
            go( route[ index++ ], "startOnPlatform" );
        else
            go_home( "finish" );
    }
    protected String allLogins = new String();
    protected int index = 0;
    protected String[] route = new String[] { "iaxp13.inf.uni-jena.de" };
    protected String foundHosts = new String();
```
import de.unijena.tracy.agent.*;
import de.unijena.tracy.agentsystem.*;
import de.unijena.tracy.agentsystem.comm.*;

public class WhoSystemAgent extends SystemAgent {
    public void defineAgent() {
    }
    public void startAgent() {
        printMessage("Who agent started");
    }
    protected void systemFailure() {
    }
    protected void handleLastMail(Message inMsg) {
        if (inMsg.getTyp().equals("who")) {
            java.util.Set setOfLogins = null;
            printMessage("I was asked for a list of users.");
            try {
                /*
                * start new process
                */
                Process whoProcess = java.lang.Runtime.getRuntime().exec("who");
                java.io.InputStream is = whoProcess.getInputStream();
                java.io.InputStreamReader isr = new java.io.InputStreamReader(is);
                java.io.BufferedReader br = new java.io.BufferedReader(isr);
                /*
                * parse process input stream and store logins in a set
                */
                String line = null;
                String login = null;
                java.util.Set setOfLogins = new java.util.TreeSet();
                while( (line = br.readLine()) != null ) {
                    java.util.StringTokenizer t =
                        new java.util.StringTokenizer(line, " ");
                    setOfLogins.add(t.nextToken());
                }
                catch( IOException e ) {};
            } catch( IOException e ) {
        /*
        * generate sorted reply string
        */
        String allLogins = new String();
        java.util.Iterator iterLogins = setOfLogins.iterator();
        while( iterLogins.hasNext() == true ) {
            allLogins = allLogins.concat( (String)iterLogins.next() + ";");
        }
        /*
        */
    }
Section 3.6. Example

```java
* send the reply message
 */
try{
    Message reply = new Message( name, "reply-who", allLogins );
    sendMessage(inMsg.getSender(), reply );
} catch (MessageQueueException e) {
    System.out.println(inMsg.getSender() + e.getMessage());
}
```
Chapter 4

Architecture of an Agent-based System

After we saw how to program Tracy agents, 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 to use services of other agent servers using mobile agents of their own. Different types of agent-based systems are distinguish by the way the agent server is connected to other applications on that host. In Fig. 4.1 on the next page 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 picture the Java Virtual Machine and the Operating System, here. All hosts are connected using a network. The four configuration types in the top row are the ones that are actually working, the one in the lowest row gives an idea of what will be possible in future.

The first example (dagobert.uni-jena.de) shows a host on with only a Tracy agent server running. 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 type of system agent. We will show how to start and configure an agent server in Sec. 5.1 on page 41.

An agent server may run on its own for the whole life-time, but can also be temporarily connected to a graphical user interface, as shown in the second example (daisy.uni-jena.de). 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. We will describe the graphical user interface in detail in Sec. 5.2 on page 44. To close a graphical user interface for a period of time during the user does need not to control the agent server make sense for performance reasons. 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 work-around, a graphical user interface could only be executed when the user wants to control or view the agent server explicitly. The following two examples in Fig. 4.1 show hosts on which an application uses an agent server to offer services to foreign agents, or use services of other agent servers using own mobile agents. It is the privilege of applications to start gateway agents (see Sec. 2.2 on page 8) that are able to offer services within the agent server and direct requests to the connected
application. In the first case (goofy.uni-jena.de) there is an application written in the Java programming language that accesses a Tracy agent server using the Java Remote Method Invocation (RMI) technique [35]. In this case, application and Tracy agent server are loosely coupled – the only connection is drawn 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 Fig. 4.1). To protect an agent server against applications, basic security checks are already implemented. We describe the Tracy Remote Interface in Sec. 6.6 on page 55. In the current version of Tracy, it is possible that multiple applications connect to a single agent server in parallel. However, in the current version of Tracy we have not implemented any security mechanism to protect agents from being killed by another application. We will solve this problem within the next version of Tracy by introducing a private agent space for each application.

In contrast, in the second case (donald.uni-jena.de) 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. The TracyAPI is described in Ch. 6 on page 49.

The fifth example in Fig. 4.1 shows a configuration type we are currently developing. 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 not full access as compared to the graphical user interface, described above. The user can start agents, view the agent’s control messages, and kill agents. 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.
Chapter 5

Running a Tracy Agent Server

In this chapter we show how to install and configure a Tracy agent server. The main part of this chapter contains a description of the graphical user interface of Tracy.

5.1 Installation and Configuration

5.1.1 Installation

For the installation of a Tracy agent server some prerequisites are necessary. First, you need a computer system with a network connection based on the TCP/IP protocol. This network connection should be permanent, but also a temporarily connection is practicable. Second, there must be a Java Development Kit (JDK) or a Java Runtime Environment (JRE) installed on your computer system. The latter is sufficient if you only want to start existing agents. If you want to develop new agents, you must have a Java compiler available, which is part of the JDK. Tracy was developed and compiled against the Java Development Kit from Digital for Alpha, version 1.2.2 and the JDK from SUN for Intel, version 1.3. We made good experiences with the new (and very fast) Java compiler by IBM with name jikes in version 1.10, but you do not need this compiler. If you do not have a JDK or JRE, you can download latest versions from the official Java homepage at www.javasoft.com. There you will also find links to third-party providers, if you use other computer systems than Windows NT, Linux, and Solaris. Please refer to their documentation to get help for the installation process.

At last, you need the Tracy package, which can be downloaded from our Web site, which is tracy.informatik.uni-jena.de. The current version is 0.21. Tracy can be obtained in different versions. Basically, you only need the Tracy JAR file and the start scripts. From our Web site you can also download a version which contains the full documentation, and one version which also contains the source code of Tracy. The installation process of Tracy is very easy. You just have to uncompress and extract the delivered package by

```
gzip -dc tracy.tar.gz | tar xf -
```

on a UNIX based system. If you have a Windows NT based computer system, you should download the Tracy package in the ZIP-format, and use tools like unzip to extract the archive. Extracting results in a directory tree which looks like this, provided that you only downloaded the small package.
Part I Chapter 5. Running a Tracy Agent Server

$TRACYHOME/bin/tracy
  /tracy.bat
  /javacc
  /.TracyAuthority
  /tracy.policy
  /examples/WhoAgent.java
    /WhoSystemAgent.java
  /lib/tracy.jar

The variable $TRACYHOME stands for the Tracy root directory. Extensive packages contain more directories. The bin directory contains start scripts for UNIX and Windows NT, and our own shell script to start the Java compiler (javacc). If you are going to change source code of any Tracy package, you should compile them using this script, rather than using any other Java compiler manually. File .TracyAuthority is the keystore file that is used for the TracyRMI-API (see Sec. 6.6 on page 55). File tracy.policy contains the default policy implementation that we will describe in Sec. 5.3. In the examples directory you find the source code of both WhoAgent examples shown in Sec. 3.6 on page 34. In the lib directory you find the Tracy Java archive file.

To start an agent server, type in the following command in the Tracy root directory: bin/tracy. We recommend to insert directory $TRACYHOME/bin in your PATH environment variable, so that you only have to type: tracy to start the agent server. The agent server starts up and prints the following messages on the console.

Tracy Version 0.21 beta, Copyright (C) 2000 Friedrich Schiller University Jena

Tracy comes with ABSOLUTELY NO WARRANTY! This is free software, and you are welcome to redistribute it under certain conditions; see file COPYING for more information. Please consult this URL:
http://tracy.informatik.uni-jena.de/

Starting agent system .... done!

As a default, Tracy is started together with the graphical user interface. If an error occurred during installation, an error message will give hints to the reason of this error.

5.1.2 Command Line Options

As already remarked in the last section, a Tracy agent server is started using a default configuration. This default configuration is described in detail in Sec. 5.1.3. One way to modify the configuration is to use the command line when launching the agent server. Another way is to use so-called configuration files that are introduced in the next section.

To configure a Tracy agent server, sevral properties can be set. Each property has a name and a typed value, which can be a boolean value, a number, or a string. At the command line, properties can be set by giving their name and their (new) value:

tracy { property=value }

By tracy -help you can obtain the complete list of all properties that can be set in the current version of Tracy. In the following we will only explain the most important ones. In the follow-
Section 5.1. Installation and Configuration

-ing \texttt{<string>} stands for a string literal, \texttt{<number>} for a number literal, and \texttt{<boolean>} for a boolean literal (i.e. yes and no, not (!) true and false).

\texttt{propertyfile=<string>} To configure Tracy from a configuration file. See the following section for details. If a configuration file is used, additional command line options overwrite configuration file entries, but they are not saved when Tracy terminates.

\texttt{gui=<boolean>} Start Tracy with or without graphical user interface. TracyGUI can be started later without any disadvantages.

\texttt{net.tcp=<boolean>} This option must be used to initialize the SATP/TCP transmission strategy.

\texttt{net.rmi=<boolean>} This option must be used to initialize the SATP/RMI transmission strategy.

\texttt{net.tcp.port=<number>} This option can be used to specify a different TCP/IP port number for network transmission. The default port number is 4444. This option is often needed, when on the local host the default port number is already engaged. Note, that in the current version of Tracy the TCP/IP port number must be equal on \textit{all} agent servers! In the next version of Tracy it will be possible to use different port numbers on each agent server.

\texttt{rmi.remoteapi=<boolean>} By this property you can define whether the Tracy Remote Interface should be started, as described in Ch. 4 on page 39. The Remote Interface is necessary to control an agent server via RMI method calls. To activate the Remote Interface it is necessary to start the program \texttt{rmiregistry} which is part of the Java Development Kit resp. Java Runtime Environment. It can be started manually (please refer to the documentation of your local Java system for more information) or started by using property \texttt{rmi.registry}, see below.

\texttt{rmi.registry=<boolean>} Using this option starts the program \texttt{rmiregistry} automatically and closes it when the agent server shuts down. If the Tracy Remote Interface should be started, this option is mandatory. This property is automatically set \texttt{yes} if any other property concerning Java RMI is defined.

\texttt{rmi.registry.port=<number>} The default TCP/IP port number used by the program \texttt{rmiregistry} is 1099. If this port number is no longer available, e.g. because any other program uses it, it can be changed using this option. Note, that in the current version of Tracy the RMI port number must be identical on all agent server. Further versions of Tracy will allow to use different port numbers on each host.

\texttt{rmi.keystore=<string>} The name (and path) to the keystore file which can be used by the TracyRMI-API.

\texttt{rmi.keystore.passwd=<string>} The keystore password to access the keystore file.

\texttt{monitor.dispatcher=<boolean>} Defines, whether the monitor dispatcher should be initialized. Without the monitor dispatcher it is not possible to add local monitors. See Sec. 8.5 on page 92 for more information about monitors.

\texttt{monitor.<name>.class=<string>} Defines the name and the class of a monitor.

\texttt{monitor.<name>.init=<string>} Defines the name of a file that contains a collection of message types that this monitor should observe.
Part I Chapter 5. Running a Tracy Agent Server

There are about 20 other properties concerning the graphical user interface, e.g. defining alerts, etc. You can also specify an agent that will be started immediately after agent server initialization:

agent.<name>.class=<string> To define an agent with name <name> with the given class file.

agent.<name>.number=<number> Defines how many agents of the given class should be started. If this option is used, the agent’s name is concatenated with a number, e.g. Webster01.

If no agent.<name>.number property is given, only one agent of this class will be started. For example, using the following command line, Tracy is started without graphical user interface, and two agents with name Webster01 and Webster02 of class examples.Runner are started:

```plaintext
tracy gui=no agent.Webster.class=examples.Runner \ agent.Webster.number=2
```

5.1.3 Configuration

Following the same pattern described in the section above, you can summarize several property entries in a file, in which you can define one property per line. As described, the property (propertyfile) is used to define the property file that should be read in. Note, that this property file is also written when Tracy terminates, so that any modifications you made during the session can be seen in this property file. Of course, any agent.* entries are not changed. If you have used a property file and command line properties, then these command line properties overwrite the ones in the property file but are not saved in the property file.

The default configuration, which is valid if you do not use command line properties or a property file is the following:

```
1 gui=yes
2 net.tcp=yes
3 net.tcp.port=4444
4 net.rmi=no
5 rmi.registry=no
6 rmi.remoteapi=no
7 monitor.dispatcher=no
```

5.2 Graphical User Interface – TracyGUI

A graphical user interface (GUI) for Tracy was developed in a student research project [12]. The agent server can be started with or without this GUI. In this section we will show the use of the GUI and give an overview of its features.

5.2.1 Overview and Concepts

There are several purposes for (creating and) using a GUI. Sometimes, it is easier to have a graphical overview than to decode log-files. Furthermore, it is important to have a user interface, graph-
ical or not, to control the programmed application. So we have developed our TracyGUI and now we can look at our agents, at the blackboard, and at the things going on in the agent system, and we are even able to control these things.

While Tracy is running the user can admin and control the actions and information within the server. It is possible to start a new agent, to stop running agents, to suspend and resume agents, and to send commands (messages) to agents (broadcast possible). To a certain extent the GUI-user can observe the actions of the agents. So he can observe the messages which will be written to the agent-console window by agents and the messages which will be written to the system information and logging window by the system.

With the help of the GUI the blackboard can be read and changed. Files and directories can be created, deleted, and renamed. The content of a file can be viewed and changed. These are the same actions an agent can perform, introduced in Sec. 3.2 on page 22.

Let us have a quick look at the outfit of the TracyGUI (see screenshot Fig. 5.1). Upside you can see a menu bar and a tool bar with icons. The main parts of the GUI are the tree view on the left and the window area on the right. Last but not least there is a status bar at the bottom with an alert lamp and a clock counter. The alert lamp is only displayed if a defined alert event has come to hand.
5.2.2  The Tree View

On the left side in Fig. 5.1 on the page before you can see a tree with two parts – the agent directory and the blackboard directory. The agent directory is a sub tree with all agents within the agent server sorted by agent types. Removed agents that have left the server by a migration or by the end of execution are listed too (number is configureable). The blackboard directory shows the directory tree of the blackboard items, this means all entities on the blackboard which are organized like a file system will be displayed like a directory tree with directories and files.

5.2.3  The Window Area

Within the window area there will be displayed all internal frames (windows). There are several internal windows which will be introduced now.

Agent-Console Window

The agent-console window displays agent messages. The agents can write various messages within this window. The window can be opened via the agent menu or via the correlating icon at the icon bar. If no agent is marked at the agent tree a console window will be opened which displays messages from all agents within the server. If an agent was marked, the console window will only display messages from the marked agent. Now the user is able to observe agent messages.

Server Information and Logging Window

Server messages can be observed within the server information and logging window. Within this window there will be displayed messages for events, e.g. a new agent was started, has entered the system or has left the server. The window can be opened via the agent menu or via the correlating icon at the icon bar.

Blackboard View Window

Via the blackboard menu the user is able to open the blackboard view window. This window shows information about the files within the directory. The file has to be marked in the tree view before the window is opened. If a set of files is marked in the tree view, the view window will display all marked files.

5.2.4  Menu Bar

All functions which are provided by the GUI can be reached over the menu bar. Here is a quick guide through the menu.

- Server Menu
  
The system menu is used to control the agent system and the GUI.
Figure 5.2 Screenshot of the TracyGUI Preferences Dialog

- Agent Menu

Agents can be started and controlled via the agent menu. Within this menu the console and the logging windows can be opened. There is a possibility to define alerts for incoming or outgoing agents.

- Blackboard Menu

This menu provides all functions to show and change the blackboard.

- GUI Menu

Console and logging windows can be cleared and the preferences of the GUI (see Fig. 5.2) can be changed via the GUI menu.

- Window Menu

The window menu can be used to organize the windows within the window area.

5.2.5 Tool Bar

Within the tool bar you can see (Fig. 5.1 on page 45) several icons. These icons are divided into two parts. The left part is relevant for agents and the right part for the blackboard. There are the same functions as in the agent and blackboard menu.

5.2.6 Status Bar

There is a clock counter which counts seconds, minutes, and hours. This is the running time of Tracy. An alarm symbol (lamp) is displayed too if a defined alarm event has come to hand (see Fig. 5.1 on page 45). A window which has more information on the alert will be opened if you click on the alert symbol.
5.3 Security

In the current version of Tracy we have already implemented a solution for one very important security aspect. By activating the Java sandbox model for mobile agents, we can guarantee that mobile agents have no permission to access the agent host system, except of some uncritical and absolutely necessary services. We have not solved the opposite problem of malicious agent servers that might manipulate mobile agents yet.

Central component of the Java security model is a so-called policy file that defines what permissions are open to code of a specific code source. It is intended that all stationary agents have full permissions to read and write the local file system, open graphical user interfaces and open network connections, etc. Mobile agents have also full permissions, if they are executed on their home platform. On a remote platform, no mobile agents has the permission to read or write the local file system. The policy file that is deployed with the current version of Tracy looks like this:

```java
grant codeBase "file:/-" {
    permission java.security.AllPermission;
};

grant codeBase "ftp://*.inf.uni-jena.de" {
    permission java.net.SocketPermission "*:*", "accept, connect, listen, resolve";
    permission java.lang.RuntimePermission "modifyThreadGroup";
    permission java.lang.RuntimePermission "modifyThread";
};
```

This file is named `tracy.policy` and is placed in the bin directory. When a Tracy server is started (using the `tracy` script) this policy file is defined by using the Java command line option `-Djava.security.policy=./tracy.policy`. If you want to use another policy file you must modify the `tracy` script.

The first grant entry gives all permissions to code that is loaded from the file system. This entry is necessary so that the Tracy server works at all. It is safe to grant all permissions to a Tracy server, here (belief us). The second entry grants permission to mobile agents that were defined on home platforms within domain `inf.uni-jena.de`. Only the following three permissions are necessary to start mobile agents from those platforms. Using this policy file makes it impossible to start any mobile agents that were defined on platforms outside domain `inf.uni-jena.de`. Thus, you must configure your policy file in order to be able to start your own agents!
Chapter 6

The Tracy Application Programming Interface

In this section we will show how to use the Tracy agent server as a software component in an application. A Tracy server can be completely managed by the TracyAPI. We will show the principles for using the methods, including important information about the order in which they should be invoked. For sake of simplicity we omit explaining exception handling, and direct the interested reader to the online documentation at our Tracy WebSite tracy.informatik.uni-jena.de/docs/.

6.1 Instantiation of a Tracy Server

To use a Tracy agent server within an application, class Tracy from package de.unijena.tracy must be instantiated. Multiple agent servers might exist on one hardware platform, but must not use identical communication ports. Problems can occur if multiple agent servers try to start a Remote Method Invocation (RMI) server. In this case, only the first agent server is allowed to start such a RMI server. However, all further platforms are to connect to this server, too. They must be configured so that they do not start a server on their own.

The Tracy agent server can be created in two ways:

```java
public Tracy()
Creates a new Tracy agent server with default configuration.
```

This method builds up an agent server with a default configuration, see Sec. 5.1.3 on page 44 for details. If you want to configure the agent server according to your own requirements, you can use the following constructor.

```java
public Tracy( String nameConfigFile )
Creates a new Tracy agent server according to preferences given in the configuration file.
```

The configuration file must be given in a specific format, described in Sec. 5.1.3 on page 44. Calling one of these constructors instantiates the complete agent server with all sub-components according to the given configuration. A RMI server object will be started automatically if necessary. Note, that the agent server is alive immediately after the constructor returns, so that it may happen that agents migrate to this agent server before you have installed a listener object to become notified of this fact. If sub-component installation fails for some reason the whole instantiating process could fail. In case of errors, console printouts will warn you and the instantiation process will be
stopped by throwing an exception. See the following example for instantiating an agent server:

```java
import de.uni_jena.tracy.*;

try {
    Tracy myTracy = new Tracy("./tracy.config");
    // add agents
}
catch( TracyException e ) {
    System.err.println( e.getMessage() );
    e.printStackTrace();
}
```

There are three methods to get some information about the agent server after it was instantiated successfully.

- `public String getVersion()`: Returns the actual version number, currently 0.21.
- `public String getName()`: Returns the agent server's name, currently Tracy.
- `public String getHostName()`: Returns the name of the underlying host system, e.g. dagobert.uni-jena.de.

### 6.2 Creating, Maintaining, and Disposing Agents

In this section we describe all methods that are necessary to control agents.

- `public String startAgentFromFile( String className, String agentName )`: Creates a new agent with name `agentName` from class `className`.

This method creates a new agent in the local agent server. All types of agents can be created by this method. The class name is given by `fileName`, which must be the full-qualified name (i.e. package name and class name) of the class file related to the environment variable CLASSPATH. For mobile agents, it is not allowed that the class file is archived in a JAR file. First, the given agent name is checked whether there is not another agent with this name already. If not, then the agent is instantiated using the default constructor of the class `fileName`. Then the agent is initialized by calling method `Agent.init()` to set the agent's home platform to the current platform and set the agent's name. Note, that a correct agent name consists of the `agentName` and the name of its home platform. This name is used to store the new agent in the local agent directory, and this is the agent's name that you must use from now on to communicate with your agent. This new agent name is the return value of method `startAgentFromFile()`. As a programmer using the TracyAPI you have no access to a reference to any agent. After the agent is inserted in the local agent directory a new thread of execution is created and method `startAgent()` of your agent is called. For the following example, assume that the agent server is started on platform dagobert.uni-jena.de.
Tracy myTracy = new Tracy("./tracy.config");
String myAgent = myTracy.startAgentFromFile("examples.RunnerAgent", "Bond");
if( myTracy.isAgentHere( myAgent ) == false ) {
    System.out.println("Agent " + myAgent + " has already left this platform.");
}

Variable myAgent has value Bond@dagobert.uni-jena.de. Note, that it is possible that your agent has left its home platform before method `startAgentFromFile()` has terminated. In this case you get the correct agent name, but you do not have any access to your agent because it already left the current platform. You can check whether an agent resides on this agent server using method

    public boolean isAgentHere( String agentName )
    Returns true if there is an agent with the given name on this agent server.

The following methods can be used to get basic information about an agent and to control it.

    public int getAgentType( String agentName )
    Returns the type of the given agent. The return value is one of these values: Agent.GATEWAY_AGENT, Agent.SYSTEM_AGENT, or Agent.MOBILE_AGENT.

    public boolean getSuspendedState( String agentName )
    Returns true if the agent is suspended.

    public void killAgent( String agentName )
    Kills the agent with the given name.

    public void quitAgent( String agentName )
    Asks the agent with the given name to quit itself.

    public void suspendAgent( String agentName )
    Suspends the agent's execution.

    public void resumeAgent( String agentName )
    Resumes the agent's execution.

    public void forceMigration( String agentName, String destination )
    Forces migration of the agent with the given name to the given platform.

The next method is to obtain a list of all agents currently residing at the agent server.

    public Vector getAgentNames()
    Returns a Vector object with the names of all agents that are residing in the agent server.

The results contains the names of all agents regardless of their type and in no particular order.

In the following example we first get a list of all agents that are currently in our agent server, and then send each of them a message to leave this server immediately.
// define name of the alternative agent server
String altServer = "pluto.uni-jena.de";

// get all agent names
Vector allAgents = myTracy.getAgentNames();

// get an enumeration of theses names
Enumeration e = allAgents().enumerate();

// run thru all names ...
while( e.hasMoreElements() == true ) {
    // ... and send a message to each agent
    myTracy.forceMigration( (String)e.nextElement(), altServer );
}

6.3 Listeners

The TracyAPI programming model is event-based. The model allows the programmer to plug-in customized listeners into the Tracy agent server. Listeners are informed in case of any events, e.g. agent arrival or agent disposal. Two kinds of listeners exist:

- **Agent listeners** can be used to get informed of any changes concerning agents.
- **Blackboard listeners** can be used to get informed of any changes concerning blackboard entries. Blackboard listeners are explained in Section 6.5.

Agent listeners are objects of class TracyListener and must be registered and unregistered using the following two methods.

```java
public void registerListener( TracyListener tracyListener )
    Registers a new Tracy listener.

public void unregisterListener( TracyListener tracyListener )
    Unregisters a Tracy listener.
```

A concrete listener must override class TracyListener and can override some of the following methods.

```java
public void agentWasStarted( String agentName, int agentType )
    Is called when a new agent was started.

public void agentHasEntered( String agentName )
    Is called when a new agent has entered the agent server.

public void agentHasLeft( String agentName )
    Is called when an agent has left the agent server.

public void agentWasRemoved( String agentName )
    Is called when an agent was removed from the agent server.
```
public void agentSuspended( String agentName )
Is called when an agent's execution is suspended.

public void agentResumed( String agentName )
Is called when an agent's execution is resumed.

public void printConsoleMessage( LogMessage consoleMessage )
Is called when an agent wants to print some message on its console.

public void printLogMessage( LogMessage logMessage )
Is called when the agent server wants to print some message.

In the following example a new listener is registred in the Tracy agent server which only prints
a message when a new agent has arrived. By use of anonymous classes it is very easy to define
a listener object with specialized behavior. If the listener is not as easy as in the example, it is a
good idea to define a new class for the listener by subclassing TracyListener.

```java
Tracy myTracy = new Tracy("./tracy.config");

String myAgent = myTracy.startAgentFromFile("examples.RunnerAgent", "Bond");

myTracy.registerListener( new TracyListener() {
    public void agentHasEntered( String name ) {
        System.out.println("New agent with name " +
                          name + " has entered!");
    }
});
```

### 6.4 Messages

Within Tracy we have implemented communication by messages, see Sec. 2.4 on page 9 and
Sec. 3.2 on page 22. The interface AgentSystemInterface provides the following methods
for sending messages:

- `public void sendMessage( String agentName, Message message )`
  Sends a message to the specified agent.

- `public void sendBroadcast ( Message message )`
  Sends a broadcast message to all connected agents.

These methods are used by the GUI for example to send messages, but no message can be received
by the GUI – agents can only write to the agent console, see Sec. 5.2 on page 44.

### 6.5 Blackboard

We have allready introduced the blackboard concepts in Sec. 2.4 on page 9 and Sec. 3.2 on page 22.
The TracyAPI provides access to the blackboard using the following methods.
public boolean existEntity(String anEntity)
Tests whether or not an entity does exist on the blackboard.

public BlackboardContent getEntity(String anEntity)
Reads information from the blackboard.

public void getEntityWhenChanged(String anEntity,
EntityObserverInterface anObserver)
Puts an observer to a blackboard entity. When something has changed in the entity, the observer will be informed.

public void removeObserver(String anEntity, EntityObserverInterface anObserver)
Removes an observer from a blackboard's entity. When something has changed in the entity, the observer will not be informed any longer.

public void removeObserverAdapter(String anEntity, Object adaptedObserver)
Removes an adapted entity-observer. All RMI-based entity-observers have adapters and must be removed this way.

public void putEntity(String anEntity, BlackboardContent aValue)
Writes information to the blackboard.

public void deleteEntity(String anEntity)
Deletes information from the blackboard.

public Vector getChildsOfFolder(String anEntity)
Lists all entities on the blackboard within a specified folder entity or specified leaf entity (file) only.

public Vector searchEntity(String someKeywords, boolean caseSensitive)
Searches for entities on the blackboard which contain a specified keyword.

public String getRootEntityName()
Gets the name of the root-entity, or null if it does not exist.

public void renameEntity(String entityPath, String newName)
 Renames an entity.

public String getEntityOwner(String entityPath)
Gets the owner of an entity.

The observation of items on the blackboard will be done in a different manner than observing agent activities. An application can pin an observer object to the blackboard entity. This observer object has to implement the EntityObserverInterface from package de.unijena.tracy.agentsystem.comm. Entities do a direct method call to the registered observer object if something has changed in contrast to a observing agent. The following methods have to be implemented by the observer object (methods from EntityObserverInterface):

public void bbNewLeafWasAdded(String folderPath, String leafName)
Called by a parent node entity (folder) when a new entity was added.

public void bbLeafWasRemoved(String leafPath)
Called by an leaf entity when the entity was deleted.
public void bbNewFolderWasAdded(String folderPath, String folderName)
Called by a parent node entity (folder) when a new entity has created in this folder.

public void bbFolderWasRemoved(String folderPath)
Called by a node entity (folder) when the entity was deleted.

public void bbLeafHasChanged(String leafPath, BlackboardContent content)
Called by a leaf entity when the content has changed.

public void bbEntityWasRenamed(String oldPath, String newName)
Called by a leaf or node entity when the entity name was renamed.

public boolean isRMIObserver()
Test whether or not this observer is a remote one.

The last method indicates that there is also a RMI-variant of this interface. This is the RMIEntity-ObserverInterface which works in a similar fashion.

6.6 Working with Remote Tracy Agent Servers

A Tracy agent server can be configured in a way so that it is possible to supervise it from the same or a remote computer system via Java Remote Method Invocation. The main advantage of this feature is that new agents can be added, or existing agents can be killed, even if the agent server was started without a graphical user interface. Without the TracyRMI-API there would not have been any chance to access a running agent server at all.

As already described, a graphical user interface can be connected to a remote agent server. The remote server can be completely controlled and supervised. In the rest of this section we will show how to use TracyRMI-API in your own programs.

First, we will show how the TracyRMI subsystem is installed in an agent server within your application when the property is set as described above. The main class in this context is RMICOnnector which provides the connection between the Tracy agent server and the RMI component. The function of this class is twofold. On the one hand, it is a subclass of TracyListener, so it is able to get informed about internal events. On the other hand it binds a special RMI-able version of the TracyAPI so that it can be accessed via RMI. Let's first look at the following example:

```
String keystorePasswd = "tracykeys";
String keystorePath = ".TracyAuthority";

try {
    Tracy myTracy = new Tracy();
    RMICOnnector theRMICOnnector = new RMICOnnector(
        keystorePath, keystorePasswd, rmiPort, myTracy);
    registerListener( theRMICOnnector );
} catch ( Exception e ) {
    // handle exceptions here
}
```
That is all needed to install the RMI subsystem. In the constructor of class RMIConnector two other objects are instantiated. First, class RMITraceAPI defines the server-side interface to the agent server. This class defines all methods of TraceAPI, but in a way, so that they can be accessed via RMI. The main difference is that each method can now throw exception RemoteException which is triggered in case of any transmission failure. An object of class RMITraceAPI has a reference to the local agent server and just forwards every method invocation to the local agent server. This object is also able to handle listener objects that are not local. The second object which is created during the construction of RMIConnector is of type RMIServerAccess. An instance of this class is bound to the local RMI registry under the name TracyRMIAccess. This class does only define one method:

```java
public RMI AgentSystemInterface login( String alias, String pwd )
Login at a remote agent server.
```

This method must be used by a remote application to connect to the RMI subsystem of the local agent server. The method returns a reference to a stub object of type RMI AgentSystemInterface which is the mirror object of the RMITraceAPI object on the local agent server.

Now, we change the roles. The local server is the one that wants to connect to a remote agent server. Look at the following example:

```java
String hostName = "dagobert.uni-jena.de";
String loginName = "..."; // defined somewhere else
String password = "..."; // defined somewhere else

try {
    RMIServerInterface newRMIServer =
        (RMIServerInterface)Naming.lookup( "//" + hostName + "/TracyRMIAccess" );
    rmiTraceAPI = newRMIServer.login( loginName, password );
} catch( Exception e ) {
    // handle exceptions here
}
```

Let's assume that we want to connect to a remote agent server which lies on host dagobert.uni-jena.de. First, a connection to this host must be opened. The name of the remote server object must be known. As mentioned above this is fixed to TracyRMIAccess. As a result you get a reference to a remote object of class RMIServerInterface. Using method login() you login to the remote agent server and you get a reference to a remote object of class RMITraceAPI. In the last step you can register a listener object. Such a listener object lies on the local server and wants to get informed about remote agent server events. The class which defines the interface for such listeners is RMIAgentSystemLayerListener. To define a new listener you must implement this interface and extend a special Java class from package java.rmi.server.

```java
import java.rmi.server.*;

class MyTracyListener extends UnicastRemoteObject
        implements RMIAgentSystemLayerListener {
    void agentHasEntered(java.lang.String agentName) {
```
You can register this listener simply by using the method `registerRMIListener()` of class `RMITracyAPI`. 

```java
    // add code here

    // define methods for other events here
}
```
Part II

Inside Tracy – Design and Implementation
Chapter 7

Architecture of a Tracy Agent Server

In this chapter we provide an overview of the software architecture of a Tracy agent server. The agent server has a three layer architecture, see Fig. 7.1 on the following page. As in the ISO-O SI architecture model, this makes higher level services independent from lower level, network oriented services. Each higher level builds on the services of the lower level, treating it as a black box with defined interface and services. Layers are, thus, exchangeable components in a framework and allow for adaptation and portability. In the following three sections we describe each layer with its main components and the task it has to fulfill within an agent server. The last section in this chapter deals with an agent server’s component that is vertically spread over all three layers. This component has the function to summarize logging messages and direct them to various output devices.

7.1 Agent System Layer

The Agent System layer (ASL) is the upper layer of Tracy. All agents are managed within this layer. Therefore, the Agent System layer has to provide modules for controlling agents and for agent communication. All tasks related to mobile agent migration are delegated to the Package Manager layer that is arranged below. A detailed description of the ASL can be found in Sec. 8.2 on page 67.

As shown in Fig. 7.1 it is distinguished between agents that are currently active (drawn as circles) and agents that are temporarily suspended (drawn as boxes). There exists an execution thread for each active agent, whereas suspended agents are just passive objects that are waiting to become awake again. Only active agents can communicate to the main component within this layer, i.e. the Agent Manager. As already described in Sec. 2.3, communication to the Agent Manager is controlled by an interface for each type of agent. By this it can be ensured that agents of a specific type can only access those Agent Manager services that they are allowed to use. Suspended agents cannot communicate to the Agent Manager.

The Agent Manager is the core component within this layer and manages the communication between specialized sub-components. The Agent Manager serves like a facade according to the facade design pattern described by Gamma et al. [11]. For an agent is acts like a single object which offers various services. Behind the facade it works like a manager and delegates tasks to specialized sub-components. Using a facade at this point gives the opportunity to exchange sub-components easily.

All agents residing in the agent server are registered in the Agent Directory under their unique name. The Agent Directory ensures that no two agents with identical names are started at one server at the same time. Whenever the Agent Manager needs to direct requests to a specific
agent, it has to ask the Agent Directory for a reference (in the meaning of the Java programming language) to the corresponding object. Each agent is registered in the Agent Directory when it is started on the local host. It makes no difference whether the agent is started locally by the user (or an application etc.), or has migrated to the local agent server. The Agent Directory’s entry exists for the whole agent’s life-time on the local platform. Even when the agent is suspended, the entry remains active. When an agent is killed, dies, or when it leaves the server by migration, the Agent Directory entry is deleted.

The Blackboard component is responsible for all services concerning to the blackboard functionality described in Sec. 2.4 on page 9. All agent-to-agent communication is managed by the Inter-
**Agent Message Handler** (IAMH) component. As already described in Sec. 3.2 on page 22 an agent has to address messages to other agents via their name. The Agent Manager forwards message request to the IAMH which manages message spooling and their delivery. Messages are deposited in a message queue for each agent. The Migration component is responsible to send a mobile agent to a remote agent server. It uses the Package Manager for this task. In the case of a migration failure it informs the Agent Manager to restart the agent.

### 7.2 Package Manager Layer

The **Package Manager layer** (PML) is the intermediate layer in the agent server architecture. The PML is responsible to manage the whole migration process of a mobile agent. If an agent wants to migrate, the ASL uses services of the PML to initiate the migration process. The PML gives feedback to the ASL whether migration was successful, or not. It uses the underlying **Net layer** (see the following section), to transmit data via a network to a remote agent server. In case of an arriving mobile agent, the PML receives the agent from the Net layer in pieces according to our mobility model (Sec. 2.6.3 on page 15). The PML is responsible to compose the pieces and inform the ASL to start the agent. A detailed description of this layer can be found in Sec. 8.3 on page 72.

The PML itself has a three-layer architecture. The upper layer is the **Package Manager** which is responsible to delegate the whole migration process. It uses the **Agent Package Manager** component and the **Migration** component for specific tasks. The mobility model is implemented in the PML, so this layer handles mobile agents in form of mobile units, state objects, and data items. The management of those pieces is done in the Agent Package Manager component, whereas the process of migrating units and data with different migration strategies is placed in the Migration component.

For each mobile agent there exists an **Agent Package Manager** (APM) that maintains the agent’s data and its mobile units during the entire life-time. Such an APM exists on the home platform as well as on the platform the agent actually resides at. In contrast to an Agent Directory entry in the Agent System layer, the APM is not deleted automatically when the mobile agent leaves a platform. On the agent’s home platform the corresponding APM exists during the whole agent’s life-time, even if the agent is currently not active on the home platform, Mobile units or data can be downloaded from this platform. All APM objects are managed by the **Agent Package Manager Directory** in which a reference to each APM is stored under the agent’s name.

The Migration component is responsible to migrate a mobile agent according to a specific Migration Strategy. As described in Sec. 3.3 on page 27 the mobile agent can choose by which strategy it wants to migrate by a name, e.g. "ACTIVE". The Package Manager component uses the **Strategy Factory** to obtain a specific Migration Strategy object according to the given name. The Migration Strategy decides which mobile units and which data items must be transmitted to which destination platform and generates a script in the **Migration Definition Language** (MDL) language. This script is executed by the **MDL Engine** and conducts the complete transmission process. For this task it uses the **Transmission Manager** which establishes the connection to the **Net layer**.

### 7.3 Net Layer

The **Net layer** (NL) is the lowest layer in the agent server architecture. Agent servers communicate using this layer and exchange data. The Net layer is used by the Package Manager layer to exe-
cute specific communication tasks, e.g. to transmit a set of mobile units, together with the state information. Transmitting of data, i.e. an agent or only state information, is always done by only one call of the NL. When a transmission error occurs it gives notice to the Package Manager layer about this. In the case of an arriving agent the NL is responsible to receive all transmission pieces and prepare them to get processed by the Package Manager layer. A detailed description of the NL can be found in Sec. 8.4 on page 86.

The Net Manager component within this layer defines an interface for the specific communication tasks mentioned above. For the network transmission it uses an own transmission protocol, named Simple Agent Transmission Protocol (SATP) which is implemented in the SATP Engine. For the real network transmission the SATP Engine uses specific network transmission components. Each transmission component corresponds to one transmission strategy, compare Sec. 2.5.3 on page 14 and Sec. 3.3 on page 27. Currently there exist two transmission components, i.e. TCP Transmission and RMI Transmission. The first one uses raw TCP/IP communication to transmit data, whereas the second uses the Java Remote Method Invocation technique to do so. Next versions of Tracy will include more sophisticated transmission techniques, e.g. secure transmission using Secure Socket Layer, or compressed data transmission.

### 7.4 Monitor

The Monitor component of the Tracy software architecture is vertically spread over all three layers mentioned above. The Monitor component is not pictured in Fig. 7.1, because it does not have to exist always. A detailed description of this component can be found in Sec. 8.5 on page 92.

The Monitor component is associated to each of the above mentioned layers and is responsible to collect logging messages and forward them to specific user-defined output devices. Within each layer logging messages are produced for a variety of reasons, e.g. when an agent migrates, or an agent arrives. Most of these logging messages are only of interest for people who want to trace the agent server. Each logging message is typed, i.e. messages are grouped and named. Using this type information, a client can filter interesting messages, e.g. messages of a specific component.
Chapter 8

Implementation Details

In this chapter we discuss some implementation details of the Tracy mobile agent system. The intention of this chapter is to introduce the main concepts so that the reader is able to further inform himself regarding more details using our online documentation. Especially, we emphasize those aspects that are necessary to adapt Tracy to varying requirements. In the first section we introduce our design pattern for layer implementation. The next three sections are each dedicated to one of the three layers, and the last section describes how monitors are implemented.

8.1 Layer Architecture and Connectivity

In the last chapter we have introduced the general three-layer software architecture of a Tracy agent server. In this section we will show how this architecture is implemented. A definite understanding of this implementation technique is necessary if complete layers or individual layer components shall be exchanged.

Each layer in the Tracy software architecture is implemented according to the following design pattern. We illustrate this pattern in Fig. 8.1 on the next page, where we show main classes of the Agent System layer and the Package Manager layer and their associations. Each layer consists of several classes which can be divided into two groups. In the first group we summarize all classes that are necessary for inter-layer connection management, event management, etc. In the second group we place classes that implement layer-specific services. In this section we only describe classes of the first group. All classes of the second group are described in the following sections.

Each layer has one object that must be used for all layer management tasks. It acts like a facade according to the facade pattern described by Gamma et al. [11]. This object is the only one that must be created during initialization of a Tracy server. All subcomponents, and all relations between them are created by this facade object. They are also responsible for layer configuration. In the Agent System layer the facade object is of class AgentSystem and in the Package Manager layer it is of class PackageManager.

This facade class must be distinguished from the interface, which defines all layer-related services, i.e. the layer API. In the Agent System layer this interface is named AgentSystemInterface. An instance of this interface class can only be obtained via the layer’s facade object. For the other direction of communication we use events. If a layer wants to inform a higher level layer about changes, it sends an event to all so-called listeners. A listener is an object of a specific class which is able to receive events. Each layer defines an interface which must be implemented as a listener class. In the Agent System layer this interface is named AgentSystemLayerListener, in the Package Manager layer it is named PckManagerLayerListener. Such an interface is
implemented in the layer above, e.g. class PckMngrListener is the implementation of interface PckManagerLayerListener. To manage listener objects each layer possesses an own object. In the Agent System layer this object is of class DispatchAgentSystemListener, and is able to manage several listener objects. In case of an event, this dispatcher objects forwards it to each registered listeners. For the Package Manager layer it is only meaningful to have one listener. Thus, class ProxyPckManagerLayerListener is only able to forward incoming events to one possibly existing object of type PckManagerLayerListener. We now describe most important methods defined in class AgentSystem, which is the fascade class of the Agent System layer. All these methods do exist in a corresponding way in every layer fascade. Layers are built up step by step. Thus, the Package Manager layer must be initialized before the Agent System layer can be initialized. The Agent System layer is built up completcly by constructing an object of class AgentSystem.

```java
public AgentSystem( PackageManagerLayer pml, String name, String version )
Creates an Agent System layer and connects it to the underlying Package Manager layer.
```

This method creates all subcomponents and connects them with each other. During this method the interface object of layer PML is requested using method getLayerInterface(), and the reference to this interface object is distributed to all objects of the Agent System layer that must access the PML API. The PML listener object is instantiated and registered at the PML using
method registerLayerListener().

After the Agent System layer is built up you can request the layer interface object by this method.

```java
public AgentSystemInterface getLayerInterface()
Returns the corresponding API of this layer.
```

You can register a new layer listener object, or delete an existing layer listener object using the following methods:

```java
public void registerLayerListener( AgentSystemLayerListener agentsystemLayerListener )
Registers a new listener object. If it is not allowed to register a new listener, an exception is thrown.

public void unregisterLayerListener( AgentSystemLayerListener agentsystemLayerListener )
Removes a listener object.
```

### 8.2 Agent System Layer

The Agent System layer is the highest layer in the three layer architecture of Tracy, see Fig. 7.1 on page 62. As mentioned in Sec. 7 on page 61, the main components are modules for controlling and for agent’s communication. Furthermore, there is an agent directory within this layer to manage the agents by the Agent Manager. An object of class AgentDirectory contains all agents within the system and generates unique agent names.

The Agent Manager is implemented by the core agent system substantially. This class (Core-AgentSystem) contains the functionality to control and manage agents, and to access the blackboard. For the realization of communication we have integrated the Inter-Agent Message Handler message queue disposer. The disposer manages the message queues, whereby every agent has one, and it takes over spooling and delivering message objects.

In figure 8.2 on the following page you can see the shortly introduced classes, a few other important classes, and the interfaces of the Agent System layer. As aforementioned, the class Core-AgentSystem is the core within the layer. Any calls for the blackboard access are redirected to the blackboard. The class Blackboard is the construct for the blackboard. Files and directories are realised by leaf and node entities. A few interfaces connect the Agent Manager and agents, applications, and the GUI. In the following subsections we try to split the several connections in the Agent System layer and we will explain each component separately.

#### 8.2.1 Using the Agent System Layer API

Using the Agent System layer API means to connect to the agent manager and using the agent system’s functionality. We have designed different interfaces to the agent manager. There is an agent interface AgentInterface in the package de.unijena.tracy.agentsystem. This interface is split into three extended interfaces according to the three available agent types: MobileAgent-Interface, SystemAgentInterface, and GatewayAgentInterface.

Another interface AgentSystemInterface provides access for applications, which use Tracy as a component, e.g. our GUI uses this interface. This interface from package de.unijena.tracy.agentsystem encapsulates functionality for applications, like blackboard access, agent control,
and agent server administration.

```java
public String getHostName()
Returns the name of local host.

public String startAgentFromFile(String fileName, String agentName)
Starts a new agent by loading it from a .class file. This task should be done synchronously (don't return without job completion). This call will be forwarded to startAgentFromFile(String, String, Hashtable) with a null value for hashtable, this means start the agent without any parameters.

public String startAgentFromFile(String fileName, String agentName, Hashtable dic)
Starts a new agent by loading it from a *.class file. This task should be done synchronously (don't return without job completion).

public Vector getAgentNames()
Returns the name of all agents currently present in the server.

public int getAgentType(String agentName)
Request the agent's type.

public boolean getSuspendedState(String agentName)
Request the state of the agent. If the agent is in state SUSPENDED then return true.

public void killAgent(String agentName)
Terminates an agent without asking.
```
public void quitAgent(String agentName)
Ask an agent to terminate. The agent can decide whether it quits execution or not.

public void suspendAgent(String agentName)
Suspends an agent.

public void resumeAgent(String agentName)
Resumes a suspended/freezed agent.

public void forceMigration(String agentName, String destination)
Forces an agent to migrate.

public void sendBroadcast(Message msg)
Sends a broadcast message to connected agents.

public void sendMessage(String agentName, Message message)
Sends a message to the specified agent.

public boolean existEntity(String anEntity)
Test whether or not an entity does exist on the blackboard.

public BlackboardContent getEntity(String anEntity)
Reads an information from blackboard.

public void getEntityWhenChanged(String anEntity,
EntityObserverInterface anObserver)
Puts an observer on the blackboard entity. When something has changed in the entity the observer
will be informed.

public void removeObserver(String anEntity, EntityObserverInterface
anObserver)
Removes an observer from a blackboard entity. When something has changed in the entity the
observer will not be informed any longer.

public void removeObserverAdapter(String anEntity, Object
adaptedObserver)
Removes an adapted entity-observer. All RMI-based entity-observers have adapters and must be
removed this way.

public void putEntity(String anEntity, BlackboardContent aValue)
Writes information to the blackboard.

public void deleteEntity(String anEntity)
Deletes an information from the blackboard.

public Vector getChildsOfFolder(String anEntity)
Lists all entities on the blackboard within the specified folder entity or specified leaf entity only.

public Vector searchEntity(String someKeywords, boolean
caseSensitive)
Searches for entities on the blackboard which contain a specified keyword.

public String getRootEntityName()  
Returns the name of the root-entity, or null if it does not exist.
public void renameEntity(String entityPath, String newName)
Renames an entity.

public String getEntityOwner(String entityPath)
Returns the owner of an entity.

Applications that want to use Tracy not as an enclosed component (internal part of the system) but as an associated component, thus going for loose coupling between application and agent server, can get a dynamic interface to the agent manager. Gateway agents are designed for that purpose. In this case applications can access the agent manager via a gateway agent which itself accesses the agent manager via the gateway agent interface. Programmers may design their own gateway agents and so utilize the dynamics of this interface.

In addition to the AgentInterface, the AgentSystemInterface, and the gateway agents we have designed the MessageQueueDisposerInterface for communication purposes and interfaces for listener which will get server information, messages and blackboard changes. See the following subsection for further information.

### 8.2.2 Being an Agent System Layer Listener

The agent manager sends information about migrating, removing, adding agents, as well as server information, logging information, and console messages from any agents to all connected listeners. Therefore there is a dispatcher class, the DispatchAgentSystemListener, which dispatches all messages from the Agent System layer to all registered listeners. This class implements the Agent System layer listener interface AgentSystemLayerListener. A short description of the methods follows now:

public void agentWasStarted(String agentName, int agentType)
A new agent was started.

public void agentHasEntered(String agentName)
A new mobile agent has entered the server (from a remote agent server).

public void agentHasLeft(String agentName)
A mobile agent has left the system (has migrated to a remote agent server).

public void agentWasRemoved(String agentName)
A agent was removed from the agent server (has ended his execution).

public void agentSuspended(String agentName)
A agent has suspended its execution.

public void agentResumed(String agentName)
A agent has resumed its execution.

public void printConsoleMessage(LogMessage consoleMessage)
The printed agent console message will be forwarded to the listeners.

public void printLogMessage(LogMessage logMessage)
The printed log message will be forwarded to the listeners.

public boolean add(AgentSystemLayerListener listener)
Adds a listener to this dispatcher.
public boolean remove(AgentSystemLayerListener listener)
Removes a listener from the list of this dispatcher.

8.2.3 Message Queue Disposer

The message queue disposer manages the message queues of all agents. We have introduced the concept of this disposer and the sequence of sending and receiving messages in Sec. 3.2 on page 22. We refer to this section for further details.

There is the MessageQueueDisposerInterface to send messages and there is the interface MessageListener for the agents to receive messages and to administrate their own message queue.

We have chosen an extern message queue for a better modularity. So we could change the spooling algorithm, which uses a simple FIFO (first in, first out) data structure at the moment, or we could change to a commercial message system.

8.2.4 Blackboard

In Sec. 3.2 on page 22 we have discussed the blackboard access by agents. For more details on the implementation of the blackboard component, we refer to the online documentation.

8.2.5 Agent Execution

An agent will be started within the agent server by getting its own thread, as soon as the agent is initialized. Hereby it plays no role whether the agent is coming from remote or is started locally. Every agent has one main thread. There can be other threads accessing the agent while setting the newMail-flag or control-flags, but these are active for a short time only.

The main agent thread is stopped after agent execution has ended or an exception is thrown by the agent. If a mobile agent wants to migrate it throws a WantToMigrate-exception by calling the go()-method. If an agent calls die() a WantToDie-exception will be thrown. Both exceptions are extended from the AgentExecutionException which will be caught and handled by the agent’s thread.

We distinguish between threads for stationary agents and threads for mobile agents because only mobile agents can migrate. So we have designed a class AgentThread and an extended class MobileAgentThread. After initializing the thread every agent has got its type specific agent interface – the connection to the agent manager, the agent has got the interface to the message queue disposer for sending messages, and the agent is registered at the message queue disposer as a message listener.

We have integrated into the agent manager a pool of agent threads and a pool of mobile agent threads for performance reasons. If a new agent is started a thread will be taken from the pool. After use the (mobile) agent thread will be put back into the pool and can be reused for other agents. Within the agent directory the reference of the agent and its agent thread is hold for accessing and controlling the agent.

Because of the deprecation of control methods for threads in Java 2, we had to implement our own control mechanism using the hints delivered with Java. So, there are several control flags.
within an agent. These flags have to be checked periodically. This is done mostly automatic, but the programmer has to build in more checks if he wants to achieve better reaction of his agent, see Sec. 3.5 on page 33. The life cycle of an agent is described in Sec. 3.1 on page 19 which gives an introduction to the programming of agents. See Fig. 3.1 on page 20 for possible agent states.

8.3 Package Manager Layer

The Package Manager layer is responsible to manage the whole migration process of a mobile agent. This includes preparation of a mobile agent using a migration strategy, i.e. division in several mobile units, data items, and state information, and to conduct the transmission process using a MDL script. Further, this layer is responsible to manage all tasks that occur during agent execution with regard to our mobility model, i.e. to load missing mobile units or data on demand. On the other hand, this layer is responsible to handle incoming mobile units, data, and state information and assemble a runnable mobile agent that is handed over to the Agent System layer.

In Fig. 8.3 on the facing page the main classes of this layer can be seen. In the top, four classes are pictured that form the Package Manager component shown in Fig. 7.1 on page 62. They are responsible to receive requests by the Agent System layer and delegate them to underlying components. Further, class ProxyPckManagerLayerListener is part of our general layer architecture pattern, and makes it transparent to other components whether there is a listener object that is interested in events generated by the Package Manager layer, or not. Fig. 8.3 contains two classes that belong to the Agent Package Manager component, i.e. AgentPckManagerDirectory and AgentPckManager. Fig. 8.3 does not contain any class of the Migration component, because they are shown in other figures, later on. The other classes shown in Fig. 8.3 belong to the Transmission Manager component, which connects this layer to the underlying Net layer, and receives and handles Net layer events. In the following subsections we describe main classes of this layer in detail. Especially, we show how we implemented the migration process and how to implement new migration strategies.

8.3.1 Using the Package Manager Layer API

To use the services of this layer, first the layer must be created by instantiating class PackageManager. There only exists one constructor for this class. This constructor must receive an array of objects of class Net, i.e. the main class of the Net layer. The services that the Package Manager layer offers are defined in interface PckManagerLayerInterface, and an implementation is given in class DefaultPckManager. According to our general layer architecture, a reference to an object of a class that implements interface PckManagerLayerInterface can be obtained using method getLayerInterface() of class PackageManager. In the following example instantiation is shown and it is demonstrated how a reference to the interface can be obtained.

```java
import de.unijena.tracy.pckgmanager.PackageManager;
import de.unijena.tracy.pckgmanager.PckManagerLayerInterface;

// netTCP is defined somewhere else
PackageManager = PackageManager(  
    new de.unijena.tracy.net.Net[] { netTCP } );

PckManagerLayerInterface pli = pckManager.getLayerInterface();
```
Now, variable pli can be used to access all Package Manager layer’s services.

In Tracy, instantiation of a mobile agent must be performed using method addAgent, if this agent ever wants to migrate using the Tracy mobility model.

    public MobileAgent addAgent( String name, String cname, java.util.Hashtable param )
    Adds a new mobile agent to the Package Manager layer, initializes an instance and returns this instance.

The reason for this is that some tasks must be carried out before such a mobile agent can be instantiated. These include for example to check whether all class files that the agent will ever use are already available on the current platform, i.e. need not to be downloaded from another platform during run-time. This is important because our mobility model assumes that it has a full view on the whole mobile agent when it determines the division into several mobile units. Method addAgent also creates an object of class AgentPckManager for each mobile agent and registers it in the Agent Package Manager Directory. The method receives the name of the agent, the name of the agent’s main class, and a reference to an object of class Hashtable that is passed to the constructor method of the mobile agent. Using this Hashtable object, parameters can be sent to the mobile agent. If all checks were successful a reference to a new object of (super)class MobileAgent is returned. The central method to start the migration process of an agent is:
public void migrateAgent( String agentName, MigrationProperties mp )
Migrates agent to a new host using the given migration properties.

The process that is hidden behind this method is described in detail in the Sec. 8.3.4 on page 80. This method throws an exception of class PckManagerException if an error occurred during migration.

If a mobile agent has died or has been killed, a client of this layer should delete the corresponding agent package manager using this method:

public void stopAgent( String agentName )
Deletes agent from the Package Manager layer.

The current implementation of our agent mobility model allows an agent to load mobile units independently. These requests are directed to the mobile agent interface which, at last, uses the Package Manager layer to execute them. The following two methods define services to load units on demand.

public void loadUnit( String agentName, int unitnumber )
Loads a mobile unit of the given agent and links it into the local agent package manager object. The unit is specified by a number.

public void loadUnit( String agentName, String unitname )
Loads a mobile unit, as above. The unit is specified by a name.

The process behind this is described in Sec. 8.3.4, too.

8.3.2 Being a Package Manager Layer Listener

A component that wants to become notified in case of any events of the Package Manager layer has to implement interface PckManagerLayerListener. This interface only defines the following two methods. The first method is called during reception of an agent’s data. One of the first information that is being sent, is the agent’s name. The following method is called to check in the layer above (Agent System layer) whether this name is valid.

public boolean validateAgentName( String name )
Checks whether there exists an agent with the given name already in this agent server.

The Agent System layer should return false if and only if there exists an agent with the given name in the agent server, already. The second method is called when a new mobile agent has fully arrived at the agent server. After the Package Manager layer has successfully assembled the agent’s units to an executable agent, the agent is instantiated. After that, the following method is called to request starting the execution.

public boolean startNewAgent( MobileAgent ma, String methodName )
Starts the agent by invoking its method with name methodName.

8.3.3 Agent Package Manager

For each agent there exists an object of class AgentPackageManager in package de.unijena.tracy.pckmanager.apm. An Agent Package Manager (APM) is used to manage all internal data structures, like units, serialized data, and state. The AgentPackageManager is the central class
of the Package Manager layer.

All APM objects are managed in the so-called Agent Package Manager Directory which is, simply speaking, a hash table from which a reference to a APM can be obtained by the agent’s name. Please refer to the online documentation to get more information about this component.

According to our mobility model, an APM exists on the agent’s home platform as well as on every platform the agent currently resides at. However, an APM might exist on an agent server even if there is no corresponding agent. This always happens, if an agent leaves its home platform. Although the agent has left the agent server, i.e. it was deleted from the agent directory in the Agent System layer, its APM object remains in the server, because code or data might be downloaded from this platform in the future. So far, we can distinguish two types of APMs. On the agent’s home platform it has type HOME, on other platforms it has type REMOTE. Due to our mobility model, we have to define a third type, that we denote with MIRROR. If an agent package manager possesses this type, it means that a set of mobile units of this agent is stored at the server for possibly being downloaded in the future, cf. 2.6.3 on page 15.

An APM is the main object to store all agent’s code and data information. Thus, this class does not only manage all mobile units, data, and state information, but does also store all class files. Moreover, an APM is also responsible to generate transmission headers, the agent unit definition block (AUDB), and to instantiate the mobile agent object itself, too.

About AUDBs and Headers

Besides mobile units, data, and state information there are two other important elements of our mobility model. First of all, the header is sent at the beginning of each transmission. Here, transmission can be either an agent migration or a unit download. The header contains all necessary information about the agent to become identified at the remote platform. In the current version of Tracy the header consists of the agent’s name, its main class, and some additional information of lower importance. The header is generated by the APM before each transmission automatically. Thus, a MDL script does not contain commands to transmit headers.

Second, the agent unit definition block (AUDB) contains all information about an agent’s mobile units. This comprises their id, name, home platforms, and additional platforms from which they can be downloaded. The AUDB must be sent to each remote platform before any mobile unit is transmitted. It depends on the type of transmission, whether it is necessary to send the AUDB in advance. For example, if the agent shall migrate with all units and state information to a platform on which the agent has not been before, the AUDB must be part of the transmission. In contrast, if the agent migrates back to its home platform, the AUDB must obviously not be sent. As we will show later in Sec. 8.3.5 the type of transmission is controlled by commands in the MDL script. Each command knows whether sending the AUDB is necessary, or not. Thus, the programmer of an MDL script need not to care about this.

Life-cycle of an Agent Package Manager

Let’s now describe the life-cycle of an APM in detail, see Fig. 8.4 on page 77. Please note, that we only mention the most important methods, now. For a full documentation of all methods defined by class AgentPckManager we refer to the online documentation. A new object of class AgentPckManager is created when a new agent name is registered in the APM directory. This happens either when an agent is added to the Package Manager layer by using the addAgent method described in Sec. 8.3.1 on page 72, or when an agent arrives via the network together with
Part II Chapter 8. Implementation Details

its header which contains the agent’s name among other things. The APM object is in state new after instantiation. If the agent is registered in the Package Manager layer using the addAgent method, the mobile agent is defined using the following method.

    public MobileAgent defineMobileAgent( String className, Hashtable parameter )
    Initiates class className and initialize it with the given parameter.

This method reads in all class files the agent uses. After that, we assume that the agent is ready to become executed, so we switch the APM to state running. For the APM it is not important whether the agent is actually executed, or temporarily suspended.

In the other case, if a mobile agent enters the agent server via the network the agent’s data in received in several pieces. First, the agent unit definition block is received and stored in the APM by the following method.

    public void takeAUDB( byte[] buffer )
    Accepts a byte array containing the description of an agent unit definition block.

This method also defines the unit table inside the APM. After the structure of the mobile units is set, the APM is in state defined. Now, mobile units can be received and stored in the APM. This does not change the APM’s state. When the agent’s state information is received, it is stored using the method

    public boolean takeState( byte[] buffer )
    Accepts a byte array containing the description of a state.

The APM is changed to state executable. After closing the connection in which state information were received successfully, the agent can be started using method startAgent(). When a running agent wants to migrate, the APM is changed to state preparing which should indicate that the agent tries to migrate, now. After migration the method stopMigration() is used, either to change the APM back to state running, or to state roaming. In the first case, migration was not successful, and the agent must be started again, as described in Sec. 8.2.5 on page 71. In the latter case, migration was successful, and the new state roaming shows that the agent in no longer in the agent server. However, the corresponding APM still exists. State roaming is important for an APM on an agent’s home platform and an agent’s mirror platform. If the agent comes back to its home platform, the APM changes from state roaming to state executable simply by accepting a new state information. The same is true for mirror platforms, from which mobile units can be downloaded, later. They continue to exist, too. In contrast, agent package manger objects of type remote are immediatly deleted from the agent server after the corresponding agent has migrated successfully. This is the first way to end an APM’s life-cycle. The other way is that the agent kills itself, e.g. if the agent calls the die() method. In this case, the APM is first in state running and then deleted from the APM Directory. In the following two sections we will describe some problems within the Agent Package Manager in detail.

**Saving and Restoring the Agent’s State**

As already mentioned in Sec. 3.3 on page 27, for marshaling and demarshaling of agents we use the Java Object Serialization technique. To give an impression how simple the task of marshaling is, we show the full source code of the corresponding method.

```java
1 private boolean serializeAgent() {
2 ```
```java
/*
 * is there a mobile agent anyway?
 */
if( mobileAgent == null )
    return false;

ByteArrayOutputStream baos = new ByteArrayOutputStream();
try {
    ObjectOutputStream oos = new ObjectOutputStream( baos );
oos.writeObject( mobileAgent );
oos.flush();
} catch( IOException e ) {
    System.out.println("Error while serializing!");
    return false;
}

/* save the result in the internal data structures */
myData.put( "SERIAL", baos.toByteArray() );
myDataState.put( "SERIAL", DATA_VALID );
return true;
```

The most important statement is line 12, where the agent is serialized to an output stream. In line 23 this output stream is converted into a flat byte array. The opposite method, 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. However, in Tracy this simple procedure leads to the following problem. When defining the new mobile agent from a data stream, classes must be downloaded if they are not already on the current platform. For class downloading, we use a new, more efficient technique (that we describe later), which seems to be incompatible with the standard Java deserialization technique. Thus, we operate like the following. First, we instantiate the mobile agent class. This process starts downloading all necessary class files. Second, we instantiate the serialized object with the following method:

```java
private boolean initializeAgent() {

    /*
    * an object of this type must already be instantiated,
    * to guarantee that all classes are already on this host
    */
    if( mobileAgent == null )
        return false;

    /*
    * get the data, if there is no, stop
    */
    byte[] buffer = (byte[])myData.get( "SERIAL" );
    if( buffer == null )
        return false;

    try {
        ByteArrayInputStream bais = new ByteArrayInputStream( buffer );
        ObjectInputStream ois = new TracyObjectInputStream(
            new TracyClassLoader( myHomePlatform,
                Factory.getByteCodeLoader(this) ),
                bais );

        mobileAgent = (MobileAgent)ois.readObject();
    } catch( IOException e ) {
        return false;
    } catch( ClassNotFoundException f ) {
        return false;
    }

    if( mobileAgent == null )
        return false;
    else
        return true;
}
```

This method uses two Tracy-specific new classes to instantiate mobile agents. Class TracyObjectInputStream overrides 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 TracyInputStream this method is overridden, so that our new class TracyClassLoader is used for this task. This class is described in the
Finding Class Files

In this section we describe the new Tracy classloader, that is defined in class TracyClassLoader. Our new classloader has three main purposes. First, each class that is used by a mobile agent, i.e. the class of the mobile agent itself and all classes that are used by instance variables of this class, must be loaded using this new classloader. This is the only way to assure that classes can be downloaded dynamically from the agent’s home or mirror platforms. Second, to transmit class files, the bytecode of this class must be loaded from the respective file or Java archive. Therefore, class TracyClassLoader defines a static method to load the bytecode of a specific class. Third, all classes that are used by a specific class must be determined.

According to the first purposes, class TracyClassLoader must be instantiated. For this, the following constructor must be used:

```java
public TracyClassLoader( String homePlatform, ByteCodeLoader bcl )
Creates a new Tracy classloader which is able to load mobile units dynamically.
```

For the latter two matters, static methods are defined in class TracyClassLoader.

If during execution a class should be instantiated of which there is no bytecode read already, the corresponding classloader is asked to load the class on demand. The Java 2 policy to load class files is the following. First, the system classloader checks whether the class to be loaded is a system class. Second, the system classloader checks whether the specific class has been loaded already, and can now be found in the cache. Third, is looks in all directories and Java archive files that are in the class path. If the class has not been found yet, the system class loader class asks program specific class loaders, in our case one of type TracyClassLoader, to obtain the bytecode needed. Here, the Tracy classloader acts and has two more possibilities to load the bytecode. At first, the Tracy classloader asks the APM, if there is any mobile unit that contains the requested bytecode. If not, it starts dynamic downloading of the requested class (resp. its bytecode) from the agent’s home platform (or any other mirror platform, of course – the policy to determine from which platform the code is loaded actually is not important at this point). For the connection between class TracyClassLoader and the APM, class StdByteCodeLoader is used.

The second purpose of this class is to determine the bytecode of a class, i.e. return a byte array for a given class name. This is implemented in the following method:

```java
public static byte[] getBytecodeForClass( Class c )
Returns the bytecode of the given class.
```

In this method, we use standard Java techniques to first determine the directory path and file name according to the given class. Second, the content of this file is read and stored.

The third purpose of class TracyClassLoader is to determine all classes, that are used by a given class. To use a class means that there is an instance variable, a method parameter or a return value, or even a local variable of this type. There is a method in class java.lang.Class named getDeclaredFields() by which this information should be obtained. Unfortunately, this method is not implemented by Sun, yet. Thus, we had to implement our own method. To determine all classes that are used within a specific class, we used a tool named ByCal, which is developed by Tracy team member Steffen Schlötzer. His tool offers several services to analyze Java bytecode, transform it, and even perform sophisticated control and data flow analyzes on it.
From the bytecode of a class it is very easy to determine all used classes, simply by reading the constant table [22]. This procedure is implemented in the following method:

```java
public static Class[] getClassesForClass( Class c )
Returns an array of all classes that are used within the given class.
```

### 8.3.4 Agent Migration

Now let's look in detail at the process of migrating an agent. The whole process is conducted by the following method which is defined in the interface `PckManagerLayerInterface` and is implemented in the class `DefaultPckManager`.

```java
public void migrateAgent( String agentName, MigrationProperties migProp )
Migrates the agent to a new host using the given migration properties.
```

The migration is performed in four steps. First, the migration strategy is determined using a `String` object that is stored in object `migProp` as an identifier. To get this identifier, the following method of class `MigrationProperties` must be used.

```java
public String getMigrationStrategy()
Returns the name of the migration strategy that should be used.
```

The identifier can be used to obtain a reference to an instance of the class `MigrationStrategy` which implements the requested migration strategy. For this, the class `Factory` in package `de.unijena.tracy.pckgmanager` can be used. This class defines the following method among other things.

```java
public static MigrationStrategy getMigrationStrategy( String identifier )
Returns an object of type `MigrationStrategy` according to the given identifier.
```

Class `Factory` is implemented using the design pattern Factory described by Gamma et al. [11]. All migration strategies must be implemented by overriding class `MigrationStrategy`. Thus, for example, the push-all-to-next strategy is implemented in class `ActiveMigrationStrategy`, whereas the pull-per-class strategy is implemented in class `PassiveMigrationStrategy`. In Sec. 8.3.6 on page 84 we show how to implement new migration strategies. Class `MigrationStrategy` defines three methods.

```java
public MDL migrateFromHomePlatform( MigrationProperties mp, AgentPckManager apm )
Migrates the agent from its home platform.

public MDL migrateToHomePlatform( MigrationProperties mp, AgentPckManager apm )
Migrates the agent back to its home or a mirror platform.

public MDL migrate( MigrationProperties mp, AgentPckManager apm )
Migrates the agent from a remote platform to another remote platform.
```

These three methods correspond to the three ways to migrate an agent, as described in Sec. 2.6 on page 14.

The second step in migrating a mobile agent is, now, to invoke the specific method of the given
object of class MigrationStrategy. The return value of each of these methods is an object of class MDL, which stands for Migration Definition Language. As already noted earlier, MDL is a script-like language that defines how units and data should be transmitted. Refer to the following section to get more information about MDL.

Third, the transmission strategy must be determined. All information about the transmission strategy the agent has chosen can be obtained by calling the following method of class MigrationProperties.

```java
class MigrationProperties {
    public TransmissionProperties getTransmissionProperties() {
        Returns an object of type TransmissionProperties which contains properties of the transmission strategy that should be used.
    }
}
```

Result is an object of class TransmissionStrategy. Now, the task is to determine a Net layer which offers a transmission strategy that matches the agent’s demands. For this, we use class SetNetLayerInterface which, on the one hand, knows all Net layers currently active in the agent server and, on the other hand, is able to determine a Net layer according to the demands given in the TransmissionProperties object.

The last step is now to execute the MDL script obtained by the migration strategy using the following method.

```java
class MigrationProperties {
    public boolean execute(NetLayerInterface net, AgentPckManager apm) {
        Executes the MDL script, i.e. start the net transmission.
    }
}
```

It can be seen that the transmission strategy is a parameter of this method. This method returns true if and only if the transmission was successful. In case of an error, the execute method returns false and the method migrateAgent in class DefaultPckManager throws an exception of type PckManagerException.

### 8.3.5 Migration Definition Language

Migration Definition Language (MDL) is the name of a script-like language that is used to conduct the transmission process. A MDL script is generated by a migration strategy and is executed to control transmission of units, state, and data items. MDL scripts are always used whenever data must be sent via a network. That is true not only for agent migration, but also for unit downloading or data updating. MDL can be seen as a formal language which is defined by a context-free grammar. In this version of Tracy we did not define MDL in this way, but used an object of class MDL to describe the transmission process. Such an MDL object acts like a container of declarations and commands. It is planned to complement the implementation of MDL scripts towards a real language which is interpreted during execution. Scripts using this language will be more human-readable than the current solution. However, the API we will present now will not become obsolete and is a useful tool for automatically generated migration strategies.

In this section, we describe the structure of an MDL object. In the following section, we will show how to program migration strategies using the example of the push-all-to-next migration strategy. As can be seen in Fig. 8.5 on the following page, an object of type MDL consists of several objects of class UnitDesc, each containing the description of exactly one mobile unit, one object of type StateDesc, which contains the description of the state that should be transmitted, and several objects of class Command. Objects of type UnitsDesc exist only if the MDL script is used to migrate an agent from its home platform, because only then it is allowed to define mobile units. The other two parts, i.e. objects of type StateDesc and Command, must exist in every MDL
Part II Chapter 8. Implementation Details

Figure 8.5 Class diagram of MDL.

![Class diagram of MDL](image)

script.

Before we will explain how an MDL script is executed, we will give a concise introduction in how an MDL script is defined. We start with the class `UnitDesc` which contains the description of a mobile unit. A mobile unit always has an numerical identifier and a name. Thus, the only way to create a new object of class `UnitDesc` is to use the following constructor:

```java
public UnitDesc( int id, String name )
Creates a new object with id and name.
```

Mobile units can contain several **code elements** and several **data elements**. Therefore, class `UnitDesc` defines two methods to add elements.

```java
public void addCodeElement( String command, String parameter )
Adds a new code element to the unit description.

public void addDataElement( String name )
Adds a new data element to the unit description.
```

In the method `addCodeElement` the parameter `command` is used to define what type of code element is stored, whereas `parameter` defines the content of the code element. In the current version of Tracy only complete class files can form a code element. In this case the first parameter must contain the command "CLSS" and the second parameter contains the name of the class, as shown in the following code line.

```java
aUnitDesc.addCodeElement( "CLSS", "examples.DanielWebster" );
```

In the method `addDataElement` the parameter contains the name of the data item to add. In the current version of Tracy only the serialized agent itself can be accessed, but no single variables for example. Thus, the only parameter useful is `SERIAL` to indicate that all agent variables should be added to this unit. The class `StateDesc` defines the structure of a state description. The state consists of some data items, the name of the class that will be initiated first on the remote platform, and the name of the method that should be invoked. Thus, the constructor is the following.

```java
public StateDesc( String className, String command, String methodName )
Creates a new StateDesc object.
```

In the current version of Tracy only the complete serialized agent can be sent within the state, so
the only useful command name is "SERIAL". In the next version of Tracy it will be allowed to choose specific data items to be members of the state. Objects of type StateDesc and UnitDesc are stored in a MDL object using the following method.

```java
public boolean addUnitDesc( UnitDesc aUnitDesc )
Adds a new unit description object.
```

```java
public boolean setStateDesc( StateDesc theStateDesc )
Defines the state description.
```

The last part of a MDL script are objects of type Command. A command defines which units shall migrate to which agent servers, and to which server the state shall be transmitted. Class Command is a container for all types of commands that exist. A Command object is completely defined by its constructor. The most important thing to know about Command is that there are currently six different commands that can be used. Each command is identified by a number within the constructor of a Command object. It depends on the type of command what other parameters are necessary to construct a Command object. We will now explain each command in detail.

**ACTIVE** This command must be used to transmit mobile units and state information to the destination agent server. The second parameter of the constructor must contain the name of the destination platform. The third parameter must be an array of integer numbers containing the identifiers of those units that shall be transmitted.

**PASSIVE** This command must be used to transmit only the agent’s state to a destination agent server, if the remote server does not hold any information about the agent, yet. The second parameter of the constructor must contain the name of the destination platform.

**PREFETCH** This command must be used to transmit only units to a destination agent server. The second parameter of the constructor must contain the name of the destination platform.

**STATE** This command must be used to transmit only the agent’s state. It should be used only if the destination agent server already holds information about the agent. The second parameter of the constructor must contain the name of the destination platform.

**LOADUNIT** This command must be used to load mobile units from a home or mirror agent server. The second parameter of the constructor must contain the name of the platform, from which the units shall be loaded. The third parameter must contain an array of integer numbers containing the identifiers of the units to be loaded.

**MIRRORUNIT** This command must be used to define the current agent server to become a mirror agent server. The second parameter must contain an array of integer numbers containing the identifiers of those units, that shall be mirrored here.

A Command object is stored in a MDL object using the following method.

```java
public boolean addCommand( Command aCommand )
Adds a command to the MDL script.
```

When the MDL script is executed, first the unit definition and state description are stored in the APM. Then, all commands are executed sequentially in the order in which they were added to the MDL script. In the following section we give an example how to program a MDL script.
8.3.6 Implementing Migration Strategies

In this section we will show how to implement migration strategies using MDL scripts, and how to register new migration strategies in the Factory. We use the push-all-to-next strategy as an example and split the source code of the whole class into the methods. We only show those statements that are important to understand how to generate MDL scripts. All additional statements for checking parameters etc. are omitted.

The first examples show how to generate a MDL script for transmitting the whole agent with all classes and state information to the destination agent server.

```java
public MDL migrateFromHomePlatform( MigrationProperties mp,
                                       AgentPckManager apm ) {

    String method = mp.getMethod();
    String dest = mp.getNextDestination();

    try {
        int unitid = 0;
        String[] classNames = null;
        MDL aMDL = new MDL();

        /*
         * get names of all classes
         */
        classNames = apm.getClassNames();

        int[] mustUnits = new int[ classNames.length ];
        int mustUnitsIndex = 0;

        /*
         * create one unit description (i.e. unit) per class
         */
        for( int i=0; i<classNames.length; i++ ) {
            if( classNames[i] != null ) {
                UnitDesc aUnitDesc = new UnitDesc( unitid, classNames[i] );
                mustUnits[ mustUnitsIndex++ ] = unitid;
                aUnitDesc.addCodeElement( "CLSS", classNames[i] );
                aMDL.addUnitDesc( aUnitDesc );
                unitid += 2;
            }
        }

        /*
         * create state description and add it
         */
        StateDesc aStateDesc = new StateDesc( apm.getClassname(),
                                               "SERIAL", method );
        aMDL.setStateDesc( aStateDesc );

        /*
         * create the command and add it
         */
        Command aCommand = new Command( Command.ACTIVE, dest, mustUnits );
        aMDL.addCommand( aCommand );
    }
}```
return aMDL;
}
catch( MDLException e ) {
  System.out.println( e.getMessage() );
e.printStackTrace();
  return null;
}

In the push-all-to-next strategy, to migrate from one host to another is equal to leave the home platform. All units and the state must be transmitted, so we do not present the source code for method migrate() here. The only difference is that no unit declaration is necessary in this case. Now, we show how to generate a script to migrate back to the home platform.

To learn more about programming migration strategies you should look at the example files in package de.unijena.tracy.pckgmanager.migration. To make a new migration strategy usable for all mobile agents, you must register it using the command line property:
pckgmanager.migration.strategy.<name>.class=<string>, where <string> is the name of the class.
8.3.7 Receiving Net Layer Events

As already indicated in Ch. 7 on page 61, it is possible that below the Package Manager layer several Net layers exist. The advantage of this is that an agent can dynamically decide by which transmission strategy it wants to be sent to the destination platform. For the other direction of communication, i.e., receiving an agent from a remote platform, several communication steps must be performed from one Net layer to the single Package Manager layer. Because several Net layers might access the Package Manager layer in parallel, the Package Manager layer must be able to assign each communication step to exactly one Net layer. In this section, we only describe our solution of this problem. All events that a Net layer uses to transmit data to the Package Manager layer are described in Sec. 8.4.2 on page 88.

We solved this problem by using a session identifier. When a Net layer wants to transmit data to the Package Manager layer, it must first obtain a session id, that is a number by which a transmission can be identified uniquely. All individual data transmissions (units, state, etc.) must be marked by the session id to assign them to one specific session. After the transmission has been completed, the session id must be released so that it can be used for another transmission again. The class that implements the Net layer listener interface is NetListener in package de.unijena.tracy.pckgmanager. The two methods responsible for session management are:

```java
public int getSessionId( String hostname )
A connection was opened from host hostname, receive a valid session id.

public void freeSessionId( int sessionId )
After the connection was closed, give the session specific identifier back.
```

The whole session management is implemented in class SessionId.

8.4 Net Layer

The Net layer is responsible for transmitting the elements of our migration model to a destination agent server. The Net layer is used by the Package Manager layer when an agent wants to migrate to another platform, mobile units must be downloaded, or data items must be sent to the agent's home platform. We have identified seven different useful transmission types. Each is defined by one method in the Net layer API, which is described in the following section 8.4.1. The Net layer is completely implemented, even for those parts of our mobility model that are not realized in the Package Manager layer (e.g., data downloading), yet. On the other hand, the Net layer is also responsible to receive data from other agent servers and forward requests to the Package Manager layer. Each Net layer has a three-layer architecture, as shown in Fig. 8.6 on the next page. In the top there is the interface layer, which builds the connection to the upper Package Manager layer. As usual in our general layer pattern, there is a class ProxyNetLayerListener which makes the existence of a Package Manager layer transparent to components of the Net layer. In the middle, there is the class SATPConnection which implements the finite state machine for our Simple Agent Transmission Protocol (SATP). On the lowest level, there are classes responsible for the actual transmission. As we already noted, several Net layers might exist in parallel. Each Net layer implements exactly one transmission strategy (see Sec. 2.5.3 on page 14). Thus, if the agent server should offer several transmission strategies in parallel, multiple Net layers must exist. Different implementations of a Net layer might distinguish only by the classes on this lowest level, e.g., TCP or RMI, but can also differ in other components. For example, if a Net layer realizes the SATP/TCP
transmission strategy, the classes TCPConnection and TCPServer are used. The other transmission strategy (SATP/RMI) is realized by classes RMIConnectionAdapter and RMIServer (and some other auxiliary classes, whose names also begin with RMI...).

In the current version of Tracy, these two transmission strategies are the only ones implemented, yet. We are currently working on some further transmission strategies, e.g. to transmit data in a secure way, using the Secure Socket Layer technique.

### 8.4.1 Using the Net Layer API

The Net layer API defines seven methods to send data to a destination agent server. All methods have as the first parameter an object of type `String` which contains the name of the destination platform. The second parameter holds the `header` which contains the agent’s name and some further information needed by the destination agent server to decide whether to accept the transmission, or not. Additional parameters contain transmission type specific data.

The first method can be used to sent data to the destination agent server with the aim to start the agent immediately over there. The destination server has no information about the new agent before.
public void pushActive( String host, byte[] header, byte[] audb, byte[][] units, byte[] state )
Transmits all parts of our mobility model: header, agent unit definition block, units, and state.

The following method is used to send only some or all units but no state information.
public void pushPrefetch( String host, byte[] header, byte[] audb, byte[][] units )
Transmits header, agent unit definition block, and units.

The next method is used to send only state information. This transmission makes sense, only if
the destination platform has already received the agent unit definition block.
public void pushState( String host, byte[] header, byte[] state )
Transmits header and state information.

If the destination server does not hold any information about the given agent, the following
method can be used to send only the agent unit definition block and state information. Necessary
units might be downloaded later.
public void pushPassive( String host, byte[] header, byte[] audb, byte[] state )
Transmits only header, agent unit definition block, and state information.

These three methods are responsible to load units, and to load and save data items.
public byte[] loadUnit( String host, byte[] header, byte[] uid )
Transmits the header and a unit description.

public void updateData( String host, byte[] header, byte[] data )
Transmits the header and some data items.

public byte[] loadData( String host, byte[] header, byte[] did )
Transmits the header and the description of data items.

The last two methods are used to check whether this Net layer’s properties match the ones de-
manded by the agent. They are used by class SetNetLayerInterfaces which was already
described in Sec. 8.3.4 on page 80.
public boolean matchesProperties( TransmissionProperties tp )
Checks whether the layer properties match the agent's demands, i.e. the layer can have further
properties not demanded by the agent.

public boolean equalsProperties( TransmissionProperties tp )
Checks whether the layer properties are exactly equal to the agent's demands. The layer must not
have further properties defined.

8.4.2 Being a Net Layer Listener

As we already described in Sec. 8.3.7 on page 86, the Package Manager layer implements an
interface NetLayerListener to receive Net layer events. The mechanism of session identifiers
was introduced in Sec. 8.3.7 already.

This interface defines the following methods:
Section 8.4. Net Layer

```java
public int getSessionId(String hostname)
A connection was opened from host hostname, receive a valid session id.

public void freeSessionId(int sessionId)
After the connection was closed, give the session specific identifier back.

public void validateHEADER(int sid, byte[] buf, String hostname)
A header was received.

public void validateAUDB(int sid, byte[] buf)
An agent unit definition block was received.

public void validateUNIT(int sid, byte[] buf)
A unit was received.

public void validateSTATE(int sid, byte[] buf)
A state information block was received.

public byte[] performLOADUNIT(int sid, byte[] buf)
A request to return a unit was received.

public void performUPDATEDATA(int sid, byte[] buf)
A request to update data at the local agent server was received.

public byte[] performLOADDATA(int sid, byte[] buf)
A request to return data items was received.

public void performQUIT(int sid, byte[] buf)
A connection was closed correctly.
```

8.4.3 Simple Agent Transfer Protocol Layer

The Simple Agent Transfer Protocol (SATP) is a simple protocol to transmit parts of our mobility model in a correct way. For example, the protocol assures that a transmission is started by sending the header, or that units cannot be sent without an agent unit definition block. Fig. 8.7 on the next page shows the finite state machine (FSM) that implements our transfer protocol. We will not explain all states of the protocol, because it is very easy to understand. Besides, our transmission protocol is not a point of change – it should not be necessary to insert new states or delete states in the FSM. It should be mentioned that the state *Established* is the central state after opening a connection. In this state, units can be requested (event *loadUnit*), data can be requested (event *loadData*), and data can be updated (event *sendData*) in any order and even multiple times in succession. Besides, other transmission types can be invoked by other events. If an error occurred, the FSM changes to the state *Error*, from which it can only switch to the final state.

The implementation of the SATP protocol as a finite state machine is completely encapsulated in class *SATPConnection* in package *de.unijena.tracy.net.satp*. This class defines methods for all events that can reach the finite state machine. Each state is modelled as an own class, of which only one object can exist (using the pattern singleton), and that can handle all events that can reach the finite state machine. It does that not by defining separate methods for all events but by overriding the abstract class *SATPState*, which has empty method bodies for all events. It provides a new implementation only for those methods (means events) in which the state has to do something. This way to implement a finite state machine is called *state pattern*, and is described by Gamma et al. [11] in more detail. An object of the class *SATPConnection* has a reference to
the actual state and forwards every method call to the actual state object.

### 8.4.4 Connection Layer

The connection layer is responsible for opening and closing network connections and transmitting commands and data via an open network connection. The layer consists of an interface `NetConnection` that is used to transmit data to a remote agent server, and an abstract class `NetServer` that is the corresponding server object. At the time being we have two implementations of the connection layer: TCP and RMI. In Sec. 8.4.5 we show how to provide a new transmission strategy by implementing a new connection layer. Classes in this layer are used by an object of class `SATPConnection` to send data to a destination server and to accept incoming connections.
NetConnection

Classes that implement the interface NetConnection are responsible to perform a complete net transmission. The standard protocol for this is to open a connection first, then send commands to the remote server and receive server replies many times. Finally, the connection is closed. The interface defines methods for theses four stages in a typical communication.

```java
public void openConnection( String host )
Opens a connection to a server.

public void sendCommandToServer( String cmd )
Sends a command to the server.

public void sendCommandToServer( String cmd, byte[] parameter )
Sends a command with a parameter to the server.

public void closeConnection()
Closes the connection.

public void getReply()
Reads in the reply of the server. Always use this method to read in the server reply.

public String getReplyCommand()
Reads in the last reply command. Use this method only after getReply().

public byte[] getReplyParameter()
Reads in the last reply parameter. Use this method only after getReply().
```

NetServer

The class NetServer defines methods to start a server object as a new thread of execution. The method startServer() is defined abstract and is supposed to be overridden by subclasses. It is called when launching the server object.

```java
protected abstract void startServer()
Starts the server object to accept connections.
```

8.4.5 Implementing new Transmission Strategies

To realize a new transmission strategy, it is necessary to implement both classes mentioned in the last section, i.e. NetConnection and NetServer. Further, the class Net must be adapted, so that a specific Net layer can be generated according to the given transmission strategy. This includes to offer an object of the class TransmissionProperties, which is used by the methods matchesProperties() and equalsProperties() to compare the layer’s properties with the agent’s demands. Last, the main class of Tracy, i.e. de.unijena.tracy.Tracy must be adapted so that this new Net layer can be used.
8.5 Monitors for Viewing and Debugging

We have implemented in Tracy a feature that we call monitors by which a user has the possibility to view and inspect a running agent server. This feature is not interesting for people who run Tracy as a production system, but is designed for programmers who have modified internals of Tracy and now want to control whether the agent server behaves as assumed. In the following sections we describe the programming interface, the design of our monitor concept, and programming of client monitors.

8.5.1 Programming Interface

Our concept of monitoring an agent server’s activities is based on typed messages that are produced within Tracy components and which can be displayed on various output devices. The type of a message is given by a string, e.g. net, to summarize all messages that are produced within the Net layer. Additionally, types can have subtypes, e.g. net.tcp or net.rmi. Subtypes can also have subtypes, and so on. This name is used at two points. The message producer, usually any component, sends a monitor message by invoking the static method of the class TracyMonitor:

```java
public static void log( String type, String message )
Prints a message of type type and content message.
```

The second point where the message type is important are the monitor clients. The monitor clients are those output devices that are able to display (save, print etc.) monitor messages. All these monitor clients are able to be configured so that only messages of special types are displayed. This message type filtering makes it possible to get rid of unimportant information and only observe a subset of all monitor messages. A monitor client can be configured so that all messages that have a given type or a subtype of this type, are displayed. For example, a client might be configured to display all messages that have type net.*. This means that all messages of type net are displayed and also all messages with subtypes of net, i.e. net.tcp, net.rmi, etc.

The second feature of our monitor programming interface is to measure periods of time and display elapsed time. Assume that the execution time of a method should be measured. Let’s start by describing the programming interface for the simple case that an object wants to print elapsed time messages, and it is assured that only one object of this class exists at a time. The two methods of interest are:

```java
public static void startTimer( String type, int number )
Initializes a timer of type typ and id number.

public static void stopTimer( String type, int number )
Stops timer with type type and id number.
```

As a message, a timer has a type and additionally a number. Thus, multiple timers in one type scope can be used in parallel. Within the printMsg method described above, the content of a timer can be accessed simply by inserting its type and number information in the message string. See the following example:

```java
1 public class C {
2
3   public void foo() {
4```
Section 8.5. Monitors for Viewing and Debugging

The type and the number of the timer whose content should be displayed within this message are connected and enclosed with the percent character. To print the percent character use `%%`.

Now we have to make some notes about the case that multiple instances of a class exist. Assume, there would exist two instances of the class in the example above. Object A executes the statement in line 5 which defines the timer with type `'net.tcp.openConnection'` and number 1. While in this object statements between the two timer-statements are executed, object B executes the first statement. This leads to the problem that the timer with number 1 is *redefined* with the time stamp of thread B. This problem occurs because there is a single object which uses an object of class `Hashtable` to manage all timers. If object A reaches the statement in line 9 the wrong start value of this timer is used to calculate the elapsed time. As a solution, for objects where multiple instances may exist, no constant timer number must be used. Instead, a timer number which is guaranteed to be identical for all objects. Our suggestion is to use the number obtained by method `hashCode()` which is defined in class `Object` and so is available in every object. Look at the following example:

```java
TracyMonitor.startTimer( "net.tcp.openConnection", this.hashCode() );
// do some work here
TracyMonitor.stopTimer( "net.tcp.openConnection", this.hashCode() );
TracyMonitor.printMsg( "net.tcp.openConnection",
                 "This operation took: %net.tcp.openConnection1% ms." );
```

Multiple timers in identical objects can also be realized by modifying the object's hash code.

### 8.5.2 Design

In this section we give a concise overview of the design of our monitor component. As already noted in Ch. 4 on page 39, the monitor component is spread vertically over all three layers of the Tracy agent server software architecture. In Fig. 8.8 on the next page you can see the general architecture of our monitor component.

As a monitor is for viewing and debugging a Tracy agent server during the development of new features, it is desirable (for performance reasons) to deactivate it in a production system. For this, we introduced a property entry with name `monitor.dispatcher` which must be set to `yes` to activate, and `no` (default value) to deactivate the monitor component. This property can be defined when starting a Tracy agent server, and cannot be changed in a running server anymore, cf. Sec. 5.1.2 on page 42. However, for the programmer it is transparent, whether the monitor
component is launched, or not. Any logging message can be deposited using the static method `log()` in class `TracyMonitor`, as shown above. If the monitor component is installed, all logging messages are automatically forwarded to an instance of class `MonitorDispatcher`. In the other case, logging messages are ignored.

Class `MonitorDispatcher` implements all necessary methods to manage timer and format output messages. Additionally, this class manages a list of all registered client monitors. To decouple the main thread in which logging messages are produced from distributing these messages to all clients, we use a separate thread for this task (`MonitorThread`). This monitor thread uses a queue to manage all incoming messages.

A client monitor must implement the interface `ClientMonitorInterface`. As most clients should be configurable in a way that only those messages are shown, whose type matches a specific pattern, we introduced an adapter class `ClientMonitorAdapter`, which provides methods to filter messages with regard to their type, etc.

### 8.5.3 Programming Client Monitors

The simplest way to implement a new monitor client is to extend class `ClientMonitorAdapter` and override method `printMSG()` as shown in the following example.
package de.unijena.tracy.monitor;

import java.io.*;
import java.util.*;

/**
 * creates a monitor writing messages to CRT
 */
final public class TextMonitor extends ClientMonitorAdapter {

    public TextMonitor(String File_Filter) {
        super( File_Filter );
    }

    public TextMonitor() {
        super();
    }

    public void printMSG( String typ, String message ) {
        if( printTyp( typ ) == true )
        {
            String newtext= getDate() + " - " + typ + ": " + message;
            System.out.println(newtext);
        }
    }

    Method getDate() is defined in class ClientMonitorAdapter and returns a string with date and time. This class can be configured by a file name (given as parameter in the constructor). Such a file consists of several lines, where each line contains a type or a pattern of a type. See the following example:

    tracy
    net.*

    The first line declares that all messages of type “tracy” should be displayed. The second line declares that all messages whose type begins with “net” should be displayed. This includes “net.tcp”, “net.ssl”, “net.ssl.satp”, etc.

    To install a client monitor you must perform the following steps when launching a Tracy agent server:

    1. activate the monitor dispatcher using property monitor.dispatcher=yes
    2. register a client monitor using property monitor.*.class=classname; replace “*” with an arbitrary name and replace “classname” with the class name of your client monitor; all client monitors must be in package de.unijena.tracy.monitor
    3. configure your client monitor by a file name using property monitor.*.init=filename; replace “*” with the name of your monitor and replace “filename” with the name of the file

    We refer to the online documentation for more information.
Chapter 9

Conclusions and Future Work

In this technical report we described our mobile agent system Tracy. We showed the basic principles of our mobile agent system and described how to program agents. We gave a general idea of design and implementation aspects and guided the reader to implement own migration strategies and transmission strategies.

As we have already noted, Tracy is a system which is still under permanent development. Therefore, this report could only capture an interim state of Tracy. Several improvements are currently under development, others are scheduled for an upcoming version.

The following list includes improvements that are almost completely implemented:

- Implementation of new transmission techniques, as for example SSL (Secure Socket Layer), UDP for a fast (but unreliable) data transmission, SMTP (Simple Mail Transfer Protocol) to send agents via electronic mail attachments
- Agent transmission via Remote Method Calls, class downloading using the build-in RMI features
- Implementation of new migration strategies, e.g. one to transmit only the agent’s class (other classes must be downloaded on demand)
- Optimization of the transmission process
- Completion of our migration model implementation (own data serialization)
- Client monitors can connect to a running agent server via RMI

This version is scheduled for December 2000.

The following improvements are scheduled for the next major release of Tracy:

- Agent server naming, so that multiple agent servers can be started on one platform; to migrate an agent, only the symbolic name of the destination server must be known (not the hostname and port number)
- Blackboard persistence
- Blackboard security
- Agent persistence, i.e. to be able to freeze an agent, save it on disk, and reactivated it later
• Implementation of the Migration Definition Language as script language which can be used by the agent itself to configure the migration process
• Mobile unit cache, so that mobile units can be shared between several agents of the same type
• All Net layers will become multi-threaded, so that several agents can be received and sent in parallel
• Tracy domain management by agents
List of Figures

2.1 Example for a Tracy infrastructure, consisting of three platforms. .................. 8
2.2 Classification Of Tracy Agents ....................................................... 9
2.3 Architecture of a Tracy agent server from the agent programmer’s point of view. 10
2.4 Overview of migration strategies. ................................................... 12
2.5 Collaboration diagrams for the four migration strategies described in this section. 13

3.1 State diagram for Tracy agents. .................................................... 20

4.1 Five configuration types of an agent-based system. .................................. 40

5.1 Screenshot of our TracyGUI ......................................................... 45
5.2 Screenshot of the TracyGUI Preferences Dialog .................................... 47

7.1 Overview of the Tracy architecture. ................................................ 62

8.1 Class diagram to illustrate layer connectivity. ...................................... 66
8.2 Class diagram of the Agent System layer. ......................................... 68
8.3 Class diagram of the Package Manager layer. ...................................... 73
8.4 State diagram of class AgentPckManager ......................................... 77
8.5 Class diagram of MDL .................................................................... 82
8.6 Class diagram of the Net layer. ......................................................... 87
8.7 State diagram of the Simple Agent Transfer Protocol. ............................ 90
8.8 Class diagram of the Monitor component. .......................................... 94
References


REFERENCES


[27] Holger Peine and Torsten Stolpmann. The architecture of the Ara platform for mobile agents. In Rothermel [30].


