Technische Universität München
Institut für Informatik

Diplomarbeit

Redesign and Implementation of a Mobile Agent System compliant with the MAFFinder part of the MASIF Standard

Thomas Letsch

Aufgabensteller:	Prof. Dr. Johann Schlichter
Betreuer:	Dr. Michael Koch
	Dr. Andres Marin Lopez (Universidad Carlos III de Madrid)
Abgabedatum:	15.08.2000

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.08.2000 _____

Acknowledgements

First of all I want to thank my parents for financially and personally supporting my studies and for enabling my stay here in Madrid.

I am also very grateful to my advisor, Dr. Andres Marin Lopez, for his support and guidance through my time here in Madrid and for his many valuable ideas during our discussions. Also I want to thank Prof. Johann Schlichter and Dr. Michael Koch from the TUM for their advice and for providing me with the official framework for this diploma thesis in Spain.

Many thanks to all my friends here in Madrid for supporting my thesis with their comments and ideas, and for their patience while I was busy with working. They all provided me with an excellent atmosphere to work and live during this year in Spain.

Abstract

This thesis shows the design and development of a Mobile Agents System based on the standard of the Object Management Group (referred to as OMG) called MASIF [MASIF98] (Mobile Agent System Interoperability Facility).

It will first describe what an Agent is and, more specifically, what a Mobile Agent is. Since Mobile Agents are a very new part of computer science, the most accepted terminology and specification will be used in this thesis. The theoretical work is mainly done by the OMG, the FIPA (Foundation for Intelligent Physical Agents) and universities doing research in this sector.

The next main topic is the description of the former version of the *TAgents* (*Traveling Agents*) platform. At the Universidad Carlos III de Madrid, where this project is processed, an Agent System was already built when this thesis was started. Due to some reasons mentioned in the Section 5, it was decided to write a new System with the improved knowledge of Mobile Agents available now.

The main part of this document treats the new System. It will describe the requirements given to this work. Then, it will give an overview of the design developed in relation to the specification. This section will talk mainly about the module-based structure that helps keeping the platform itself highly extendable. The next section gives a detailed insight of the implementation of the *TAgents System*. It describes the main parts and classes used and how they work together. After describing the parts of the System, it will explain the developed services as there is a region registry, two communication services, and two administration services. These are the basic components needed to run an Agent System.

After this follows a manual on how to build your own Agent and System Agent (service) and about the requirements to build them. Since the goal of the System was to provide the programmer with an easy-to-extend basic Agent or System Agent, it should enable the reader to develop his own part of the *TAgents Mobile Agent System*.

The last section describes the work which has to be done in the future, as there are still some things missing to provide a full featured Mobile Agent System ready for use in a real world scenario. Some of the work has already begun and some will soon be addressed.

Contents

I Introduction **1**

1 Motivation **3**

2 Mobile Agents **5**
2.1 Agents . . . 5
2.2 Mobility . . . 6
2.3 Advantages of Mobile Agents . . . 6
2.4 Java for Mobile Agents . . . 7
2.5 Related Work . . . 8
2.5.1 Java based Systems . . . 9
2.5.2 Other Systems . . . 9
2.5.3 Standard Organizations . . . 10

3 MASIF **11**
3.1 Need of a Standard . . . 11
3.2 Content . . . 11
3.2.1 MAFAgentSystem . . . 12
3.2.2 MAFFinder . . . 13
3.3 Future Work . . . 15

II The former TAgents Platform **17**

4 Design and Structure **19**
4.1 System . . . 19
4.1.1 Communication . . . 19
4.1.2 Creating a TAgent . . . 21
4.2 TAgents . . . 22
4.3 TAgentSys . . . 22

5 Limitations of the old Platform **23**
5.1 Communication . . . 23
5.2 Moving an Agent . . . 23
5.3 Standards . . . 24
5.4 Agents and System Agents . . . 24

III The new Platform **27**

6 Requirements **29**
6.1 General Targets . . . 29

7 Design and Implementation **31**
7.1 Design . . . 31
7.1.1 Resource Access . . . 31
7.1.2 Naming Conventions . . . 32
7.1.3 Helper Classes . . . 34
7.2 Packages . . . 36
7.3 System . . . 36
7.3.1 Configuration . . . 37
7.3.2 AgentSystemID . . . 37
7.3.3 Hashes . . . 38
7.3.4 MASIF Classes . . . 38
7.3.5 AgentSystem Class . . . 39
7.4 System Agents . . . 44
7.4.1 Communication Services . . . 44
7.4.2 RMI Communication Service . . . 50
7.4.3 Plain Socket (PLS) Communication Service . . . 54
7.4.4 MAFFinder Service . . . 58
7.4.5 Admin System Agents . . . 63
7.5 Agents . . . 67
7.5.1 AgentIFace . . . 67
7.5.2 Structure of an Agent . . . 69
7.5.3 Agent Class . . . 74

8 Users Guide **79**
8.1 Creating an Agent . . . 79
8.1.1 Extending the basic Agent . . . 79
8.1.2 Command Queue . . . 81
8.1.3 Resource Access . . . 81
8.1.4 Helper . . . 82
8.1.5 Example Agent . . . 83
8.2 Creating a System Agent . . . 83
8.2.1 Extending the basic SysAgent . . . 84
8.2.2 SysAgentHelper . . . 85
8.2.3 Service Proxies . . . 86

IV Conclusion and Future Work **89**

9 Future Work **91**
9.1 Security . . . 91
9.2 MASIF . . . 92
9.3 Intelligent Agents . . . 92

9.4 Improvements of the Platform 93

10 Conclusion **95**

V Appendix **97**

A Examples **99**
A.1 Example Configuration File 99
A.2 Example Agent 101

Bibliography **102**

List of Figures

4.1 Structure of the old TAgents System 20

6.1 Process Flow of the Future Result Message 30

7.1 Proxy Registration and Use 33
7.2 Proxy Design 33
7.3 System Design 35
7.4 General Structure of a Communication Services 45
7.5 Structure of AgentCommIFaces 46
7.6 Moving an Agent 48
7.7 Sending Future Messages 49
7.8 Structure of the MAFFinderServiceProxy 50
7.9 RMI Communication Service 52
7.10 Plain Socket Communication Service 55
7.11 Protocol of the Plain Socket Communication 57
7.12 Structure of the MAFFinder 59
7.13 GUIAdmin Main Window 67
7.14 Agent History for AgentEx3 68
7.15 Agent History for MAFFinderService 68
7.16 System History 68
7.17 Structure of an Agent 69
7.18 Creating an Agent 71

Part I

Introduction

Chapter 1

Motivation

The *Mobile Agent* technology is a very new part of computer science. Its origin is to be found in the Agents development which goes back a few years. Nowadays, one can already see some commercial products of this sector like Grasshopper[IKVG] and Gossip[GOSSIP].

Some universities are also developing *Mobile Agent Systems* in different languages. To mention, but a few, there are the Ara platform for Mobile Agents [ARA] and D'Agents [DAGENTS]. But these platforms are written in TCL/TK or other languages different from Java [JAVA]. In order to build a platform written in Java with open source for everyone (LGPL) [FSF], the *TAgents* (*Traveling Agents*) project was started in 1999 at the Universidad Carlos III de Madrid. The main purpose was to build an environment to support research in the sector of Mobile Agents with a good platform available to start with.

The main advantage of Mobile Agents is the smaller use of bandwidth or, more generally, telecommunication infrastructure. Since the backbone of the internet is getting improved in a very fast way the access methods for the end user are still limited to a small bandwidth (most 56kbps through analog modems). There are upcoming faster possibilities like ADSL [ADSL] or Cable Modems, but the main trend is that of wireless communication, especially through mobile phones, where there is once again a small bandwidth and no guarantee for a continuous connection. These are the main fields where Mobile Agents have a big advantage due to their concept of working locally on the server and not being dependent on a connection. Another aspect is the cost of a connection, for example in relation to the upcoming protocol UMTS [UMTS] for mobile phones, which will be through high costs for licenses and infrastructure, probably very expensive for the consumer. With Mobile Agents you just have to be connected to send your personal Agent and to call him back. This can save a lot of money when having to search and acquire a lot of information.

Chapter 2

Mobile Agents

2.1 Agents

This chapter should give a description of a special kind of Agents, the Software Agents. Since this work is just about this kind of Agent, all the use of the term Agent is referred to Software Agent.

Keeping in mind that the Agent technology is a very new part of computer science research, there is no overall accepted definition of an Agent. The most accepted and therefore very basically designed definition states, that an Agent is

> *an autonomous software entity that can interact with its environment. [OMG00, page 9]*

Autonom means in the case of an Agent that it can act on its behalf. An Agent gets a special task to process and then there is no need to interact with this Agent anymore. It can take over tasks which would have to be done by the user, but which do not require a human to act. This gives the user the possibility to transfer complete tasks to the software, thus helping in automation parts of the process which formerly required interaction of human users.

Instead of continuously interacting with the user, the Agent is capable of interacting with its environment and to change its way to accomplish its tasks depending on what it has at its disposal. It should be able to react to the changes of its environment, which is very important when considering mobility.

At the same time, it is able to collect experience in doing its tasks, in order to improve its work for the next time. So it is possible that an Agent processes the same tasks in different ways depending on its former experience and on changes in the environment. An Agent can adapt itself to a new environment if necessary.

Another aspect of interaction is the exchange of information or experience with other Agents executing in the same environment or the fact of work together on one special task. This is done by the exchange of knowledge between the Agents and the use of special functions others Agents might offer. Some tasks benefit a lot when they are addressed by more then one Agent taking the advantages of distributed processing.

The use of these possibilities of interaction with the environment and with other Agents is based on the level of intelligence the Agent has. Since no user interaction is necessary, it should nevertheless be possible for the (human) user to ask for the current status of the Agent and control the Agent's further behavior. Also an Agent should provide a log book covering all actions it took to process its task.

2.2 Mobility

Agents with the ability to move from one system to another in order to process their tasks locally on that system are called *Mobile Agents*. The traveling is done on their behalf, like acting on special requirements of their environment.

The Agents keep their current executing state wherever they travel, making it possible continue with their work on the other system. Some languages provide the facility of transferring the execution state (stack) of the Agent during execution, where others just provide the transfer of global variables.

2.3 Advantages of Mobile Agents

The advantages of *Mobile Agents* are depending on their goals or on the tasks they have to achieve. The main advantage is to work locally on a system and not to be dependent on a connection to this system. This can improve the monitoring of systems or processes, which can be done locally on a machine.

One field of use can be locally monitoring the status of a computer or a segment of a network. Instead of monitoring the segment from a special computer in a centralized way, Agents can travel to a certain segment and watch locally the status of that segment. When an error occurs, another Agent with special knowledge for this error can travel to the system and try to correct it, also, the watching Agent can alert the human administrator of the error. Since the normal monitoring process requires polling the system status every certain period of time, it uses a lot of bandwidth. Sending Agents to the target system and letting them monitor the status locally saves this bandwidth and gives at the same time the possibility to let the Agent (or another, more specialized Agent) try to correct the error on the system. Another benefit of this approach is that it will work even when there are network problems, since the Agent executes locally and is not dependent on a continuous connection. Another field of use are systems which are not easy to access by humans because of their locality. This can be, e.g. the nodes of a wide area network which are normally located in the underground and far away from their supporter.

Whenever a new version of the Agent is produced, it just travels to the target system and then replaces the old Agent there. So there is no need of locally installing any piece of software once an Agent System is installed on the machine. The same technique can also be used to distribute a software update in a large network of computers. Just by sending a *Mobile Agent* with the knowledge how to update the systems this task will be done without human interaction. The Agent should have enough intelligence to choose the best installing method on

every machine depending on the information it gets from its environment. This method is not limited to computers, it can be used for any kind of *Mobile Agents* enabled devices e.g. network components like routers, switches etc.

The second main advantage of *Mobile Agents* is the collecting and processing of information. The collecting of information is still done in a very centralized way. This means that you have a central point which collects all information and then processes it on the central machine. This is very dependent on a good connection to all of the information providers. If the connection fails, then the result is probably unusable. With the use of *Mobile Agents* you get a much more rough version to collect information. Since the Agent depends on a connection just to travel to and from the system which provides the information, it does not depend very much on a good connection. The search for the specific kind of information is done locally and also the first filtering of the information. The Agent will then only carry with it the really relevant information. So the Agent can process the information on the same machine where it has collected them and return back with just the results of this process. Almost the whole process is independent of the connection. The connection can be down some time or can change its protocols or ways through the network ,without having any effects on the Agent and the information itself.

2.4 Java for Mobile Agents

To build a *Mobile Agent System* an interpreted language has to be chosen. The language has to be interpreted, because moving of a running object requires access to the global variables, or better to the current execution pointer and stack. The best way to achieve this is using a virtual machine which executes the interpreter language. To provide this with a compiled language would require a lot of work and would be a very complicated task. The languages used by the others systems are, to mention the most important, Java, TCL/TK or special versions of C. in the case of the *TAgents Mobile Agent System,* Java was chosen as language for the following reasons:

Platform Independence The Java language is designed as a language to write programs for different operating systems. The Java source code is compiled to a special byte code which is then interpreted by the Java Virtual Machine (JVM). There are JVMs for almost every known platforms and operating systems. To mention the most important, there are Windows (all Win32 compatible versions), Linux, SunOS (or Solaris), AIX, PalmOS, and many others. An Agent System written in Java should be able to run without modifications on every one of these platforms.

Object Oriented Design Java is an object-oriented language which makes it very easy to extend the parts of the System, like the basic Agent and System Agent class. Also makes the object-oriented programming (OOP) it easier to maintain larger projects with many different parts. OOP is normally more readable for other developers. With an appropriate design, a Java program can be extended very easily.

Serialization Java has a built-in serialization mechanism. Serialization is used to store the current state of a program to, for example, send this program over the network. This provides a *Mobile Agent System* with a standard way of sending the running Agents through the network. It can also be used for storing the current state of an Agent in memory, i.g. hard disks or RAM for backups, saving or pausing them.

Networking The Java language provides the programmer with a very good package for doing all kind of networking. These classes are part of the java.net package which is included in the standard Java class libraries.

Security Java provides a very good security mechanism in its version 2. With this it is possible to provide fine grained access to all parts of an program and the standard libraries.

Reflection Java provides access to the methods and variables of an object during runtime. Therefore, the standard libraries contain the package `java.lang.reflect` which covers access to particular methods, as well as global variables of an object running in the JVM.

These are the main advantages of using Java for *Mobile Agents*. There are unfortunately also some disadvantages which are mainly a result of the platform independence of Java.

Execution Pointer The stack of a program cannot be accessed in Java. If a program is serialized, just its global variables will be serialized and not the current execution pointer and stack.

Threads Another disadvantage is the inability to control programs (Threads) in Java. The existing methods for stopping / resuming a Thread had been deprecated due to serious problems with the monitors that the thread was holding. There is no method to kill (destroy) a thread. The method mentioned in the class libraries has never been implemented (for the same reasons as above). The usage of memory and CPU time[1] cannot be controlled. This will be addressed in further Java versions.

Especially the lack of thread control leads to a lot of problems when programming a *Mobile Agent System*. This is because the Agent can be foreign code from which the System has to be protected and which should be able to be destroyed by the platform whenever, for example, security is violated.

To summarize the points above, it has to be said that Java has reasonable advantages for building a *Mobile Agent System* and was therefor elected for the *TAgents* platform.

2.5 Related Work

This section will give an overview over the existing *Mobile Agent Systems*, including most of the university research and the commercial implementations.

[1] Except the possibility of giving a thread a certain priority

2.5.1 Java based Systems

Because the *TAgents* System chooses Java as language to build it, this chapter will first describe the already existing *Mobile Agent Systems* implemented in Java.

Aglets Aglets [AGLETS] was developed at the IBM Tokyo Research Lab. It is one of the most widespread and used Mobile Agent currently available. The Aglets Workbench is the platform for running Aglets. Aglets is an event-based and very secure Agent System.

Grasshopper Grasshopper [IKVG] was developed by the German company IKV++. It was one of the first available commercial Mobile Agent platform. It is written in Java 2 and uses extensible the new security possibilities of this new Java version.

Concordia Concordia [WPW98] is a Mobile Agent developed by the Mitsubishi Electric ITA Laboratory in Waltham, Massachusetts. It mainly addresses security and reliability concerns. It uses a message queuing subsystem for reliable communication.

Sumatra A Mobile Agent developed by the University of Maryland. The main attribute of Sumatra [RASS97] is that it uses a modified JVM. The JVM was changed to make an object capable to stop and resume execution at any point in the program. With this approach it therefore solves one of the main disadvantages of the Java languages for Mobile Agents with this approach. Unfortunately it runs for that reason only on a operating system with the modified JVM available.

Voyager Voyager [GGV] is not only a Mobile Agent, but also a framework for mobile objects and distributed computing. It was developed by ObjectSpace and provides beside the support of mobile objects and agents also an ORB (Object Request Broker), and DCOM for Java. It provides messaging and communication for mobile objects. These features made Voyager a very widespread used packages, but in many cases not especially for Mobile Agents.

2.5.2 Other Systems

Telescript This Mobile Agent developed by General Magic was the first commercial written system. It uses Telescript, an interpreted language similar to Java. Much of the experience with Telescript [JEW96] were used for writing the MASIF (see next chapter) standard.

d'Agents d'Agents [DAGENTS] is the successor of Agent TCL which was developed by the Dartmouth University. Agent TCL was based on TCL/TK, but was improved to a language independent system, d'Agents. d'Agents has a language independent core with a well designed API for plugging the different languages (Java, Scheme, and C/C++) into.

Ara Ara (Agents for Remote Actions) [ARA] is, like d'Agents, a language independent Mobile Agent. It was developed at the University of Kaiserslautern, Germany. It supports TCL, C and Java. All Agents are able to travel while preserving their current state of execution (stack).

2.5.3 Standard Organizations

There are mainly two organizations involved in creating a standard for Mobile Agents.

OMG The Object Management Group, Inc. [OMG] is an international organization to promote the theory and practice of object-oriented technology in software development. It developed the Common Object Request Broker Architecture (CORBA). The standard developed by the OMG for Mobile Agents is called MASIF (Mobile Agent System Interoperability Facility). The next chapter will give a detailed description of this standard, as it is used in the *TAgents Mobile Agent System*.

FIPA The Foundation for Intelligent Physical Agents [FIPA] is a non-profit association to promote the success of emerging agent-based applications, services and equipment. It is developing a specification for agent-based applications, services and equipment. The FIPA papers include also many parts on Mobile Agents.

The MASIF standard was used for this thesis because it was specially created for Mobile Agents, while the FIPA specification was generally designed for agent-based applications. Efforts are currently started to merge the MASIF and FIPA specification. The next versions of MASIF will be designed to reach interoperability between the different standards. This will enable MASIF compatible application to be as well FIPA compatible in the future.

Chapter 3

MASIF

3.1 Need of a Standard

The number of different *Mobile Agent Systems* has been increasing during the last few years. The industry is getting more and more interested in the field of *Mobile Agents* and building its own platforms for them. *Mobile Agent Platform* are built in interpreted language like Telescript, TCL/TK or Java. So we face now the problem of different systems being written in different languages with no possibility to interact or communicate between each other.

Therefore a standard had to be defined to achieve a standardized way of providing information about Agents and Platforms, and also to provide a standard way of interaction between them. Because of the use of different languages and protocols, the method chosen to be used in the standard was CORBA (Common Object Request Broker Architecture) [CORBA]. CORBA provides a way of inter-process communication, which is independent from the language and the operating system. The communication is done by the capability of calling methods of a different program, written in a different language, and on a different computer through the standardized mechanism of CORBA. CORBA is used as the so called middle ware between the different Agent Systems.

The Mobile Agent System Interoperability Facility [MASIF98] was developed by the Object Management Group [OMG], which also developed the CORBA standard.

3.2 Content

As the MASIF standard uses CORBA, the standard itself is defined as a set of IDL (Interface Definition Language) [CORBA] rules that have to be implemented by the Agent System. IDL rules are general interface specifications to be implemented by the CORBA enabled application. The IDL rules for the MASIF, the MAF (Mobile Agent Facility) modules, are divided into two parts: The `MAFAgentSystem` and the `MAFFinder`. The following section will give a rough overview about these two modules, as they are defined in the OMG document [MASIF98]. This was the latest public available version of this standard when the diploma thesis was started and the one used to create the *TAgents*

Mobile Agent System.

The CfMAF IDL definition is not intended to describe how an Agent System handles the Agents internally or how an Agent has to be designed. It is intended to allow access to several kinds of different Agent Systems and to the MAFFinder through a known interface and the use of CORBA. This could be used to administrate different Agent Systems or to use the more indicated Agent System to process certain tasks and start there the appropriate Agent. This citation of the OMG provides the best description of the main goal of the MASIF standard:

> An important goal in mobile agent technology is interoperability between various manufacturer's agent systems. Interoperability becomes more achievable if actions such as agent transfer, class transfer, and agent management are standardized. When the source and destination agent systems are similar, standardization of these actions can result in interoperability. However, when the two agent systems are dramatically different, only minimal interoperability can be achieved.[MASIF98, page 6]

3.2.1 MAFAgentSystem

One part of the standard is the IDL module called MAFAgentSystem. It defines the basic Agent operations as create, receive, suspend and terminate an Agent. The MAFAgentSystem is mainly designed to control the Agent System from outside, like to remotely create and terminate an Agent. As the MAFAgentSystem was not implemented in this version of *TAgents*, this section will only give a rough overview about the definition of this part of the standard. The MAFAgentSystem interface consists of the following methods (For a detailed description see the MASIF document [MASIF98]):

`create_agent()` Creates a new Agents on this platform.

`fetch_classes()` Is used to get some classes from this Agent System, for example to instantiate an Agent which needs this classes.

`find_nearby_agent_system_of_profile()` Returns the location of the next Agent System which fits to the given profile.

`get_agent_status()` Returns the current status of the Agent.

`get_agent_system_info()` Returns the AgentSystemInfo object for this System.

`get_maffinder()` Returns the MAFFinder of this System or region.

`list_all_agents()` Returns a list of all Agents residing in this platform.

`list_all_agents_of_authority()` Returns a list of all Agents that have the given authority.

`list_all_places()` Returns a list of all places available in this System.

3.2.2 MAFFinder

The MAFFinder is the main registry of a region. A region is a virtually defined place where one or more Agent System are running and which have one common region registry like the MAFFinder.

The following pages describe the particular methods defined in the MAFFinder module.

`lookup_agent`

The lookup_agent is used to search for a specific Agent by giving its Name, or any Agents matching a given AgentProfile.

Parameter `in Name agent_name`
`in AgentProfile agent_profile`

Returns `Locations`

Raises `EntryNotFound`

`lookup_agent_system`

This method returns the location of the given Agent System registered with the MAFFinder. If an AgentSystemInfo is given, it return all registered systems that match the AgentSystemInfo.

Parameter `in Name agent_system_name`
`in AgentSystemInfo agent_system_info`

Returns `Locations`

Raises `EntryNotFound`

`lookup_place`

Returns the location of the given place.

Parameter `in string place_name`

Returns `Location`

Raises `EntryNotFound`

`register_agent`

Registers the given agent with the MAFFinder. An agent can be registered several times, for example when moving to another place. It (the MAFFinder) stores only the latest location.

Parameter `in Name agent_name`
`in Location agent_location`
`in AgentProfile agent_profile`

Returns `void`

Raises `NameInvalid`

`register_agent_system`

Registers the given system with the MAFFinder. The MAFFinder does not allow to register a system more than one time. If there is already an agent system registered with this name, it throws an exception.

Parameter `in Name agent_system_name`
`in Location agent_system_location`
`in AgentSystemInfo agent_system_info`

Returns `void`

Raises `NameInvalid`

`register_place`

Adds the place to the list of places registered with the MAFFinder. A place can only be registered once.

Parameter `in string place_name`
`in Location place_location`

Returns `void`

Raises `NameInvalid`

`unregister_agent`

Removes the agent from the list of agents which are registered with the MAFFinder.

Parameter `in Name agent_name`

Returns `void`

Raises `EntryNotFound`

`unregister_agent_system`

Removes the registered agent system from the list of agent systems registered with the MAFFinder.

Parameter `in Name agent_system_name`

Returns `void`

Raises `EntryNotFound`

`unregister_place`

Removes the given place from the list of places registered with the MAFFinder.

Parameter `in string place_name`

Returns `void`

Raises `EntryNotFound`

3.3 Future Work

The main part missing in the MASIF standard is an interface for Agent to Agent communication. This is important for sharing tasks, as Agents also are intended to work in groups and to share information and experience between them. To perform inter-Agent communication, CORBA should be used and the relevant interfaces for each group of Agents should be defined for your own, because the development of a common communication interface is a very complex and not so easy to process task.

Furthermore it does not cover a basic interface for an Agent, which means that with this interface implemented you could execute the same Agent in another System which is written in the same language. This could be done later when more open source Agent System implementations are available and a good basic interface is found. The actual MASIF version is intended to achieve basic interoperability and to be extended in future to more parts of *Mobile Agents* when they are more deeply known.

Part II

The former TAgents Platform

Chapter 4

Design and Structure

This chapter will describe the former TAgents System. This System was developed by Jose Luis Hernandez as his diploma thesis at the Universidad Carlos III de Madrid in 1999 / 2000.

The Figure 4.1 shows the structure of the old platform. The basic class, `TAgentSite`, is created through a RMI request from the `TAgentSiteLauncher` (or `XTAgentSiteLauncher`). The launcher application is used to control the platform over the net. The `TAgentLauncher` has to be started after creating a new site to start and maintain Agents on this site. This means that every platform has its own `TAgentSiteLauncher` to control it and to start Agents and every Agent has its own `TAgentLauncher` to control it like starting / stopping it and sending messages to it. The launchers are graphical or text mode applications.

4.1 System

The base of the former TAgents System was the class `TAgentSite`. Its methods are called over RMI by the `TAgentSiteLauncher` and the `TAgentLauncher`. It starts and maintains the Agents, provides the communication methods for them and holds information about their location.

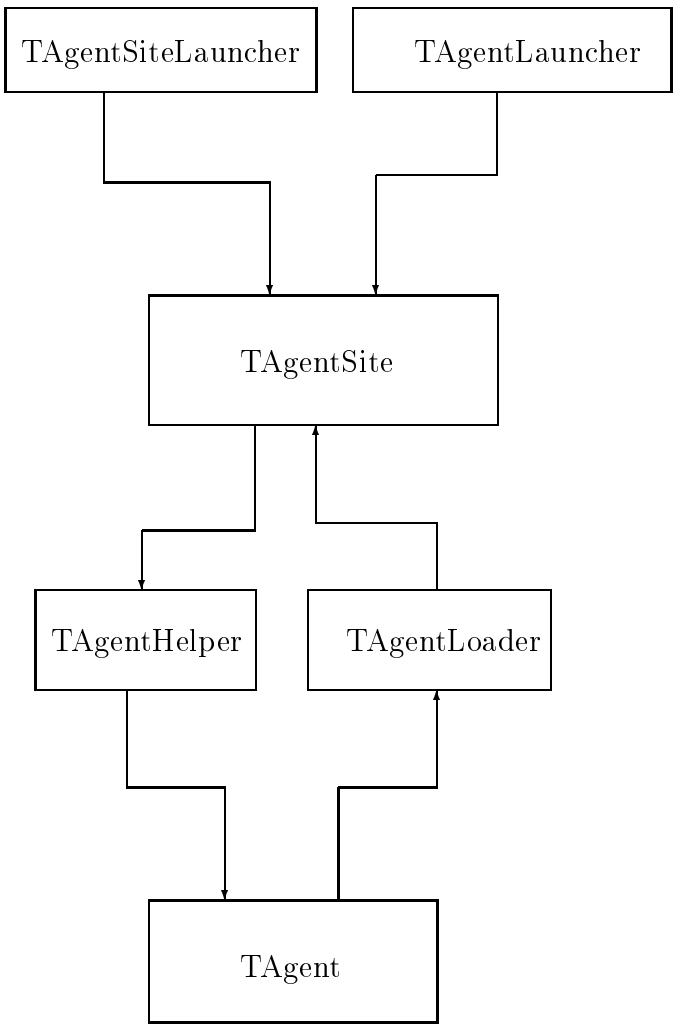
4.1.1 Communication

The communication of an Agent exists of two parts, the transfer of an Agent to another System and of sending a message to an Agent residing on another System.

The transfer of a `TAgent` is started with the `sendMe2OtherSite` method of the `TAgentSite`. This method can be accessed through the `TAgentHelper` (see next section about Agents) by an Agent or through RMI using the `TAgentLauncher`. The platform uses the following protocol for transporting an Agent:

1. The client (Agent) calls the method `sendMe2OtherSite(TAgentId tagentid, String host, String name)` which instructs the platform to

Figure 4.1: Structure of the old TAgents System



send the Agent with the specified id to the System with the specified name running on the given host.

2. The local System calls the method `request2Send` through RMI on the target System.

3. The target System answers by calling the method `ok2Receive` on the local System which instructs the local System to stop the Agent.

4. Now calls the target System the method `requestResources` which returns the URL of the classes of the Agent to be transfered.

5. The next step is the call of the method `requestAgent` on the local (sender) System. It should return the TAgent object which had to be transfered. Now the target System has the Agent.

6. The target System calls now the method `endOfReceive` to end the transmission. The sender System stores the new location of the Agent in its hash table `hashtableTravellerTAgents` to be able to access the Agent in future. At last the AgentId is removed from the hash table of running Agents on this (the local) System.

To keep track of the Agents when they are traveling, the platforms are storing the next location of an Agent when traveling in a hash table.

The transfer of a message to an Agent is done by the following protocol:

1. The client (Agent) calls the method `sendTAgentMessage(TAgentId tagentid, TAgentMessage tagentmessage)` on the platform, through its helper.

2. If the Agent resides in this platform, the System looks for the TAgentLoader and calls the method `handleTAgentMessage` on it.

3. If the Agent does not (or no longer) reside in this platform, its next location should be stored in the `hashtableTravellerTAgents`. If it is found there, the System just calls the same method (`sendTAgentMessage`) with the same parameters on the next platform through RMI. If it is not found, it returns a `TAgentResponse` with an error code.

The sending of a message to an Agent is only possible when the sender Agent resides on a System that has created the receiver Agent, or it has been there at least one time while traveling.

4.1.2 Creating a TAgent

The creation of an Agent is done of a call to the method `loadTAgent` from the `TAgentLauncher`. Then the platform creates a new id for the Agent and calls the `RMIClassLoader` for loading the required class from the URL given by the requester. The new Agent is then instantiated and a new `TAgentLoader` for this Agent is created and advised to start the Agent. At last the loader is put with the `TAgentId` into the `hashtableTAgentLoaders` to store it.

4.2 TAgents

The basic class of an Agent of this TAgents platform is called `TAgent`. It contains the following methods:

init() Called one time in the life of the TAgent after it had been created.

start() Called every time when the TAgent is started, first after being created and every time when traveled to another TAgentSite.

stop() Is called when the TAgents has to stop executing.

conclude() Should be called when terminating the TAgent.

dispatcher() Is called to invoke the method given in the `nextState` variable. Should be called in the end of the `start`() method.

setTAgentHelper(TAgentHelper tagenthelper) Is called by the System to give the TAgent its helper.

4.3 TAgentSys

This section will describe the class `TAgentSys` which could be used to build system agents. The purpose of the `TAgentSys` was to provide a special kind of stationary agents that are providing resources for the agents. It has therefor the following methods:

invoke This method invokes a method given by the `method_name` parameter with the given parameters (`method_param`).

getMethods Returns a hashtable with all methods starting with the special word "handle" which should be public accessible. The methods to access the resources of the system agents are distinguished from the normal methods by starting with this word.

A `TAgent` was thought to access to the resources as with sending message. It would have to give the name of the method and the parameters and would get in return the result of the method invocation with the `TAgentSys`.

Chapter 5

Limitations of the old Platform

This chapter will describe the limitations of the former Agent System and why it was decided to rewrite the whole System using a new design. The need of a different design came from the new requirements for this diploma thesis as described in the next chapter.

5.1 Communication

The communication protocol was hard coded into the former platform. The used protocol was RMI and the whole platform was build as a RMI server and client. This would have made it very hard to implement a new communication method, like for example plain socket communication. This made the former platform very dependent to the RMI and under some circumstances it could be necessary to use another protocol for communication. For example there are a few Java implementation which do not provide a RMI service like in the palm pilot. Especially these small devices like PDAs or handhelds are one of the targets for *Mobile Agent Systems*. Also it could be that in some environments you have to use another protocol, because of for example a strict firewall. One of the most important reasons is the use of the CORBA protocol in the MASIF standard. When trying to use this standard one has to implement the CORBA protocol. This are the main reasons why most of the major *Mobile Agent Systems* are supporting more protocols like plain sockets or CORBA.

The next problem was that there was missing an asynchronous messaging method. Asynchronous messages are very important for tasks which can last much time. The messaging routines were not defined by keeping in mind to make them easy to change to be asynchrony[1].

5.2 Moving an Agent

The protocol used to move an Agent is very complex. The handshake between the two systems exists of six method calls. For most of the methods it is not

[1] Messaging had some implementation liimitations, like there was not returned the real result of the method invocation on the target Agent.

necessary to put them into one single call. This makes it all more dependent on a good connection which should not be required for a *Mobile Agent System*. Such a System is specially designed for minimizing the use of communication and the dependence of a good connection. The loading of classes for example can be done by a designated ClassLoader which would be the way it should be implemented in Java. The use of the RMI ClassLoader has the advantage to be able to load classes from FTP / HTTP server, but it is better to use a class loader which is derived from the normal ClassLoader of Java. It is not difficult to implement the loading of classes from a FTP / HTTP server using the good network classes of Java. It can also be that the classes are available in the System for example they are in the CLASSPATH or cached. The last call, `endOfReceive`, is obsolete when using exceptions to point an error during transfer or starting[2].

5.3 Standards

There would have been some difficulties when trying to adapt the System to the MASIF standard. First, most of the methods used in the MAFAgentSystem are not available in the platform or for a System Agent. The protocol and structure was not created to be some day conform to this standard.

The MAFFinder would have to be an extra program due to the lack of a good extension mechanism. The main disadvantages of making the MAFFinder a single program are described in 7.4.4 on page 58. The methods to track an Agent are complex and can require a lot of calls over the net when the Agent traveled a lot.

5.4 Agents and System Agents

The main problem of the Agents is that they are not really separated in their execution from the platform. This can only be achieved by the use of own threads for all calls to methods from the Agent. The usage of the stop and conclude[3] methods is not defined and it cannot stop an Agent. Stopping and terminating an Agent is not a trivial tasks in Java due to the disadvantages in controlling threads, but it can be made as secure as possible by the use of threads and an improved AgentLoader.

The mechanism to offer and access resources goes through calls to method signed by a special word at the beginning of the method name. Java provides through the use of interfaces a far better possibility to just make a group of methods available for another object. This is the main field of use for interfaces in programs. The Agent has to know the exact syntax of the methods anyway to be able to call them. Also is the direct way (or through proxies) faster then the invoking of a method using the reflection API of Java. At the end the use

[2] The sender platform would not be able to recognize that an error occurred when starting the Agent on the other side, because the end of the handshake was sent already before starting the TAgent on the target System.

[3] The conclude method of a TAgent is never called in the whole System anyway.

of interface provides a faster and cleaner way to offer and access resources. The next thing which is missing is the infrastructure to offer and to look for the resources offered by the System Agents. Neither the platform nor the helpers have methods for accessing / providing the resources offered by System Agents. There exist only the two methods in the TAgentSys class which are designed for handling resources.

As one can see there were several things missing in the old platform and several thing would have to be rewritten to achieve the requirements for this diploma thesis. The basic old TAgents System is lacking a appropriate design for the new requirements for Agents as well as for System Agents and communication. It would have been at least the same work to correct these things as to design and implement a new System, which would have a good basic design at the end.

Part III

The new Platform

Chapter 6

Requirements

6.1 General Targets

The basic requirements for this work were to extend the old *TAgents* platform with plain socket communication, asynchronous messaging for RMI and plain socket and to build a MAFFinder. Due to the reasons mentioned in Chapter 5 it was very fast clear that it would mean major rewrites. So it was decided to rewrite whole the platform implementing a good design and making it ready for future enhancements.

This leaded to some more requirements for this work. Now the whole platform had to be redesigned and rewritten in addition to the former requirements. The new requirements to met were the following.

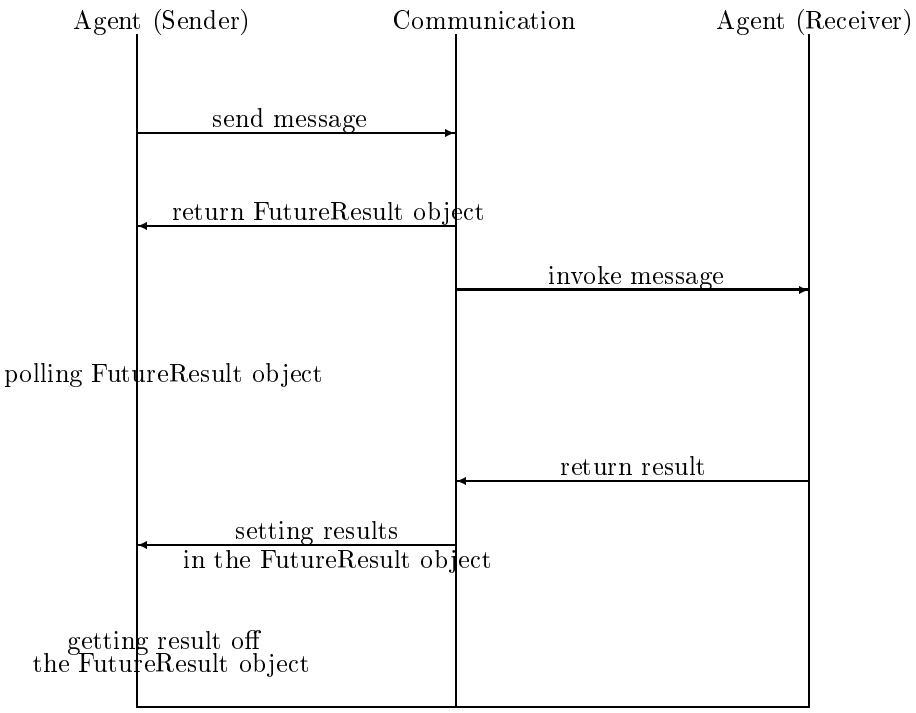
First the new design should be easy extendable by more communication protocols like CORBA or secure sockets layer. Since there exist already two protocols in this work the new design would have to be independent from the communication protocol anyway. This can only be achieved by separating all the communication methods in an extra module and provide the System with a full interface to all communication needed.

Second requirement was to implement asynchronous communication. Asynchronous communication means to send a message and to be able then to continue execution without having to wait for an answer. In other words the object which sends a message is not blocked until the answer arrives, which means until the message was processed and the response was transported back to the sender. There exist two main approaches for this communication method.

The first is event based. An agent sends a message and continues executing. When the response is arrived it gets an event that contains the response. This is used for example by the IBM Aglets. It fits good in the more event based design of Aglets.

The other approach uses a so called FutureResult object to store the response in. In detail sends the Agent a message and gets immediately a FutureResult object back. This object will contain the response after the message was processed and the answer arrived in the senders Agent System. The Agent polls the FutureResult object until the response is there. This method does not block as well. The following image will give a an overview about the calls made when

Figure 6.1: Process Flow of the Future Result Message



sending a FutureResult message.

The third part of the general requirements was to implement a MAFFinder. Since the use of CORBA requires significant knowledge in this communication method and just runs on JDKs newer then 1.2.2 it was decided to just implement the MAFFinder without the use of CORBA. It means that there had to be used the interface definition of the MAFFinder but another, standard communication method.

There exists also a MASIF standard for an *Mobile Agent System* which is called MAFAgentSystem. It was decided not to intend to make the System compliant to this standard because of the status of the standard and the difficulties to implement. The difficulties were mostly missing methods and not so easy to implement method calls. It has shown that in newer versions of the standard the MAFFinder interface has been kept almost the same since major changes have been in the MAFAgentSystem. So this decision was proved to has been correct this time.

Chapter 7

Design and Implementation

7.1 Design

7.1.1 Resource Access

The new design of the TAgents platform tries to fulfill all the requirements of a highly extendable system. In the concept of Mobile Agents are stationary Agents used for providing special functionality to the other Agents. By extending this concept one can use so called System Agents to extend the System with all kind of resources as there are at this moment communication, a region registry and administration tools. These resources have to be public available (with security checks) to the System and to the Agents residing in it. The administration and distributing of the resources is done by the platform where all resources have to be registered.

The first design for this work used interfaces with the definition of access to the provided resources to be published in the System. These interfaces where implemented by the System Agents and then registered in the platform. The platform hold a string describing the interface as the key to the resource. This string was just the name (with package prefix) of the interface registered in the platform. Unfortunately it was now possible for an Agent to do a recast of the interface reference to the real System Agent. The platform was managing the resources in a hashtable with the name of the interface as key and a reference to the interface implementation of the System Agent. To give an example the (value:key) entry in the hashtable was something like

```
interfacename :  reference_to_interfaceimplementation
“tagents.mafffinder.MAFFinderIFace” :  (MAFFinderIFace)
maffinderservice
```

With the interface implemented by the System Agent. An Agent could now do a recast to the System Agent:

```
SysAgent  sysagent = (SysAgent) reference_to_interfaceimplementation;
```

Now the Agent had access to all the standard methods of a System Agent like e.g. `suspendAgent` or `terminateAgent` to mention the worst. So it could do something like:

```
sysagent.terminateAgent();
```

Which would terminate the MAFFinder of this System. Since the recast is not restricted to the basic System Agent class but to all other interfaces and methods implemented by this special System Agent it would have been a too big overhead to check the security at every call to a method of a System Agent. So the design of the resource access had to be changed to provide the Agent only with an object containing no more than the methods defined in the interface.

This object which implements only the methods of the interface registered in the platform works as a wrapper or proxy between the user (Agent) and the provider (System Agent). It will be referred as proxy. The System Agent still implements all the methods needed for accessing the provided resources but creates then a proxy to register the proxy instead of itself to the platform. The proxy implements the same interface as the System Agent and contains internally a reference to the service providing System Agent (also called service). All calls to methods of the proxy are wrapped to the calls of the System Agent. Some methods now need to know which Agent has called them, making it necessary to personalize the proxy given to the Agent. First the platform had just one reference to the proxy and provided this reference to all Agents which where allowed to use one or more methods of this interface. The Agent had now access to all methods of the interface without giving the service the possibility to check if the Agent is allowed to call this method. For example it was possible for an Agent to call the `unregisterAgentSystem` method of the MAFFinder which is used to remove the entry for the given Agent System out off the MAFFinder.

So it was clear that the proxy must know which Agent is using it. This implies that every Agent using a proxy must have its own personalized clone of the proxy. Every proxy now has to extend the `ProxyIFace` which provides a method to clone the proxy with providing the ID of the Agent. This clone is then provided by the platform to the Agent requesting the resource. To do not let the Agent create its own clone of the proxy with another ID the proxy can only be cloned once from the proxy provided by the System Agent. See image 7.1 on page 33 for a process diagram.

For an overview of the relations between the proxies, the System Agent, the Platform and the Agent see Figure 7.2 on page 33.

As one can see go all the calls to the System Agent through the proxy and not through the platform. The platform does not know which kind of resources are accessed by the Agent which makes the platform fully independent of the resource and service providing System Agents. The platform does not have to be changed when adding a new service (System Agent) like a new communication protocol or other resource access providers.

7.1.2 Naming Conventions

To provide a good key for resources there was developed a hierarchy of interfaces for the services (proxies). First some naming conventions. For easy identifying an interface it should end with the word "`IFace`" like the MAFFinder interface is called "`MAFFinderIFace`". Some resources belong basically to the same group. For example the communication methods offered to an Agent are basically the same independent from the protocol used. This basic collection

Figure 7.1: Proxy Registration and Use

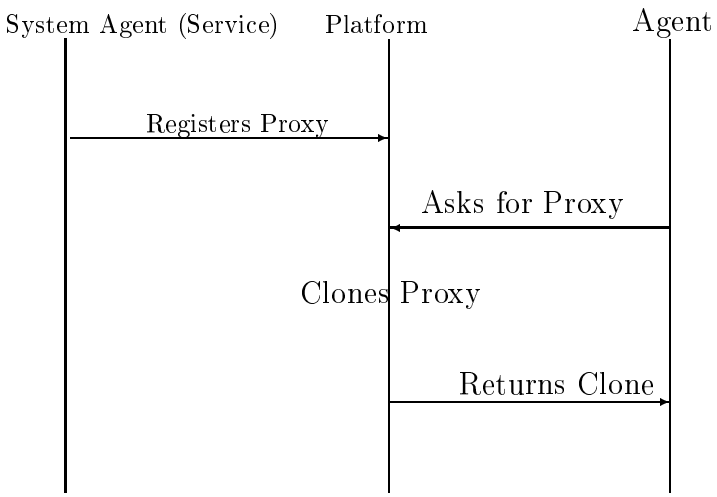
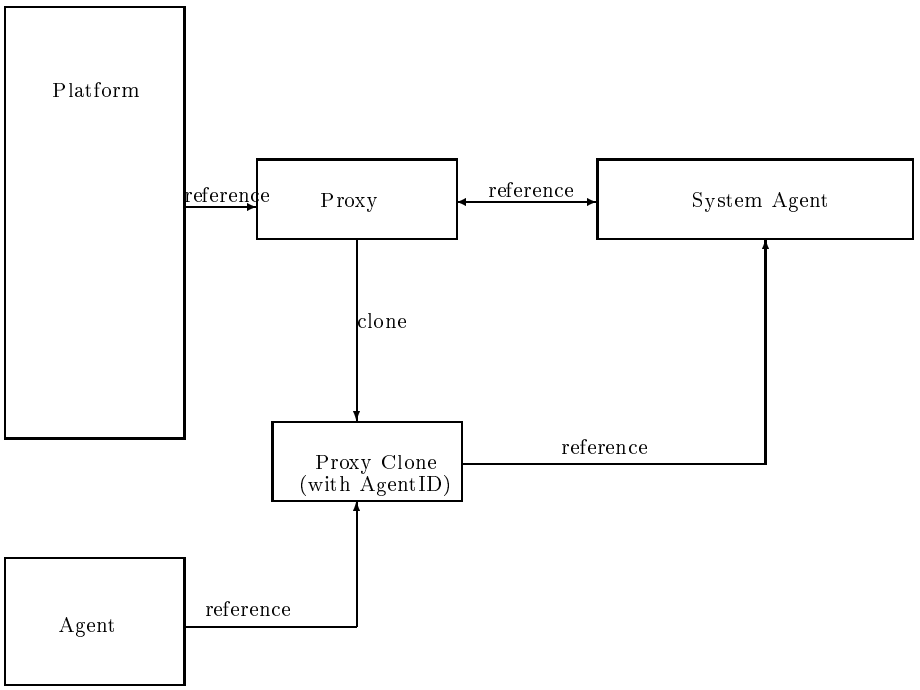


Figure 7.2: Proxy Design



of methods is called "`AgentCommIFace`". To give an Agent the possibility to choose a special protocol, each communication service provides a derived interface with the common abbreviation of the protocol at the beginning. For the RMI communication service means this that the published interface has to be called "`RMIAgentCommIFace`". The same mechanism is used for the plain socket protocol and the interface is called therefor "`PLSAgentCommIFace`".

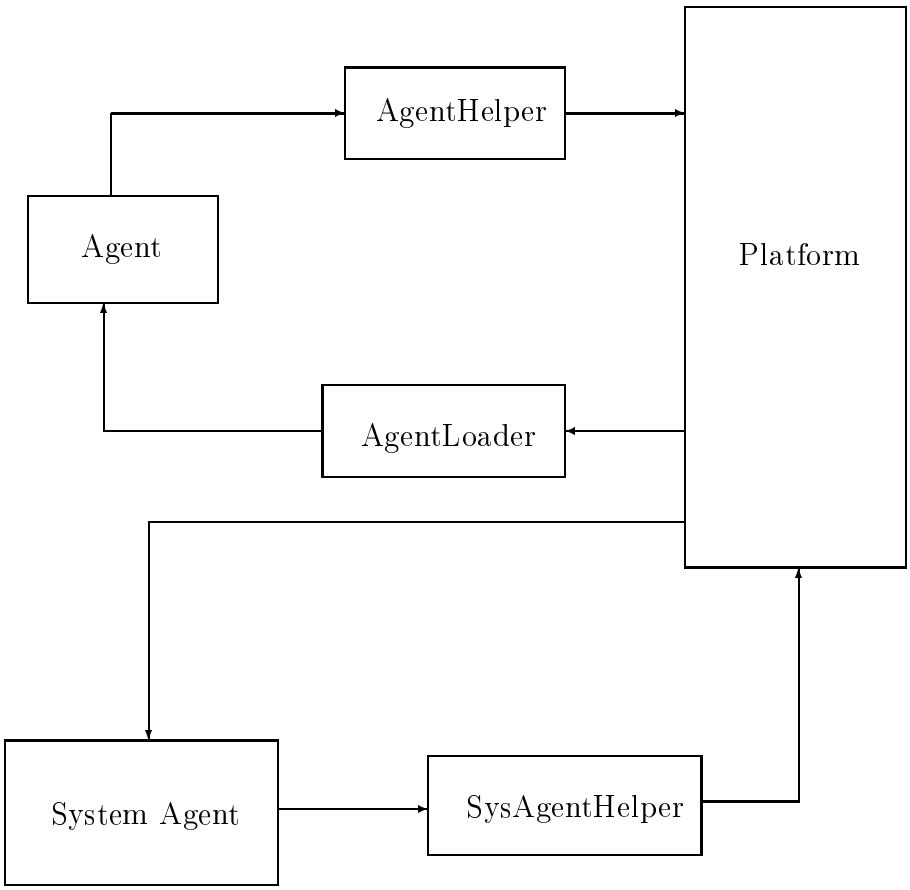
The same rules are applying for the packages where the communication services are belonging to. The main package is "tagents.sysagent.communication" and every protocol has its own sub-package in the form of the abbreviation in lower case letters. The RMI communication service resides in the package "`tagents.sysagent.communication.rmi`", is called "`RMICommService`" and implements the interface "`RMIAgentCommIFace`". These rules are used in the Agent class to search for a suitable communication form between two Agent Systems. Any protocol implementation which is not conform with these conventions will be ignored by the search algorithm.

Another basic interface to mention is the "`tagents.sysagent.admin.AdminIFace`" which is used for being implemented by admin System Agents. In opposite of the "`AgentCommIFace`" this interface is used as it without deriving it.

7.1.3 Helper Classes

To do not let the Agents directly access the platform itself all the calls are wrapped through the `AgentHelper`. The `AgentHelper` provides all necessary methods for the Agent to access resources, do its output and get some general information about the System where it resides. The Figure 7.3 on page 35 shows the rules of direct access in the System. The Agent accesses the platform through the `AgentHelper` as well as the System Agent it does through the `SysAgentHelper`. All of the calls are wrapped to the System with the AgentID as parameter to let the System check the right to call this method. The platform accesses the Agent through the `AgentLoader` which provides all methods to control the Agent. The System Agents are accessed directly by the platform because they are executing as part of the System and not in special threads.

Figure 7.3: System Design



7.2 Packages

This section should provide an overview of the packages build in the TAgents project. To keep the project at a good level of administration the Java packaging possibility is highly used in all parts of the project.

tagents The basic package is called "tagents" and contains only sub-packages and no classes. This is used to identify that a sub-packages that belongs to the TAgents platform.

- *system* The system sub-package contains all the files used by the platform and which are necessary for running it.
  - *exceptions* This package contains all exceptions that can be thrown by the System. Only the exceptions for the MAFFinder are residing in the maffinder package.
- *agent* The agent sub-packages contains all files belonging to an Agent.
- *sysagent* This package contains the basic SysAgent class and all the sub-packages for the different types of System Agent.
  - *admin* This package contains the basic interfaces and proxies for the Admin System Agents and sub-packages for the two existing kinds of Admin System Agents, the "ConsoleAdmin" and the "GUIAdmin".
  - *communication* The communication package contains the different protocols for communication. Every sub-package is an implementation of one communication protocol as there exist a package for RMI ("rmi") and for plain socket ("pls").
  - *maffinder* This contains the region registry MAFFinder as described in the standard. It does not contain any communication specific parts since this belongs to the protocol implementations.
- *examples* This packages contains example Agents which are used to test the functionality and to show how the System works.

7.3 System

This section should give a detailed description of the content of the package system especially of the main class `AgentSystem`. This class is also referred in the documentation as the platform. The `AgentSystem` mainly maintains all parts of the System as there are for example the Agents residing in it, the System Agents and the proxies.

7.3.1 Configuration

Once a new instance of the `AgentSystem` is created it reads it configuration from a text file which has to be given as parameter when starting the System. The configuration file is read and parsed by the class `Configuration`. It can contain comments which are identified by a leading "#" at the beginning of the line and empty lines which contain only tabulators or spaces. All other lines are interpreted as a section label or a key / value pair. The configuration is divided into several sections. Every section has a label which has the form `[name of section]`. The first section is labeled `[SYSTEM]`. This section contains all parameter for the platform means for the class `AgentSystem`. At the time of writing there are the following parameters known to the System:

- `NAME` sets the name of the Agent System. This name should be unique in the computer where the System is running on.
- `WAITFORSHUTDOWN` sets the time the System is waiting for all Agents to die. As parameter you have to give the time in 100ms steps. Like the parameter 10 means the Systems waits 1000ms or 1 second for all Agents to terminate. See 7.3.5 on page 42.

The second section labeled `[STARTUP]` contains all things that have to be done on starting the Agent System. In this version of *TAgents* it contains only a list of System Agents that have to be loaded when starting the platform. The syntax is `LOAD=name_of_SysAgent` where the name is the full name (containing all package names) of the derived System Agent (without the .class extension). One should load at least an admin System Agent when starting the platform to be able to administrate it, e.g. to start other System Agents or Agents. This is the only option now which has the same key for one or more values.

The other sections contain parameters for some System Agents. They are labeled as `[full_name_of_SystemAgent]`. For example the section containing parameters for the `RMICommService` would be named `[tagents.sysagent.communication.RMICommService]`. For details on the parameters of the services see the sections describing them.

For an example see appendix A.1 on page 99.

7.3.2 AgentSystemID

Every System has to be able to be addressed with an unique ID. To provide a net wide unique identifier *TAgents* uses a constellation of the ip number of the machine where the System is running and a name which is unique on this machine. These two identifiers combined produce a good and unique ID for the System. Normally the ID is represented as a string in the form of `ip-number + "/" + name`. It is the responsibility of the administrator to choose different names for more Agent Systems running on the same machine. In fact the System terminates with an error when already another Agent System is registered with the same ID in the MAFFinder (region registry).

The `AgentSystemID` as well as the `AgentID` are not the same as the Name object used by the MASIF standard. But they provide methods to convert them

to each other in both directions. The basic ID as described above is usable as key for the ID objects as well as for the Name object. The main differences are that the ID objects (`AgentID` and `AgentSystemID`) both provide a `Authority` entity. It could be decided in the future to just use the `Name` object especially when making the Agent System complaint to the `MAFAgentSystem` standard. You would then have to provide the Authority as an extra parameter when necessary.

7.3.3 Hashes

The platform uses hash tables for many purposes. These are in particular hashes for the Agents, the System Agents, the service proxies and the configuration. Since some of them do not fit in the normal unique key - to - value scheme, but can contain more then one value per key it was necessary to create special version of hash tables which can handle more values per key. These are called OTM*Hashtable using the acronym OTM for one - to - many hash tables. There exists two kinds of `OTMHashtable`, the normal `OTMHashtable` which can handle strings as keys and the more general `OTMObjectHashtable` which can handle any objects as you like as key and value. Since these special versions of hash tables are not as optimized for speed as the default `Hashtable` of the Java class library it is strongly recommended to use them only if it is really necessary.

The Agents residing in the System are archived in the Hashtable agents. As key the `AgentID` is used and the value is the reference to the Agent. To provide good hash codes for the ID objects and other *TAgents* proprietary objects there exists a class called `HashCode` which produces unique hash codes for a string and for a byte array. Since the IDs are normally presented as strings (in the ID objects) or as byte code (in the `Name` object) these are the two possible cases where hash codes are needed. Internally the hash code is generated by using the hash code generating method of the standard class `String` of the Java class libraries. This is achieved in the class `HashCode` by converting the byte array into an `String` and return its hash code value.

The System Agents are maintained as the Agents in a `Hashtable` with the `AgentID` as key and the reference to the System Agent as value. Separate from them exists a `OTMHashtable` for the services (proxies) they are providing. The proxies hash table uses the string of the name of the interface as key and as value the reference to the proxy. Since more then one System Agent can provide the same interface, like for example the general
`tagents.sysagent.communication.AgentCommIFace`, an `OTMHashtable` is used here to store the proxies.

Places are handled different since they do not fit into the key - value scheme but only have one value. They are stored into a `Vector`. Since places are still not supported in this version of *TAgents*, this can be changed in the future.

7.3.4 MASIF Classes

To make the platform compatible to the MAFFinder part of the standard and to prepare it to be some day compatible to the `MAFAgentSystem` as well there

were some special classes needed to be implemented.

In detail are these the `AgentSystemInfo` which is used in the MAFFinder to hold properties and a description of the registered Agent System. The class `Name` contains the ID of a general part of the System which can be the ID of an Agent or System Agent or the ID of the Agent System. Also to be mentioned here is the class `AgentProfile` which is in the agent package and contains like the `AgentSystemInfo` general properties of the Agent.

Some of the classes are using special constants for properties which are still not defined by the OMG. To provide a central class for these constants (like the used programming language or serialization mechanism) there exists the static class `MASIFConstants`. This class only contains the constants, still in a *TAgents* proprietary form, but easy to be adapted to a convention of the OMG.

The class `Location` is like an URI [URI98, Uniform Resource Identifiers] and can hold the location of an Agent System, a System Agent, an Agent or a place. The location of a component of an Agent System can be represented by a URL [URL94, Uniform Resource Locators] in the following form:

`[protocol:]//host[:port]/systemname[/place][/agentid]`

With the parameters in brackets [] are optional. When giving the URL to an Agent you have to give a placename too, otherwise it would not be easy to distinguish between a place and an Agent. This is important since the `Location` internally uses entries for places, Agents, System names, hosts, ports and protocols. The URL itself is every time generated off these values. The `Location` class is also capable of parsing a given URL to a `Location` object providing both ways of accessing. You can generate a `Location` with all the parameters (which can be `null` if they are is not used) or of an ID of a System or Agent since these contain the ip number as well as the name of the System.

7.3.5 AgentSystem Class

The `AgentSystem` is the main class of the System. It maintains the Agents, System Agents, places and proxies.

Initializing

When a new instance of it is created it calls the method `initSystem` to do initialization work. First it gets its parameter from the configuration and then it loads the System Agents that are set to be loaded. This is the part which is defined in the configuration file. If a MAFFinderService or MAFFinderProxy was loaded it uses its registered proxy to register itself at the MAFFinder. When there was already a Agent System registered with the same ID (same name at the same machine) it exits with an error.

Now the System has initialized and is ready to listen to requests from Agents or System Agents. To keep the virtual machine running there has to be a thread executing. Since the `AgentSystem` is the main part of the System it had to extended the `Thread` class and to provide a `run` method which stops only at the termination of the whole System. Therefor the `run` method of the `AgentSystem` has a internal loop which sleeps 200ms to not abuse too much processor time.

The loop exits when the boolean variable `running` is set to false which is done when stopping the System.

Resource maintaining

One of the most important parts are the methods to let a service proxy register its interface/-s in the System and of course to let them be unregistered. When a service (System Agent) registers a proxy in the platform, it calls the method `registerService(Object proxy)` which itself calls the method `getInterfaces(Class cl)` to get all the interfaces that are implemented by the proxy. Only the `ProxyIFace`, the `Serializable` and the `Remote` interfaces are ignored since they are basic interfaces. All others are stored in the `proxies OTMHashtable` with the names of the interfaces (if the proxy had more then one interface implemented) as key and the proxy as value. For example the `RMICommProxy` implements the interface `RMIAgentCommIFace` which is a sub interface of the `AgentCommIFace`. So the `RMIAgentCommProxy` implements two interfaces and will be registered two times with each interface name as key. The unregistering is much more easy because the `OTMHashtable` provides a method to remove all entries with a given value which is in this case the proxy object.

When ever an Agent or System Agent requires (through their helpers) a proxy it calls the `getProxy(String proxyinterface, AgentID agentid)` method with its ID for security checks. It gets then all proxies which are providing the given interface back. For examples a request for the interface "`tagents.sysagent.communication.AgentCommIFace`" gives you all the communication proxies back. That could be the `PLSAgentCommProxy` and the `RMIAgentCommProxy`.

The Output Framework

The next set of methods are used to handle the output of all parts of the Agent System. At default the `AgentSystem` just prints all on the STDOUT stream (`java.lang.System.out`), but if a (or more) admin System Agent has registered the `AdminIFace`, which provides all these methods, the System sends the output to the admin SysAgents. The main intent of this output framework is for an Admin System Agent which can now provide a history for each part of the System. The GUIAdmin (see Section 7.4.5 on page 63) is a good example to show the use of this concept. In detail the output framework consists of these methods:

`print(String s)` Is used for the output of the System (class `AgentSystem`).

`println(String s)` Is also used for the output of the System but inserts a newline at the end of the string.

`print(Exception e)` Is used to print the checked exception. This could be used to open an error window with the the exception description.

`print(AgentID id, String s)` Is used for the output of all Agents (including System Agents).

`println(AgentID id, String s)` The same as above but with newline on the end.

`print(AgentID id, Exception e)` Is used to print a catched exception of the Agent.

Creation of a System Agent

Some of the most important methods are the functions to maintain Agents and System Agents. Although the System Agents and the normal Agents have a common base they are created very differently. A System Agent has to be loaded from the CLASSPATH, it cannot be loaded over the net. One can think about letting them be loaded from another, trusted Agent System. A System Agent is loaded from the class given by the (`String`) parameter name of the method. The System creates first a new instance of the class and then a new `AgentID` for the System Agent. The System Agent should still not really execute although its main constructor has been called already. The next step is to create a `SysAgentHelper` and then it can be initialized and started (by calling `initAgent()` and `startAgent()`). Now the System Agent should be ready to do its work. It is now put into the `sysagents Hashtable`. At last the System tells every Admin System Agent that there has been created a new System Agent to keep their lists of System Agents up to date.

For details which is done at initializing and starting of a System Agent, see Section 7.4.

Creation of an Agent

The Agents are created different. One difference is that an Agent can, when the `AgentClassLoader` supports it, be loaded over the net from a ftp or web server. Remember that this is still not implemented in this version of *TAgents*. This is also valid for the third possibility to create an Agent, to fetch the classes from another Agent System. So we will only have a look at the creation of an Agent with its class files in the current CLASSPATH. The parameters are the name of the class (like with a System Agent) and an Authority for the Agent.

The first step is to create a new `AgentID` for the Agent arising. This step has to be done first to check the security for this Agent before instantiating it. So the next call goes to the `AgentSecurityManager` to check if the Agent is allowed to execute here (`checkExecution()`). Now the Agent can be created. The biggest part is taken over by the `AgentLoader` which is called now to instantiate the new Agent. It calls the `AgentClassLoader` to get a new Agent class back to make it easy to be used for Agents from other sources. Then the Agent gets its AgentHelper and the AgentLoader is instructed to initialize and start the Agent. The the `AgentLoader` is put with its ID into the `agents Hashtable`. At last the System tells every admin System Agent that there has been created a new Agent to keep their lists of Agents up to date.

For details which is done at initializing and starting of a System Agent, see the Section 7.5.

Controlling of (System) Agents

Like mentioned before the Agent and Sysagents are in their controlling very similar. Both have the same methods to suspend / resume / terminate them. This is a result of the basic AgentIFace. The biggest difference is that Agents are controlled through their AgentLoader which do not have System Agents. To keep the maintaining of both as simple as possible the following methods can handle Agent as well as System Agents. Both are referred as "Agent" in this section (and only in this section).

`suspendAgent` Stops the Agent by calling its suspend method.

`resumeAgent` (Re-)Starts the Agent which has do be stopped before. If an Agent is created its `startAgent` is called directly.

`terminateAgent` Terminates a stopped Agent. If the (real) Agent was not stopped before, the Agent will be stopped by its AgentLoader before terminating it.

For a detailed description of what is done when suspending / resuming / terminating an Agent see the corresponding description of the Agent or the System Agent later in this document.

Shutdown

To shut down the System you have to call the method `stopSystem()` which does nothing else then setting the `running` variable to false. When this boolean is false, the loop in the `run()` will terminate and the `terminateSystem()` method is being called. The terminateSystem method cleans up the System and terminates the program. This is done in the following order:

1. Unregistering the System from the MAFFinder.

2. Calling the `terminateAgent()` on every Agent residing in the System. This is done by calling the corresponding method on the `AgentLoader`. The `AgentLoader` creates a new `Thread` to conclude the Agent asynchronously. This is important because it can be that after calling this method an Agent still executes. But anyway the `agents Hashtable` will be cleaned after that.

3. Calling the `terminateAgent()` on every System Agent residing in the System. This is done synchronously. After that no System Agent is executing anymore and there should be no more a proxy registered in the System. The `sysagents Hashtable` is being cleaned.

4. To give all Agents enough time to terminate themselves in a clean way the System waits the in `WAITFORSHUTDOWN` given 100ms if there are still other threads running in the thread group of the AgentLoaders where the Agents do have their threads.

5. When all Agents have stopped their threads in time and so the System is in a clean state to shut down, it calls the `System.exit(0)` to shut down the JVM (Java Virtual Machine). When not all Agents where terminated it prints out a list of all running threads and exits then with the error code 1.

Methods of the PlatformCommunicationIFace

The communication services need some methods from the System to transport Agents and transfer messages. Since these are special to the behavior of the communication services, please consult the Section 7.4.1 on page 44 for a more detailed description of the methods.

When an Agent requests to be sent to another Agent System it calls the `move` method of the communication service. To get the Agent in a safe state the communication service calls the `prepareForMove` of the platform. The platform then stops the Agent and returns a reference to it. When the transfer was OK it calls the method `transferOK` of the System which terminates the Agent because it now executes in the other Agent System.

The last method, `receiveAgent`, is called when a communication service gets a request to receive and restart an Agent from outside. This method checks first if the Agent is allowed to execute here and then creates it a new `AgentLoader` and `AgentHelper`. If all went OK, it is restarted and put into the `agents Hashtable`.

General Methods for (System) Agents

System Agents need access to some parts of the System to do their work. Remember that all calls to the System are wrapped through the `SysAgentHelper`.

`getLoader` Return the `AgentLoader` (and with this the Agent itself too) of the Agent with the given `AgentID`. This method is called directly by the admin System Agent on the platform without going through the helper. The admin is the only System Agent which uses this method, so it was not necessary to put it into the helper.

`getSysAgent` Returns the System Agent of the given `AgentID`.

`getAgentStatus` Returns an integer representing the current status of the Agent of the given `AgentID`. The status codes are defined in the `AgentLoader` class.

`getSystemName` Returns the name (not the ID) of the System.

`getHost` Returns the ip address of the machine where the System is executing. It is very probably that the access to the `java.net` package is restricted by the security in future so this value is not accessible through standard library calls.

`getSystemID` Returns the `AgentID` of the System.

`getAgentSystemInfo` Returns the `AgentSystemInfo` object of the System.

`getSystem` Returns a reference to System itself. This is only for use by admin System Agents and all other request will result in a security exception.

The last method to mention here is the `main` method of this class since this is the main class of the whole Agent System it has to be started by the command line. Because the System is dependent of a given configuration file the `main` exits with an error when there is no parameter (the config file) given at the command line. After checking this, it instantiates a new `AgentSystem` and starts its thread by calling the `start()` method (which is inherited by the extended `Thread` class). Then the AgentSystem is started and is proceeding with its work.

7.4 System Agents

The base of every System Agent is the class `SysAgent`. It implements the `AgentIFace` to provide a common set of methods for both Agents and System Agents. The System Agent uses as the Agent a helper for accessing the System routines. This helper, the `SysAgentHelper`, is set when instantiating a System Agent.

The `SysAgentHelper` provides methods for register, unregister and to get a proxy. It also provides the methods needed by the communication services as they are mentioned already in the description of the `AgentSystem` class. To repeat it, these are `prepareForMove`, `transferOK`, `fetchClasses` and `receiveAgent`. A detailed description follows in the next subsection. To be mentioned also are the print methods for the output framework and the rest of the methods that are used to get information about the System.

There exist already three different kinds of System Agents. The communication services handle all communication and are implementing each one a different protocol. The MAFFinder service provides the System with a standard conform region registry. The admin System Agent provide the System with a user interface to let the user control the System, like to start Agents.

7.4.1 Communication Services

The communication service was designed to handle all communication which could be in the *TAgents Mobile Agent System*. The two parts who could need to communicate over the net are the Agents and the System Agents. The platform itself does not need communication at all. The System Agents who actually need communication are the communication services, they are using their own routines, and the MAFFinder which is handled specific by the communication services.

The Figure 7.4 on page 45 gives a general overview about the parts of a communication service. The main class is the System Agent itself which maintains all other parts and wraps the method calls through. The proxies for the in- / and output can be implemented in the same class or in two different classes, depending on the protocol used.

Figure 7.4: General Structure of a Communication Services

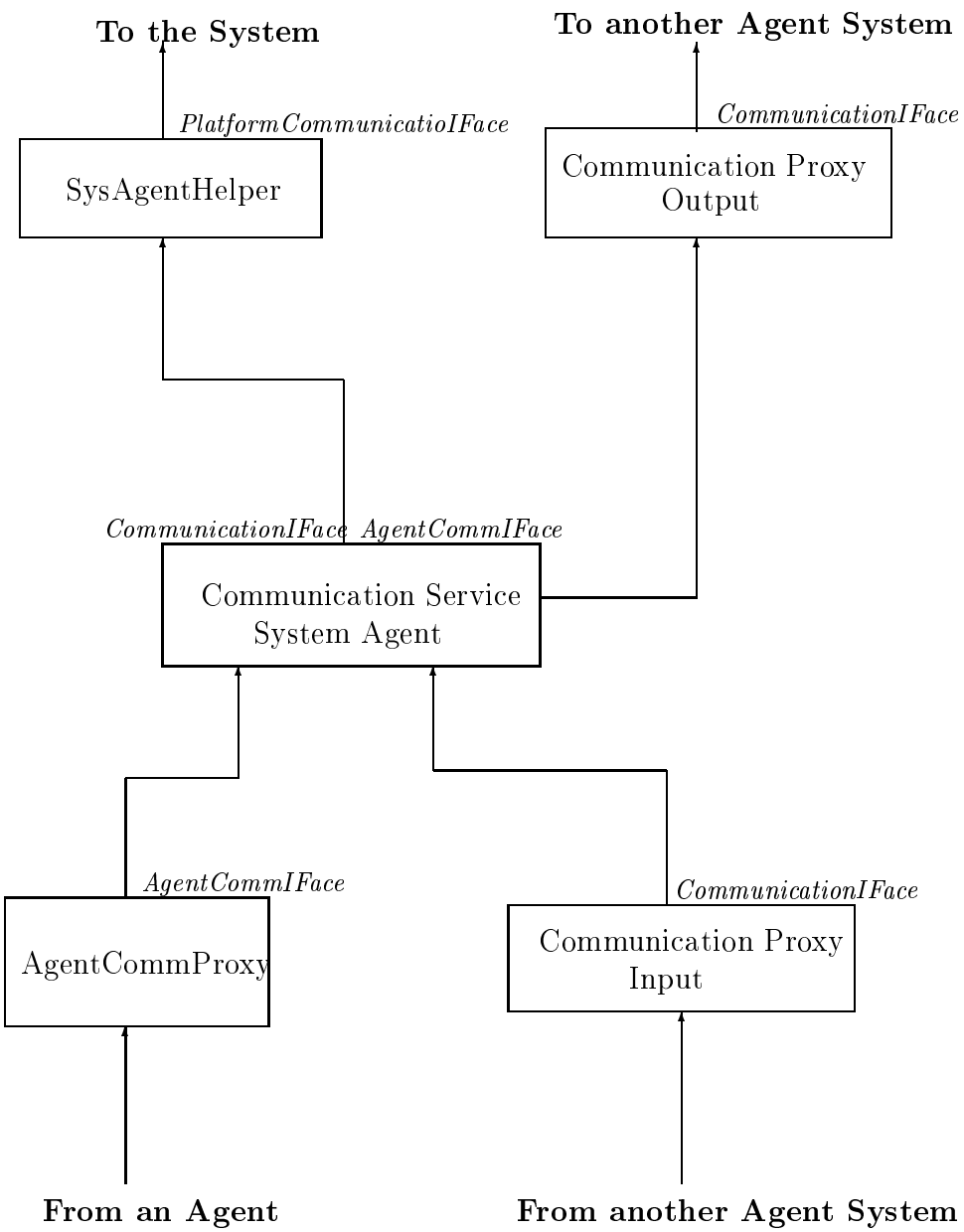
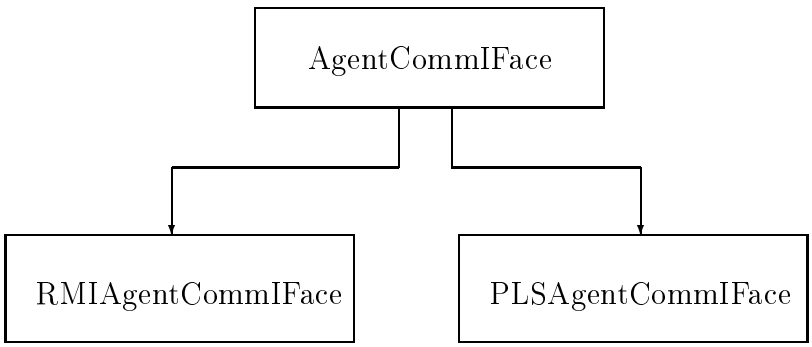


Figure 7.5: Structure of AgentCommIFaces



When an Agent uses the communication service, it calls methods of the `AgentCommProxy` registered in the System. These calls are wrapped through to the calls in the communication service. The communication service preprocesses the calls (like handling internally asynchronous messages) and passes the calls then to the proxy for the output to the other Agent System. The return values are passed the same way back.

When there comes a call from another Agent System it is received by the input proxy. The input proxy gives the calls directly through to the communication service which processes the calls. When a call needs some changes of the System, the responding method of the `SysAgentHelper` is called and this call is wrapped to the System itself. Messages are called on the AgentLoader of the receiving Agent which is gotten from the System. The return value is passed the same way back.

Interface Structures

The general interface for the communication of an Agent is called `AgentCommIFace`. This does not say anything about the protocol used. If an Agent asks for this interface it gets all entries of the communication proxies back.

Every communication service provides a specific protocol and uses its own interface. This interface is derived from the basic `AgentCommIFace` and is named according to the protocol (see Section 7.1.2 on page 32 for details about naming conventions). The interface for the RMI protocol is named `RMIAgentCommIFace` and for the plain socket protocol `PLSAgentCommIFace`. Since both implementing the `AgentCommIFace` this interface is also registered in the platform when the proxy is registered there. The routine to register a proxy registers also sub-interfaces and interfaces of subclasses. The Figure 7.5 shows the hierarchy of the interfaces. All sub-interfaces do not add additional methods, the methods of all the AgentCommIFace and the sub-interfaces are the same.

PlatformCommunicationIFace

The `PlatformCommunicationIFace` is implemented by the `SysAgentHelper` and the System. It provides methods necessary for the communication System Agent.

AgentCommIFace

All the communication that needs an Agent are collected in the interface `AgentCommIFace`. An Agent needs two kinds of communication, first it has to be able to transfer itself to another Agent System and second it has to be able to send messages to another Agents over the internet.

These methods are described in the following subsection as part of moving an Agent and sending messages.

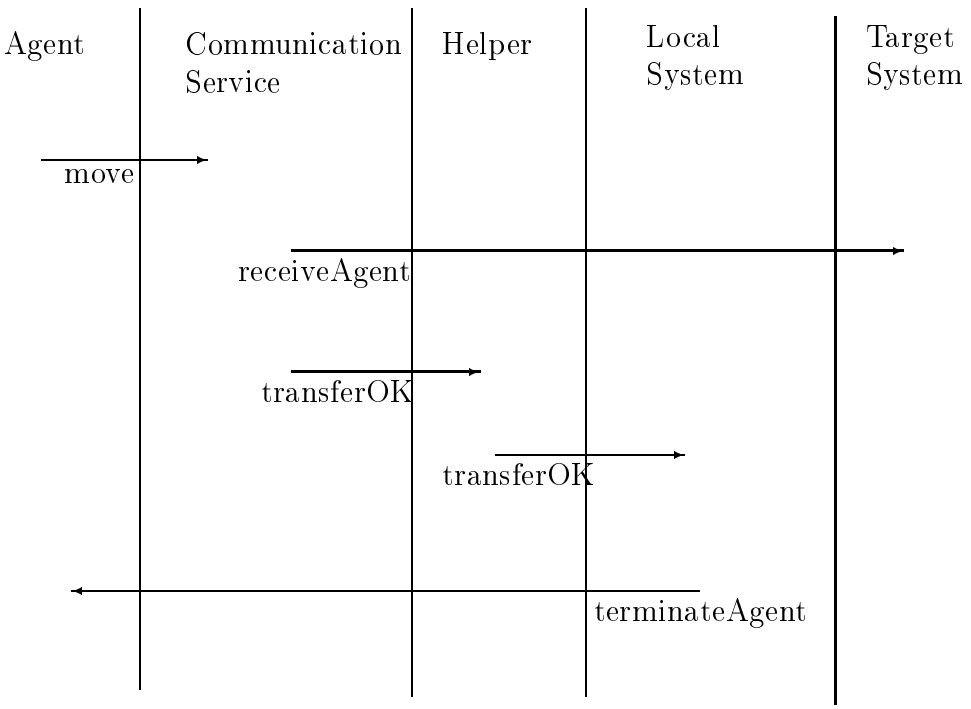
Moving an Agent

When an Agent wants to move to another Agent System it first asks the platform for a proxy which implements the `AgentCommIFace` (or derivates, see Section 7.4.1 for details). Then it calls the `move` method of the proxy with its destination. The proxy handles the call straight to the System Agent. The service then calls the `receiveAgent` method on the corresponding communication service of the target Agent System. This can be done in the System Agent directly or through the output proxy. This depends on the protocol which is implemented. If no exception occurs means that the Agent was started correctly on the target System and can now be killed here. Therefor calls the communication service the method `transferOK` on its `SysAgentHelper`. The helper wraps the call through to the platform which terminates the Agent. See the process diagram 7.6 of moving an Agent.

Sending synchronous Messages

When an Agent wants to send a message it first requests an `AgentCommProxy`. It then calls the method `sendMessage` on this proxy which wraps the method to the communication service. Like with the `move` method the CommService will call the method `executeMessage` directly on the remote communication service or through its output proxy. The target communication service looks up the Agent-Loader for the target Agent by calling the `getLoader` on its helper and then asks the `AgentLoader` to invoke the message on the target Agent. All method calls up to here are blocked until the target Agent returns the result of the message invocation. The return goes the same way back. When an exception occurs it will be given through to the local communication service which wraps them to the two thrown exception `AgentNotFoundException` and `SecurityException`. It uses a similar exception handling like the `RemoteException` of RMI which can contain a nested, different exception.

Figure 7.6: Moving an Agent



Sending FutureMessages

The FutureMessage are kind of asynchronous messages like described in the requirements section. The Agent is not blocked until the response arrives in contrast to the synchronous messages. Instead it gets a `FutureResult` object which will contain the response later when arrived. The Agent can now poll this `FutureResult` object and extract the response when available. If a message can last a longer time this is the recommended method to send messages. Anyway it can be used also for message that do not last very long but are not needed in the next instruction.

An Agent calls the `sendFutureMessage` on its communication proxy. This results in the same call on the System Agent. The System Agent now creates a new, still empty `FutureResult` object. Then it creates a `FutureResultThread` to call the message asynchronously on the remote System, gives it the `FutureResult` object as parameter to put the result later in, starts it and then returns the `FutureResult` object (the reference to it, like every time in java) back to the Agent. The Agent can now continue with its execution. The `FutureResultThread` calls the `executeMessage` on the target Agent in the same way as it is done when sending a synchronous message. When the result arrives it is put into the `FutureResult` object and the Agent can receive it there. See the Figure 7.7 for a process diagram. This diagram is slightly different as the figure from the requirements section, because it describes how this requirement was implemented.

The `AgentCommIFace` contains another asynchronous message,

Figure 7.7: Sending Future Messages

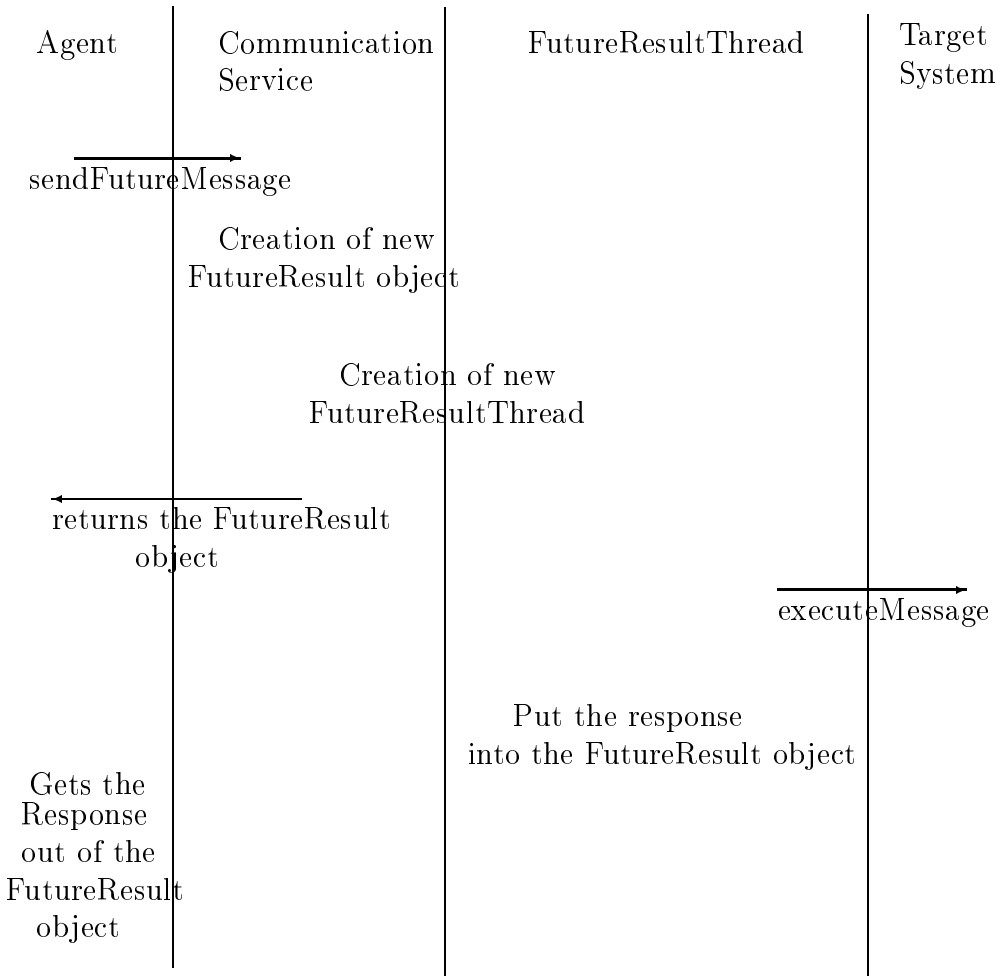
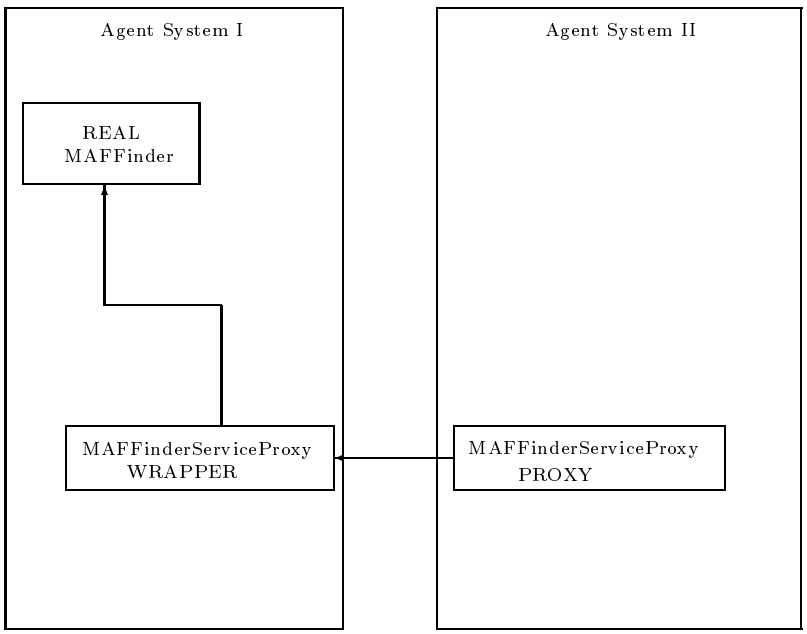


Figure 7.8: Structure of the MAFFinderServiceProxy



`sendAsyncMessage`, which is still not implemented. It could be implemented like in IBM's Aglets[AGLETS] with the use of events when a response arrives. But since the Agent System provides a good asynchronous message sending mechanism it is a not very import task to implement this.

The MAFFinderServiceProxy

The MAFFinder is separated into two parts, the "real" MAFFinder which holds and maintains all registered entries and the proxies which just forward the calls to the "real" MAFFinder. The proxy itself has two different ways of working. It can reside on the same Agent System where also the "real" MAFFinder is and just wrap the incoming calls to the "real" MAFFinder or it can execute in a different machine working as a proxy for the Agents residing there to transfer calls from this machine to the corresponding (same protocol) wrapper on the machine where the "real" MAFFinder resides. Figure 7.8 gives an overview of the parts of the `MAFFinderServiceProxies`. For more details about the MAFFinder please look at the MAFFinder Section 7.4.4 on page 58. The `MAFFinderServiceProxy` is part of the communication services because it is the protocol dependent part of the MAFFinder. The communication services provide all communication while the rest of the Agent System, like the MAFFinder, are designed to be independent from it.

7.4.2 RMI Communication Service

The last section contained already the general contents of a communication service. For this reason this section will treat only the differences and the specialties

of the implementation of the RMI (Remote Method Invocation) protocol.

RMI is a client - server protocol to call methods on a remote object somewhere in the internet. The server offers a set of methods (an interface) and makes this public available in the RMI registry. The client asks the registry for a reference to the published interface and then can call the offered methods. RMI is mostly transparent for the client and for the server. This makes it a easy to use mechanism for accessing remote objects.

RMI is the standard protocol in Java for remote calls through the internet. It is included in all JDKs (1.0.2 and higher). The RMI service offered by the Java class libraries exists of the following parts:

RMI Registry The simple naming facility for RMI. It maintains all server objects exported to it.

Server The server object is published in the RMI registry and is there available for receiving calls from other programs residing on this or other machines.

Client The object calling a method on the remote object.

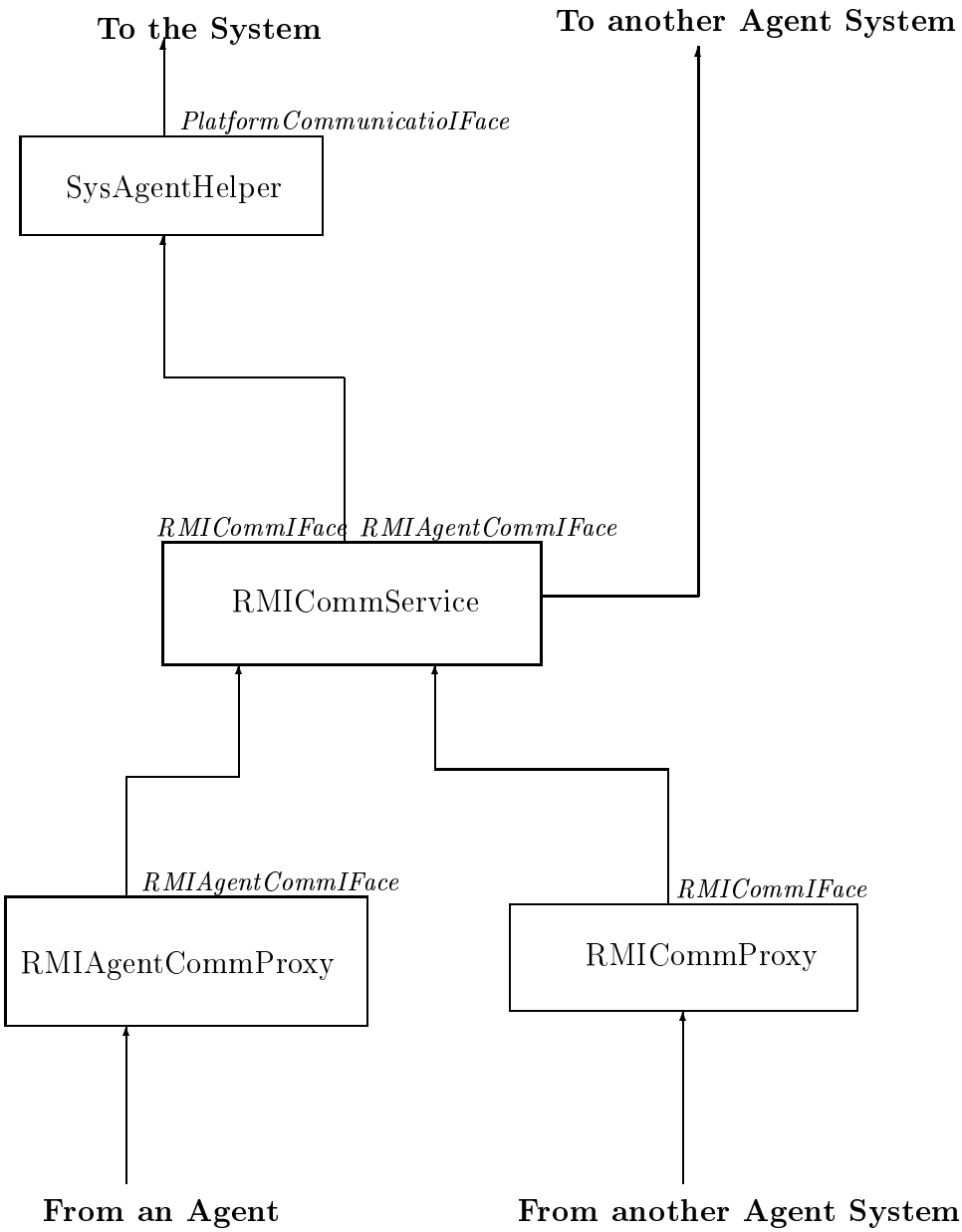
The RMI Registry

The RMI registry is a very important part of the RMI communication. Without it there is no remote method invocation possibly. Normally you have to start the RMI registry manually as a separate program which ships with the JDK or JRE (Java Runtime Environment), but you can also start it automatically from your application. To start the RMI registry manually has a few advantages. It is separate from the Agent System. When the Agent System goes down the RMI registry keeps running and is able to serve other Agent Systems residing on this machine. If the registry is built in one Agent System and there are more Agent Systems running on the same machine, the others loose their capability to communicate through RMI when the System with the RMI registry is going down. The disadvantages are the following. First it is easier to use the built in registry because this does not require any user input. The built in registry is automatically started when no other RMI registry is running. The second advantage is the use of different ports for the RMI registry. The standard port is 1099 but if you have more then one application running on the same machine which is using the RMI services, it is a good idea to start different registry for the different applications. These registry are listening on different ports. If you use the built in registry (which is recommended) you can give a port in the configuration file and do not have to worry about the other registries running on the same machine with e.g. different CLASSPATHs. It could be a task for the future to re-run a new RMI registry if an Agent System cannot connect to the one running on its machine. This would be a very fail - safe construct.

Structure and exported Objects

The main difference between the general communication structure and the RMI communication structure is the missing output proxy. As you can see in Figure

Figure 7.9: RMI Communication Service



7.9, the other parts are very similar with the exception that the RMI specific interfaces and proxies are used. But these interfaces differ just of the name from the original interfaces, the methods are the same. The output proxy is not used because of the possibility of RMI to call remote objects directly without having to use a special proxy object (well it uses the stub - impl mechanism, but this is transparent for the programmer).

There are two objects exported to the RMI registry, the `RMICommProxy` implementing the `RMICommIFace` and the `RMIMAFFinderServiceProxy` implementing the `MAFFinderIFace`. The `RMIMAFFinderServiceProxy` (if in WRAPPER - mode) is exported to let other Agent System access the local MAFFinder, the `RMICommProxy` is exported to the registry to let other `RMICommServices` call their methods on this object remotely.

RMI objects are accessed through a name given to the RMI registry when the object is exported. The names have to be unique on the machine running the registry. TAgents uses the following naming conventions: The MAFFinder wrapper (`RMIMAFFinderServiceProxy` in wrapper mode) is named "MAFFinder" and the `RMICommProxy` is named like the ID of the Agent System which means in the form <ip-address>"/"<system_name>. This was used because there should only exist one real MAFFinder on a machine and therefore only one wrapper (for the RMI protocol) for it. But there can exist more then one Agent System on one machine and to differ them the unique `AgentSystemID` is used.

Configuration Options

In the section of the configuration file for the RMICommService, `[tagents.sysagent.communication.rmi.RMICommService]`, there are three options at this moment recognized by the `RMICommService`:

PORT The built - in RMI registry can use a different port for the registry to listen on. The standard port is 1099 but you can choose every port number which is free on your machine.

MAFFINDER Defines the mode of the `RMIMAFFinderServiceProxy`. Valid options are:

- **REAL** There is a real MAFFinder running on this machine, so the `RMIMAFFinderServiceProxy` is run in "wrapper" mode which means it is exported to the RMI registry and listen to calls to the real MAFFinder from outside.
- **PROXY** If the `RMIMAFFinderServiceProxy` runs as proxy it forwards all local calls to the MAFFinder through the wrapper to the real MAFFinder on another machine. In this mode the `RMIMAFFinderServiceProxy` is exported to the System to let Agents use it to do calls to the MAFFinder. If using this modus you have to give the MAFFINDER_LOCATION all other modus do not require it.

NO If the RMICommService should not use the `RMIMAFFinderServiceProxy` at all. Could be used because there is already one protocol available for accessing the MAFFinder.

MAFFINDER_LOCATION The location of the wrapper to the real MAFFinder. It is an URL used to connect to the wrapper over RMI. It has the form `rmi://ip_number_of_host[:port]/MAFFinder`. If you use a different port as the standard port for the RMI registry one should give here the port of the RMI registry of the machine where the MAFFinder resides. To keep the form in a standard way it is recommended to give always the port. The protocol is of course `rmi` and the name of the object is "MAFFinder" as described in the former section about the RMI naming conventions in TAgents.

7.4.3 Plain Socket (PLS) Communication Service

The plain socket communication service provides the Agent System with the capability to do its communication over plain sockets and to not have to rely on RMI. Some smaller Java implementations (Personal Java Edition, Micro Java Edition) do not have a RMI service implemented and therefore a basic communication protocol was needed for the platform.

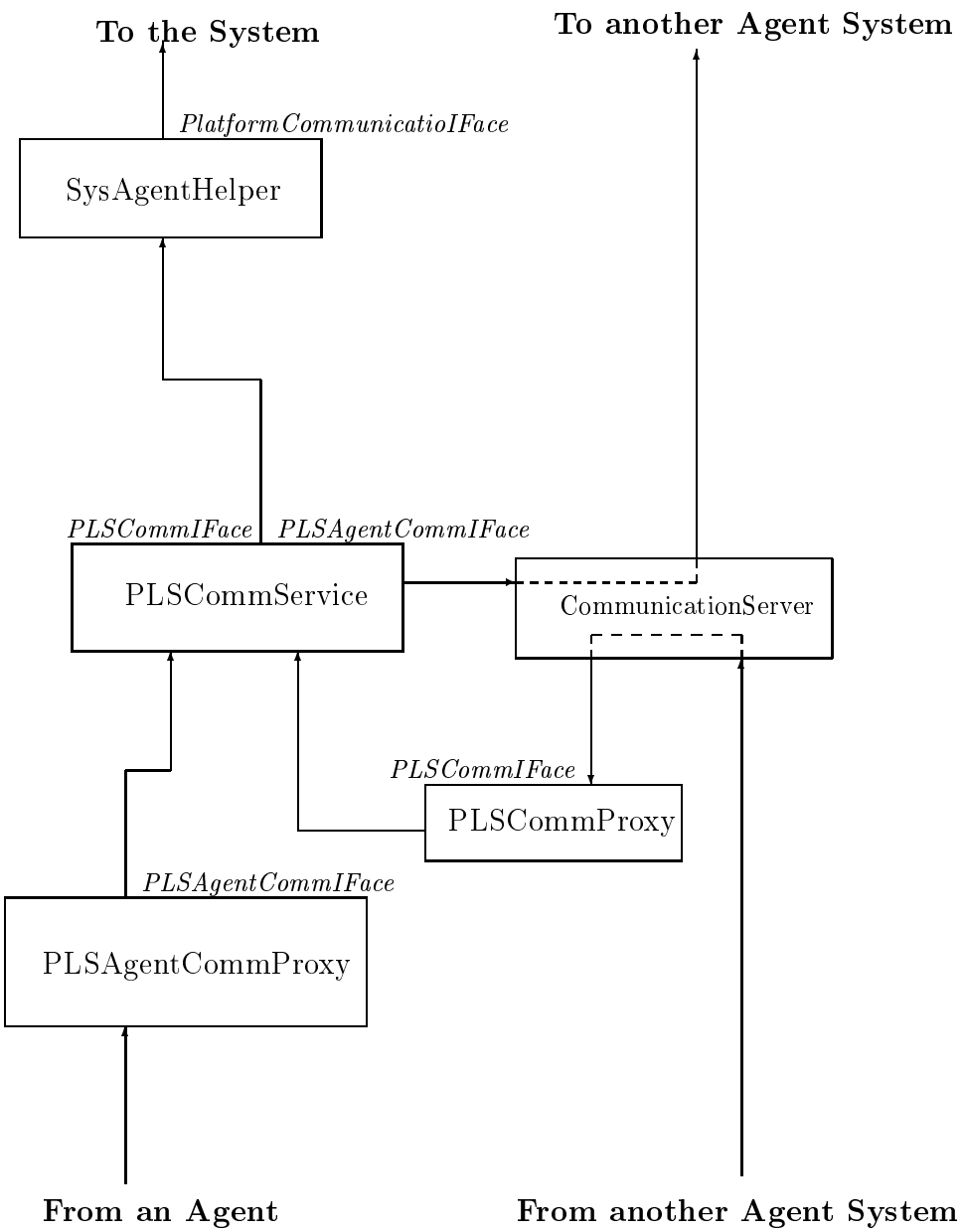
The Figure 7.10 on page 55 shows the structure of the `PLSCommService`. The main difference to the general structure (see 7.4) is the use of the class `CommunicationServer`.

The CommunicationServer

The `CommunicationServer` handles all in- and output over the plain sockets protocol. It uses for that purpose two other objects, the `ConnectionListener` and the class `Connection`. The `Connection` class holds an incoming connection and provides the using class with the input and output streams. Further it contains methods for timeout on the connection level. Since timeout it used on the socket level this option still remains, but is not used. The `ConnectionListener` is a thread which listens on the given port to incoming connections. If a new connection is demanded, it creates a new `Connection` and gives this to the `CommunicationServer`.

The `CommunicationServer` maintains all open connections and calls the desired methods on registered proxies. Also it provides the methods necessary to call methods on another System over the plain socket connections. For calling a method on another Agent System over plain sockets it provides the method `remoteCall(Location target, String methodname, Object[] parameter)`. To connect to another `PLSCommService` it also needs to know the port to connect to. The port can be provided in the target parameter. If is not provided there the method asks the MAFFinder to return the plain socket register entry of the MAFFinder for the given target System. It then calls the method `remoteCall(Location target, String methodname, Object[] parameter, int port)` to send the method to the target System.

Figure 7.10: Plain Socket Communication Service



This method creates a new socket to the target System sets the timeout and sends the method name and parameter to its partner `CommunicationServer` over the net (see subsection "The Protocol used" of Section 7.4.3 on page 56 for details). Then it waits until an answer is sent from it. This answer could be a normal object or, if there was an exception, the thrown exception which is nothing more then an object. If it returned an exception the method throws this exception, else it returns the received answer.

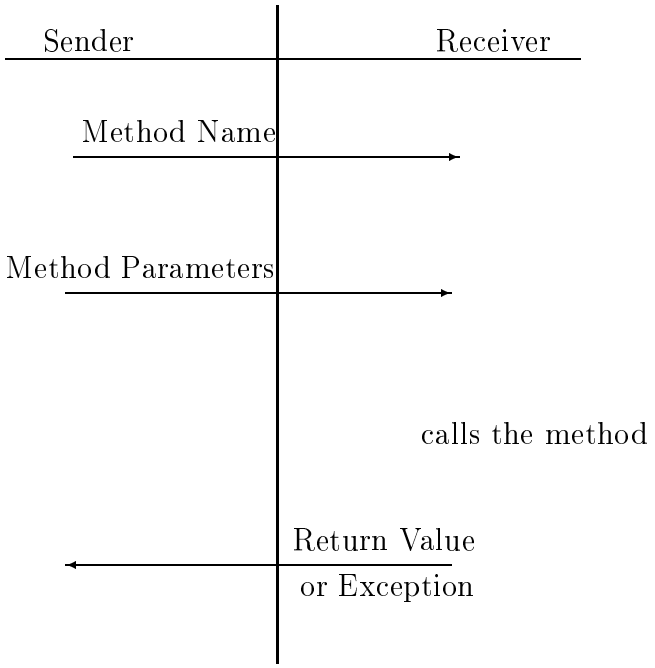
The receiving of calls from outside is also handled by the `CommunicationServer`. A `CommunicationServer` from another Agent System connects to the local communication server and sends it the name and the parameter of the method to execute. This is done by requesting a connection which results locally in creation of a new `Connection` object. When new data is available on this `Connection` object the two objects (name and parameter) are read from it. In order for the communication server to know what methods can be called and on which object, all "call - receiving" objects have to be known by the communication server. It uses therefor a very similar mechanism as the plug in mechanism of the platform. A receiver object registers itself at the `CommunicationServer` using the method `registerProxy(Object obj)`. The registered objects are stored in a `Vector`. There are two objects that are being registered in the `CommunicationServer`: The `PLSCommProxy` and the `PLSMAFFinderServiceProxy` (in wrapper modus). When now a request arrives for calling a method, the server looks at the methods the registered proxies offer and calls the one with the matching name. In this version there is no check for the right parameters implemented, resulting in the requirement to have only different names in the proxies. Since there are only a few proxies used (`PLSMAFFinderServiceProxy` and `PLSCommProxy`) it is very easy to keep this requirement.

This is also very similar to the registering of objects in the RMI registry, which also has to know what object can be called. After calling the desired method the return value is sent back to the requester. If an exception has occurred while calling the method, the exception is sent instead of the (in this case not existing) return value. This exception is then thrown on the senders machine, as described before.

The Protocol used

The protocol which is used by the communication service is quite simple. Ever method exists of the name of the method, a `String`, and a array of `Objects`, the method parameters. To call a method the caller has to provide (send) a string and a array of objects (which is handled in Java as a object itself). It then gets back a objects containing the return value or if an exception occurs the object representing the exception. See Figure 7.11 on page 57 for a process diagram of the protocol.

Figure 7.11: Protocol of the Plain Socket Communication



Configuration Options

In the section of the configuration file for the `PLSCommService`, `[tagents.sysagent.communication.pls.PLSCommService]`, there are three options at this moment recognized by the `PLSCommService`:

PORT The `CommunicationServer` (or the `ConnectionListener`) needs a port given to listen to incoming calls. The standard port is 3333 but you can use every port number which is free on your machine.

MAFFINDER Defines the mode of the `PLSMAFFinderServiceProxy`. Valid options are:

- **REAL** There is a real MAFFinder running on this machine, so the `PLSMAFFinderServiceProxy` is run in "wrapper" mode which means it is registered to the `CommunicationServer` as receiver and listens to calls to the real MAFFinder from outside.
- **PROXY** If the `PLSMAFFinderServiceProxy` runs as proxy it forwards all local calls to the MAFFinder through the wrapper to the real MAFFinder on another machine. In this mode the `PLSMAFFinderServiceProxy` is exported to the System to let Agents use it to do calls to the MAFFinder. If using this modus you have to give the `MAFFINDER_LOCATION` all other modus do not require it.
- **NO** If the `PLSCommService` should not use the `PLSMAFFinderServiceProxy` at all. Could be used because there is already one protocol available for accessing the MAFFinder.

MAFFINDER_LOCATION The location of the wrapper to the real MAFFinder. It is an URL used to connect to the wrapper over PLS. It has the form `pls://ip_number_of_host:port/name_of_system`. You have always to give a port for the wrapper. There is still no standard port used per default. The protocol is of course `pls` and the name of the System is just the name and not the whole ID of the System.

7.4.4 MAFFinder Service

This thesis treated already about some of the parts of the MAFFinder in the previous chapters. This chapter should give now a detailed description of what the MAFFinder is and where it used. The MAFFinder is a very essential part of the whole *Mobile Agent System*. It provides mainly methods to register parts of the System and to search for them.

Structure

The MAFFinder exists of the parts shown in the Figure 7.12 on page 59. There exists one main MAFFinder (the "REAL" MAFFinder) which holds all information and provides it through the proxy infrastructure. It registers its `MAFFinderProxy` to the platform and the Agents, System Agents and the System are calling their methods through this proxy. The `MAFFinderServiceProxy` is implemented by the communication services and works in two modes. If it works as wrapper it has to reside on the same machine as the real MAFFinder and listens to calls from other Agent Systems to wrap them to the MAFFinder. It also uses the `MAFFinderProxy` from the platform and calls its method when getting a request from outside. If working as proxy it resides on a machine without a real MAFFinder and registers itself in the platform. The Agents, System Agents and the System there are calling its methods which are first forwarded over the net to the wrapper and then to the real MAFFinder. Since there exists only one real MAFFinder in a region (the region is defined as a group of Agent Systems having one MAFFinder) there is no problem with more databases and their synchronization.

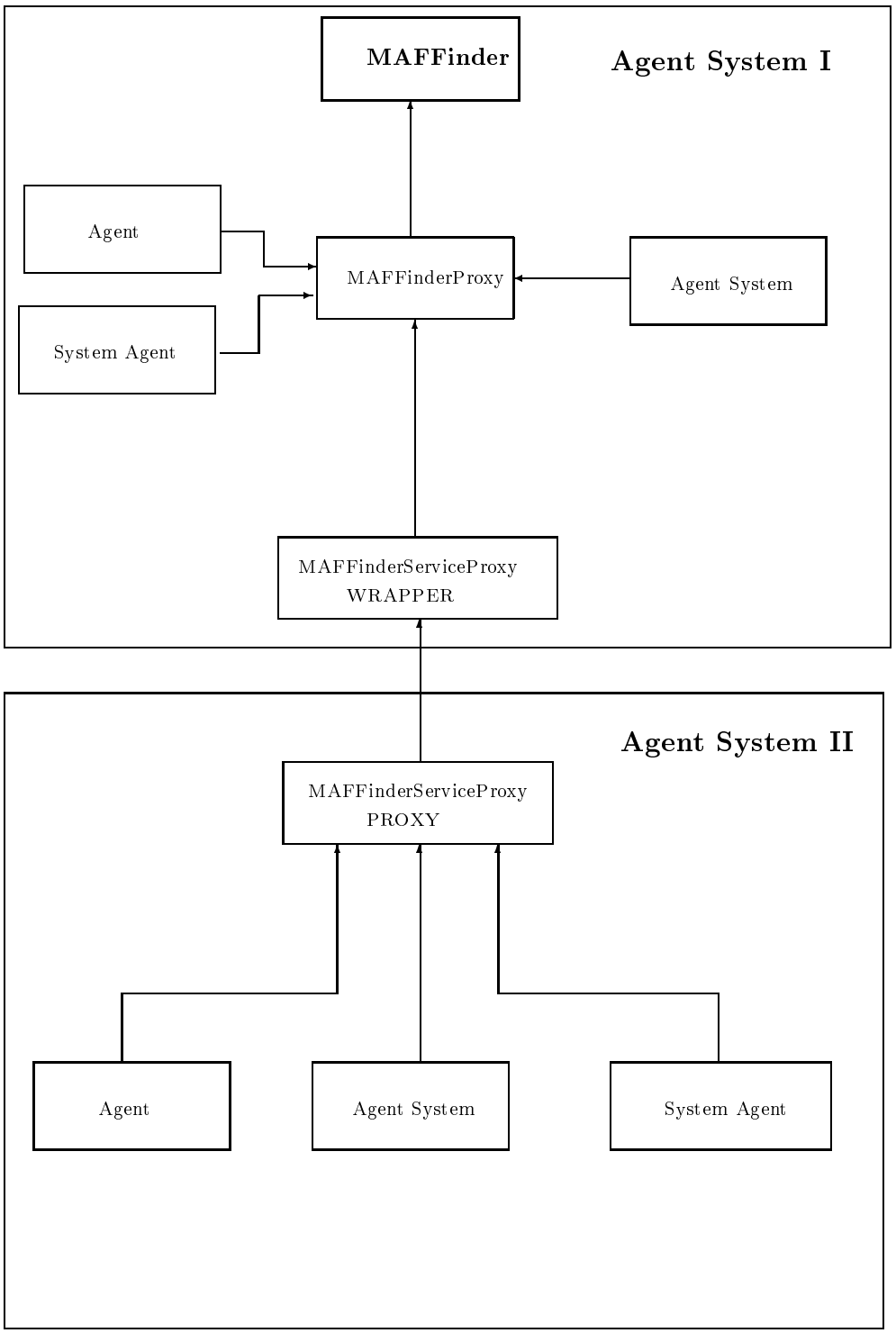
Information provided

The MAFFinder is used to register the following parts of the System:

Agent Systems Every Agent System has to be registered in the MAFFinder. This is done when starting the System in the `init()` method. When there is already an Agent System registered with the same name the new Agent System will exit with an error. Therefor it checks first if its name is already assigned to an Agent System by doing a lookup on its name and the MAFFinder should return an empty array.

Agents Every Agent registers itself in the MAFFinder when it is started. When an Agent moves to another System it is restarted (resumed) there again and will register itself with the new Location. The Agent can register

Figure 7.12: Structure of the MAFFinder



itself more times with different Location and the last registration will be kept in the internal database of the MAFFinder to provide always only the actual information about this Agent.

System Agents System Agents are normally not registered in the MAFFinder. This could be changed in the future to make the MAFFinder also to a registry of resources. The only System Agents registered in the MAFFinder are the communication services. Every communication service uses a certain port to listen to and to be connected from other Agent Systems. To provide this information every communication System Agent registers itself with its listener port with the MAFFinder. When now another communication service wants to connect to it, it asks the MAFFinder for the registry entry with the desired port to be used to connect to it. Also it provides an actual database of which communication services are available to access a certain Agent System. The communication services register not really themselves in the MAFFinder but the Agent System where they are residing. The Agent System itself registers without giving a protocol and port, the communication services register themselves with the same name but with a protocol and port. By querying for an Agent System one gets the entry of the Agent System back and a list of all possible communication protocols to access it. This is used by the basic Agent class to look for an appropriate communication method for a given Agent System.

Places Although places are not totally supported in this version of *TAgents*, the MAFFinder has already the methods to register places. A place should be used by a special group of Agents and therefor the naming of a place should be standardized in the Agent System.

As one can see, without a MAFFinder an Agent System cannot work properly. For example there would be big problems for doing the communication between two Agent Systems or for finding other Agents. Therefor it could decided in future to build a backup MAFFinder on another System. This backup MAFFinder would have to have the same entries as the real one and would have to be kept up to date whenever a entry changes. This could be done by using the same mechanisms as the `MAFFinderServiceProxies` to keep the other MAFFinders up to date. Also the MAFFinder is prepared to be able to move to another place which could be necessary when the Agent System is going down. The MAFFinder with all its data could transfer itself to the next System and continue work there. The problem is that all the MAFFinderServiceProxy would have to get the new address of the MAFFinder. But the changes should be not to much.

The real MAFFinder

The real MAFFinder maintains all the data records of the different parts of the Agent Systems in the region. It uses therefor the following hash tables:

agents A standard hash table for all Agents which are registered in the MAFFinder. The format of the entries is `<Name agentname>` :

<Location agentlocation>. It is used to get the location of an Agent with a certain name.

agentprofiles An OTMObjectHashtable for the profiles of the registered Agent. The format of the entries is <AgentProfile profile> : <Name agentname>. It is used to get the names of the Agent that fit into the given AgentProfile. The Locations of these Agents are stored in the agents hash table. An OTMObjectHashtable is used here since the format of all entries in the AgentProfile is still not totally defined and could be equal for more Agents.

systems An OTMObjectHashtable for all Agent Systems that are executing in the region of this MAFFinder. The format of the entries is <Name systemname> : <Location systemlocation>. It is used to get the Location of the System with the given name and to get all communication methods available there. An OTMObjectHashtable is used here because the communication services register themselves with the name of their Agent System and the Location with the protocol and port used to access them. So there are more then one entry for the same Agent System.

systeminfos An OTMObjectHashtable for the infos about the registered Agent Systems. The format of the entries is <AgentSystemInfo systeminfo> : <Name systemname>. It is used to get names of the Agent System fitting to a special AgentSystemInfo object. An OTMObjectHashtable is used here since the format of all entries in the AgentSystemInfo is still not totally defined and could be equal for more Agent Systems.

places A Hashtable used to store all registered places. The format of the entries is <String placename> : <Location location>. In the MASIF standard places are handled as if they are unique in their names. This is the reason to use a normal hash table here. But if the use of places get more defined it could be necessary to switch to OTMHashtables because of equal places on different Systems.

The MAFFinder implements the MAFFinderIFace which provides the same methods as described in the MASIF standard in Section 3.2.2 on page 13. The most of the work is defined well in the standard and implemented in the same way. The major difference is the handling of the AgentSystemInfo and the AgentProfile. These are not very well described in the standard and there could be some changes in the future. The next section contains more details about this.

Auxiliary Classes

The two classes used in special manner in the MAFFinder are the AgentSystemInfo and the AgentProfile. Both hold additional information about the corresponding parts of the Agent System. These information is in detail:

AgentSystemInfo It contains the name of the System (`agentSystemName`), the type (`agentSystemType`), the languages supported (`languageMaps`) with their serialization mechanism, a short description (`agentSystemDescription`), the major version number (`majorVersion`), the minor version number (`minorVersion`) and a list (array) of undefined properties (`properties`). The properties are not described in the standard and they are still not used for Agent Systems in *TAgents*.

AgentProfile It contains the language of the Agent (`languageId`), the type of the creator System (`agentSystemType`), the description of the creator System (`agentSystemDescription`), the major and minor version numbers (`majorVersion`, `minorVersion`), the type of serialization used (`serialization`) and a list of undefined `properties`. The properties are from the same kind as in the `AgentSystemInfo` but there is already one entry used for *TAgents*. The first object of the properties array is defined as the name of the class of the Agent. This could be used also as a small description of the Agent like the, here missing field but existing in the `AgentSystemInfo`, description.

Both classes provide a method called `fit`. This method compares only the defined entries of the classes. Every entry, which are defined in the class `MASIFConstants`, has a id (constant) for not being set. For example with setting the `minorVersion` to `MINOR_VERSION_NOT_SET` (-1) means that this field should not be compared. The fit method would return true even if the other object minor version is defined. This is used for a indistinct (fuzzy) search for Agents or AgentSystems which fit into some requirements. If you look after an Agent or Agent System in the MAFFinder it first checks if the name parameter is not `null`. If the name (of the Agent or Agent System) is defined it returns the location for this name. If it is not defined it checks all stored `AgentProfiles` (or `AgentSystemInfos` when looking for an Agent System) for the ones to which the given profile fits into. These are then returned back to the caller. You can for examples look for all Agents of the class "`AgentEx3`" by calling the `lookupAgent` with an empty `Name` and the `AgentProfile` with just the first entry of the `properties` array filled. All other fields of the profile have to be set to their "*_NOT_SET" values.

Standard Conformity

The MAFFinder was designed to be complaint to the MASIF standard. The main differences in this version to the standard are the following:

The name of methods of the `MAFFinderIFace` are changed to be conform to the Java code writing style from SUN[SUN]. This means that all method names containing more then one word (e.g. `lookupAgent`) are written with the first word lower case but internal words capitalized. This means for example for the MASIF MAFFinder method `lookup_agent` that it is called `lookupAgent` in Java. The parameter of all methods are the same, using Java classes for CORBA definitions.

The MASIF standard uses CORBA for all communication. The design of the MAFFinder in the *TAgents System* is basically independent of the communication form. The protocol dependent parts are separated into the MAFFinderServiceProxies, like there are implemented the `RMIMAFFinderServiceProxy` and the `PLSMAFFinderServiceProxy`. To make the MAFFinder here compatible to the standard one would have to write a CORBA communication service and providing it with a `CORBAMAFFinderServiceProxy`. This proxy would have to implement the `MAFFinderIFace` with the corrected names to fit exactly to the standard and would have to make the real MAFFinder in this way accessible for other (not *TAgents*) Systems over CORBA. This task should be able to do very similarly as the `RMICommService` and it should be possible to re - use many parts of the RMI protocol implementation.

7.4.5 Admin System Agents

This section will describe the admin System Agents. The admin System Agents are used to provide a administration interface for the System. At the time of writing there exist two different administration services, the `ConsoleAdmin` which has a text user interface and the `GUIAdmin` which has a graphical user interface (gui).

The Output Framework

The output framework was already described in Section 7.3.5 on page 40 as part of the System. This section will describe the part of the admin System Agent using the output framework.

The output of all parts of the System is sent to the platform by using the `print / println` methods provided by the helper classes and the System. The platform forwards all the output to all registered admin System Agents. The admin services have to register therefor the interface `AdminIFace` in the platform. This is done by registering the proxy `AdminProxy` which is part of the `tagents.sysagent.admin` package.

SystemAdminIFace

The `SystemAdminIFace` was designed to provide the admin of special changes of the (System) Agents residing in the platform. An admin service registers therefor the corresponding `SystemAdminProxy` to the platform. When a (System) Agent is created, stopped, resumed or terminated the platform calls the method to let the service know that there has changed something in the System. This is not used for the `ConsoleAdmin` but for the `GUIAdmin` to keep the list of Agents up to date. The interface provides the following methods for this purpose:

agentCreated(AgentID id) A new Agent or System Agent with the given ID was created. It will the be added to the list of running Agents in the admin service.

agentStatusChanged(AgentID id) The status of the given Agent has changed. The status is described in the `AgentLoader` and in the `SysAgent` class. It gives information about the execution state of the Agent which can be for example running, stopped or concluding. The complete list is in the `AgentLoader` class for Agents and in the `SysAgent` class for System Agents.

agentTerminated(AgentID id) The Agent was terminated and has to be removed from the lists.

The ConsoleAdmin

The ConsoleAdmin provides a text based command line interface for controlling the System. It writes all output to the standard output stream as defined in `java.lang.System.out`. The output of the belonging parts of the System is not written synchronously to the output stream as this could result in very mixed output. If for example two Agents writing at the same time some lines, these lines would be put mixed on the console. To prevent this case the `ConsoleAdmin` collects the output of the (System) Agents and the System in a hash table of string buffers and writes them by using a thread at once on the display. The thread used for the collected output is called `ConsoleAdminOutputThread` and does nothing more then looking at all string buffers and, if some ouput is available, writing the lines on the screen. The system buffer is handled first to provide it with a higher priority. The `run` method sleeps in every loop one second if no new data is available. If the `ConsoleAdmin` gets a request to write something on the screen it puts the lines into the belonging string buffer and interrupts the `ConsoleAdminOutputThread`. The mechanism prevents the thread from using too much CPU time, but keeps the delay time between calling one of the print method and the output on the screen very small.

The second thread used by the `ConsoleAdmin` is responsible for the input of the administrator. The `ConsoleAdmin` provides the system administrator with a command prompt and a small help. The command prompt "`ConsoleAdmin >`" tells the administrator that the admin service is ready for commands from the user. The prompt could be invisible sometimes caused by the asynchronous output mechanism. If this is the case it will rewritten to screen when pressing just `<RETURN>`. The `ConsoleAdmin` recognizes the following commands, a up to date list is always available with the command "`help`":

help Prints a list of the available commandos and a small description on the screen.

terminateSystem Shuts down the System. The command is also available through writing "`quit`". The `ConsoleAdmin` calls the `shutdown` method of the platform when given this command.

listAgents Writes a list of all Agents and System Agents on the screen. The list consists mainly of the strings representing the `AgentID` (`AgentID.toString()`). The `ConsoleAdmin` queries therefor the platform for the complete and up to date list of Agents and System Agents.

listProxies Writes a list of all registered interfaces and their belonging proxies on the screen. This list is split by interfaces and not by proxies so that proxies that implement more then one interface are appearing more times.

listThreads List all threads belonging to the main thread group and their sub-groups. This should list all threads executing in the System, because Agents should execute their threads in their desired group and System Agent can execute in the main thread group. The output is done by calling the `ThreadGroup.list()` on the main thread group. This command lists the threads then recursively through all subgroups.

createAgent <full_name> Creates a new Agent with the class name as parameter. The class name parameter has to be the full package path to the class, for example "`tagents.examples.AgentEx3`". When using authority based authentication in the System in future it could take another parameter for the authority. The Agent will be started automatically when calling the method.

stopAgent <id> Stops the given Agent. The Agent to stop is identified by the id string given as parameter. The id string is converted into an `AgentID` and then the `ConsoleAdmin` calls the `stopAgent` method of the platform.

startAgent <id> (Re-) Starts the given Agent. This is be done by calling the `resumeAgent` of the platform.

concludeAgent <id> Concludes (terminates) the given Agent.

totalMemory Shows the amount of memory available in the JVM and the bytes already used by the System.

gc Runs the java garbage collector. This should not be necessary to call since the JVM provide a asynchronous garbage collecting mechanism. This option was used for debug purposes.

The GUIAdmin

The `GUIAdmin` provides a graphical user interface for controlling and administrating the Agent System. It uses the swing classes as window toolkit. The swing classes are an additional toolkit for JDK1.1.x and standard in Java 2 (1.2.x and higher). For enable the System to be executed under a JDK 1.1.x the swing classes are provided as jar file in the `lib` subdir of the *TAgents* directory. The swing classes have several design enhancements and advantages over the normal java window toolkit, the AWT.

The main Window The main window (see Figure 7.13) contains two parts, the tree of all Agents and System Agent and the information of the selected entry. For the portrayal of the parts of the System, it uses a `JTree`. The `JTree` draws a tree on the screen and holds its information in a list of nodes and leaves.

The root node is the Agent System and the next two nodes are the Agents and the System Agents. The leaves of the `JTree` contain a general object which represents the content of the leave. In this version an `AgentID` is used as this content object. The string appearing on the screen for every leave is the result of the `AgentID.toString()` method. It gives all information to identify the Agent represented by the leave.

The right side of the window should give information about the Agent as the ID, authority, class name and status. The ID, authority and class name are extracted of the `AgentID`. For getting the status the platform is queried when creating an Agent and when getting a call to `agentStatusChanged` by the System. The `GUIAdmin` uses the `SystemAdminIFace` / `-Proxy` for being informed when an Agent is created, terminated or its status has changed. When the status of an Agent has changed queries the `GUIAdmin` the platform for the new status.

Commands

There are two main menus already in this version of *TAgents*. The System menu is for controlling the System and the Agent menu is for maintaining the Agents (as well as the System Agents). The respective menu items are:

System

Show History Opens the history window for the System output.

Shutdown Shuts down the System.

Agent

New Agent Lets the user create a new Agent.

Send Message Lets the user send a message to the selected Agent. This is still not implemented in this version of *TAgents*.

Start (Re-) Starts the selected Agent. If the Agent is already started nothing will be done. The actual status of the Agent can be viewed on the right column of the window.

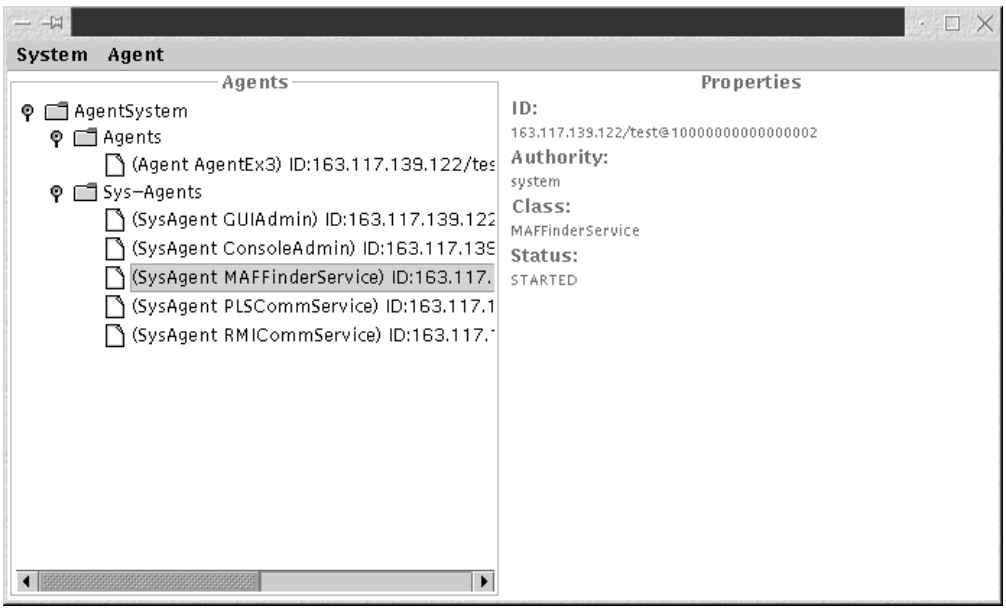
Stop Stops the selected Agent.

Conclude Terminates the selected Agent. The Agent will then disappear from the screen is not accessible anymore. An administrator should be careful with concluding an admin System Agent, possibly leaving the System without any admin System Agent. This would mean that it would not be able to administer anymore. The same is valid for the stop commando.

Show History Shows the history window with the output for the selected Agent.

The output of every Agent and of the System is shown in the history windows. The history window contains a text field with all the output this Agent

Figure 7.13: GUIAdmin Main Window



has made while executing in this System. When there are more lines than available on the screen the text field gets some scrollbars and is so able to contain all the output. The next images show the main window of the `GUIAdmin` and the history windows of a normal Agent, a System Agent and the System itself.

7.5 Agents

This section should describe the basic structure of an Agent, the basic Agent class and the helper and loader classes.

The `Agent` class had to meet several requirements. It should provide a good base to let the user write their own Agent. The use of the basic services as communication and the MAFFinder should be as easy as possible. Also should have the Agent itself an easy to understand and to extend structure. On the other side is an Agent considered as possible foreign code from which the System and the other Agents have to be protected. This results in a kind of sand box in which an Agent executes.

7.5.1 AgentIFace

The `AgentIFace` provides the public interface for basic access to an Agent. It provides the following methods:

initAgent This method is called by the `AgentLoader` at the time of creating the Agent. It is only called once in the life time of an Agent.

startAgent This method is called when the Agent is (re-) started. It is first called after the initAgent to start the just created Agent in the System.