# *Grasshopper Programmer's Guide*

Release 2.2

**IKV++ GmbH**
**Bernburger Strasse 24-25**
**10963 Berlin, Germany**
**http://www.grasshopper.de**

**Copyright (c) 1999 IKV++ GmbH Informations- und Kommunikationssysteme**

**All Rights Reserved.**

Grasshopper Release 2.2 Programmer’s Guide, März 2001

The *Grasshopper Programmer’s Guide* is copyrighted and all rights are reserved. Information in this document is subject to change without notice and does not represent a commitment on the part of IKV++ GmbH. The document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from IKV++ GmbH.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.
Microsoft, Windows, and Windows NT are trademarks or registered trademarks of Microsoft Corporation in the U.S. and other countries.
All other names are used for identification purposes only and are trademarks or registered trademarks of their respective companies.

Grasshopper Release 2.2 is based on Grasshopper Release 1.x which is copyrighted by the Research Institute for Open Communication Systems (FOKUS), a department of the National Research Center for Information Technology (GMD), Germany.

IKV++ GmbH
Informations- und Kommunikationssysteme
Bernburger Strasse 24-25
D-10963 Berlin
Germany
Email: ikv@ikv.de
URL: http://www.ikv.de

# Contents

**1 Preface** .......... **1**
- 1.1 About this Document .......... 1
- 1.2 Document Structure .......... 1
- 1.3 Related Documents .......... 2
- 1.4 Notational Conventions .......... 3
  - 1.4.1 Fonts .......... 3
  - 1.4.2 Icons .......... 3
- 1.5 How to Get in Contact .......... 4

**2 Introduction** .......... **5**

**3 Hello Agent!** .......... **13**
- 3.1 Example: HelloAgent .......... 19
- 3.2 Summary .......... 20

**4 Creation and Removal of Agents** .......... **23**
- 4.1 Agent Creation .......... 23
- 4.2 Agent Removal .......... 25
- 4.3 Example: PrintStringAgent .......... 26
- 4.4 Summary .......... 29

**5 Agent Related Information** .......... **31**
- 5.1 Identification .......... 35
- 5.2 Names and Descriptions .......... 36
- 5.3 Code base .......... 37
- 5.4 Grasshopper Addresses and Locations .......... 40
- 5.5 States and Life Cycles .......... 42
- 5.6 Example: PrintInfoAgent .......... 44
- 5.7 Summary .......... 46

**6 Move Me!** .......... **49**
- 6.1 Strong vs. Weak Migration .......... 49
- 6.2 The Migration Procedure .......... 51
- 6.3 The Data State: Mobile Information .......... 52

6.4 Structuring an Agent's Life ....................................................54

6.5 Example: BoomerangAgent ....................................................55

6.6 Summary ....................................................58

**7 Action! .................................................... 61**

7.1 Example: ActionAgent ....................................................61

7.2 Summary ....................................................64

**8 Clones and Copies .................................................... 67**

8.1 Example: CopyAgent ....................................................68

8.2 Summary ....................................................73

**9 The Communication Service .................................................... 75**

9.1 Implementing the Server Side ....................................................77

9.2 Creating Proxy Objects ....................................................77

9.2.1 Manual Proxy Generation ....................................................78

9.2.1.1 Usage of the Stub Generator .................................... 79

9.2.2 Dynamic Proxy Generation ....................................................80

9.2.3 Issues of Mixed JDK Environments ....................................................80

9.3 Implementing the Client Side ....................................................81

9.4 Simple Communication Scenario ....................................................83

9.4.1 Example: ServerAgent ....................................................83

9.4.2 Example: ClientAgent ....................................................85

9.4.3 Running the Scenario ....................................................87

9.4.4 Summary ....................................................89

9.5 Sync. vs. Async. Communication ....................................................90

9.5.1 Asynchronous Provision of Results ....................................................91

9.6 Asynchronous Communication Scenario ....................................................97

9.6.1 Example: AsyncServerAgent ....................................................98

9.6.2 Example: AsyncClientAgent ....................................................100

9.6.3 Running the Scenario ....................................................109

9.6.4 Summary ....................................................112

9.7 Static vs. Dynamic Method Invocation ....................................................112

9.8 Dynamic Communication Scenario ....................................................114

9.8.1 Example: DynamicServerAgent .................................. 115
9.8.2 Example: DynamicClientAgent .................................. 118
9.8.3 Running the Scenario .................................. 125
9.8.4 Summary .................................. 127
9.9 Unicast vs. Multicast Communication .................................. 127
9.10 Multicast Communication Scenario .................................. 131
9.10.1 Example: MulticastServerAgent .................................. 131
9.10.2 Example: MulticastClientAgent .................................. 132
9.10.3 Running the Scenario .................................. 136
9.10.4 Summary .................................. 138
9.11 Accessing Agencies .................................. 139
9.11.1 Agency Related Information .................................. 139
9.11.2 Interface IAgentSystem .................................. 141
9.11.3 Local Access .................................. 144
9.11.4 Remote Access .................................. 145
9.11.5 Listening to Agencies .................................. 146
9.11.6 Example: AgencyClientAgent .................................. 149
9.11.7 Summary .................................. 162
9.12 Accessing an Agency Domain Service .................................. 163
9.12.1 Interface IRegion .................................. 164
9.12.2 Interface IRegionRegistration .................................. 165
9.12.3 Local Access .................................. 168
9.12.4 Remote Access .................................. 169
9.12.5 Listening to Region Registries .................................. 170
9.12.6 Example: RegionClientAgent .................................. 173
9.12.7 Summary .................................. 180
9.13 Searching Grasshopper Components .................................. 180
9.14 Migrating Servers and Clients .................................. 187
9.15 Migration Scenario .................................. 189
9.15.1 Example: MigratingServerAgent .................................. 189

9.15.2 Example: MigratingClientAgent ..................................191
9.15.3 Running the Scenario....................................................199
9.15.4 Summary.......................................................................203
9.16 Interacting with External Applications ...................................204
9.17 External Communication Scenario .........................................206
9.17.1 Example: ExternalApplication .....................................206
9.17.2 Example: ExternalAccessAgent ...................................211
9.17.3 Running the Scenario....................................................214
9.17.4 Summary.......................................................................216
**10 The Persistence Service.................................................................... 217**
10.1 Example: SleepyAgent...........................................................219
10.2 Summary ...............................................................................222
**11 Special Places..................................................................................... 225**
11.1 Example Scenario for Special Places.....................................227
11.1.1 Example: PlaceService .................................................227
11.1.2 Example: PlaceAccessAgent ........................................228
11.1.3 Running the Scenario....................................................231
**12 The Security Service ......................................................................... 235**
12.1 External Security....................................................................235
12.2 Internal Security .....................................................................236
12.3 Example: SecretAgent.............................................................237
**13 Grasshopper and CORBA................................................................ 241**
13.1 CORBA Enhanced Grasshopper Agents.................................242
13.1.1 Example: CORBA Enhanced Agents ...........................246
**A Acronyms ..........................................................................Annex - 1**
**B Glossary.............................................................................Annex - 3**
**C Index ...............................................................................Annex - 21**

# List of Figures

Figure 1: Agent Class Diagram.................................................................. 15
Figure 2: AgentInfo Class Diagram........................................................... 31
Figure 3: Agent State Diagram ................................................................. 44
Figure 4: Agent Migration ........................................................................ 50
Figure 5: Structure of an Agent’s live() Method ....................................... 54
Figure 6: BoomerangAgent Scenario.......................................................... 57
Figure 7: ActionAgent Scenario ................................................................ 64
Figure 8: CopyAgent Scenario................................................................... 72
Figure 9: Communication via Proxies ....................................................... 75
Figure 10:Simple Communication Scenario............................................... 88
Figure 11:Asynchronous Communication Scenario................................ 110
Figure 12:Dynamic Communication Scenario ......................................... 126
Figure 13:Multicast Communication Scenario......................................... 137
Figure 14:AgencyInfo Class Diagram...................................................... 140
Figure 15:IAgentSystem Class Diagram .................................................. 141
Figure 16:Listening Mechanism for Agencies ......................................... 149
Figure 17:AgencyClientAgent Scenario................................................... 161
Figure 18:IRegion Class Diagram ........................................................... 164
Figure 19:IRegionRegistration Class Diagram......................................... 166
Figure 20:Listening Mechanism for Region Registries............................ 173
Figure 21:RegionClientAgent Scenario.................................................... 179
Figure 22:Synchronous Migration Scenario............................................. 201
Figure 23:Asynchronous Migration Scenario........................................... 202
Figure 24:External Application Scenario ................................................. 215
Figure 25:CORBA Object Creation and Connection Establishment......... 244
Figure 26:Migration of a CORBA Server Agent...................................... 245
Figure 27:Connection Re-establishment by CORBA Client Agent .......... 246

Figure 28:CORBA Agent Scenario ............................................................ 257

# List of Tables

Table 1: Notational Conventions ........................................................ 3
Table 2: Icons ........................................................................................ 3
Table 3: Filter Keys ............................................................................. 182

# List of Examples

Example 1:HelloAgent ........................................................................ 19
Example 2:PrintStringAgent........................................................................ 27
Example 3:PrintInfoAgent........................................................................ 45
Example 4:BoomerangAgent ........................................................................ 55
Example 5:ActionAgent ........................................................................ 61
Example 6:CopyAgent ........................................................................ 70
Example 7:ServerAgent........................................................................ 84
Example 8:IServerAgent ........................................................................ 84
Example 9:ClientAgent ........................................................................ 85
Example 10:AsyncServerAgent ........................................................................ 98
Example 11:IAsyncServerAgent ........................................................................ 100
Example 12:AsyncServerException ........................................................................ 100
Example 13:AsyncClientAgent ........................................................................ 104
Example 14:DynamicServerAgent ........................................................................ 115
Example 15:IDynamicServerAgent........................................................................ 116
Example 16:TestDataPacket........................................................................ 117
Example 17:DynamicClientAgent........................................................................ 122
Example 18:MulticastServerAgent........................................................................ 131
Example 19:IMulticastServerAgent ........................................................................ 132
Example 20:MulticastClientAgent ........................................................................ 133
Example 21:AgencyClientAgent ........................................................................ 152
Example 22:IListeningAgent........................................................................ 155
Example 23:GHListener ........................................................................ 157
Example 24:RegionClientAgent........................................................................ 175
Example 25:MigratingServerAgent........................................................................ 189
Example 26:IMigratingServerAgent ........................................................................ 191
Example 27:MigratingClientAgent ........................................................................ 194

Example 28:ExternalApplication ............................................................ 207
Example 29:ServerObject..................................................................... 210
Example 30:IServerObject ................................................................... 211
Example 31:ExternalAccessAgent.......................................................... 212
Example 32:IExternalAccessAgent......................................................... 214
Example 33:SleepyAgent ..................................................................... 220
Example 34:PlaceService ..................................................................... 227
Example 35:IPlaceService..................................................................... 228
Example 36:Place Property File ............................................................. 228
Example 37:PlaceAccessAgent.............................................................. 229
Example 38:Keytool Usage: Generate Key............................................. 237
Example 39:Keytool Usage: List Keys ................................................... 238
Example 40:Keytool Usage: Export Key ................................................. 238
Example 41:CI_CORBAServerAgent...................................................... 247
Example 42:CORBAServerAgent........................................................... 248
Example 43:CORBAClientAgent ........................................................... 253

# 1 Preface

This chapter provides information about this document itself as well as about the remaining parts of the Grasshopper manual.

## 1.1 About this Document

This document describes how to implement mobile and stationary agents on top of the Grasshopper platform. Every fundamental implementation aspect is handled, such as mobility, local/remote communication, agent localization, CORBA support, and security.

Simple example agents are introduced and enhanced step by step throughout the whole document, starting as 'minimal agents' and ending as rather complex agents that make use of most of the Grasshopper functionality.

## 1.2 Document Structure

This document is subdivided into the following chapters.

CHAPTER 1: Preface, this part of the document, gives an overview of this manual and its background.

CHAPTER 2: Introduction, provides a general description about running the examples that are described in the scope of this document.

CHAPTER 3: Hello Agent!, provides basic information that enables you to implement a first, simple agent on top of Grasshopper. The supported agent types, their class structure, and their functionality is introduced, and the chapter ends with an example that shows a minimal Grasshopper agent.

CHAPTER 4: Creation and Removal of Agents, explains how to create and remove Grasshopper agents via the platform's programming and user interfaces. Special emphasis lies on the provision of creation parameters to a new agent.

CHAPTER 5: Agent Related Information, describes the set of information that characterizes a Grasshopper agent, such as an agent's identifier, location, name, type, code base, and state.

CHAPTER 6: Move Me!, provides detailed information about the mobility aspect of Grasshopper agents. Special emphasis lies on the introduction of an agent's data state and the structure of an agent's `live()` method.

CHAPTER 7: Action!, describes how a user can trigger a running agent via the platform's user interfaces in order to force the agent to perform a certain action.

CHAPTER 8: Clones and Copies, explains how to create a copy of a Grasshopper agent.

CHAPTER 9: The Communication Service, describes how Grasshopper agents can communicate with each other, with remote agencies, and with registration servers.

CHAPTER 10: The Persistence Service, explains how agents can take advantage of the persistence mechanism of the hosting agency.

CHAPTER 11: Special Places, describes how single places within an agency can be enhanced with additional functionality.

CHAPTER 12: The Security Service, shows the impacts of the security mechanisms of an agency on the execution of agents.

CHAPTER 13: Grasshopper and CORBA, explains how Grasshopper agents can act as CORBA servers and/or clients.

ANNEX A: Acronyms

ANNEX B: Glossary

ANNEX C: Index

## 1.3 Related Documents

The whole Grasshopper manual comprises four parts:

**Basics and Concepts**. This part covers an introduction to mobile agent technology and to the Grasshopper platform.

**User's Guide**. This part describes the platform installation and its usage via graphical and command line interfaces.

**Programmer's Guide**. This part explains how to realize mobile and stationary agents on top of the Grasshopper platform.

**Release Notes**. This part lists modifications and enhancements compared to the previous release of Grasshopper.

## 1.4 Notational Conventions

Several notational conventions are used throughout the whole document in order to improve the readability and to support you in finding specific information.

### 1.4.1 Fonts

The following font types are used within this manual:

| | |
|---|---|
| Proportional font | Used for standard text |
| *Proportional italic font* | Used either to emphasize words or to indicate the first appearance of new terms. |
| `Fixed font` | Used for source code, E-mail addresses and http addresses. |
| **`Fixed, bold font`** | Used to emphasize parts of source code, such as class or method names |
| ***`Fixed bold italic font`*** | Used to emphasize comments inside source code |

Table 1: Notational Conventions

### 1.4.2 Icons

The following icons are placed at the page margins in order to indicate certain types of information:

| | |
|---|---|
| | This icon indicates information that is specific for Unix operating systems. |
| | This icon indicates information that is specific for Windows operating systems. |

Table 2: Icons

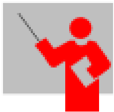
| | |
|---|---|
| | This icon indicates paragraphs that provide some background information about a specific topic. This information is not required for the understanding of the respective section and may be skipped. However, it may be interesting for you if you want to know more about the concepts of Grasshopper Development System. This background information is additionally highlighted by means of a shaded frame. |
| | This icon indicates useful tips and tricks that facilitate the usage of the product. |
| | This icon indicates paragraphs that are of particular importance and that should be read in any case, even if you want to go through the document as soon as possible. |
|  | This icon is used to indicate examples. |

Table 2: Icons

## 1.5 How to Get in Contact

To make suggestions, critics, or even compliments, please contact us by sending an E-mail to `grasshopper@ikv.de`

In order to retrieve the comments of other Grasshopper users and participate in discussions, please visit our Web site

`http://www.grasshopper.de/community`

and subscribe to the discussion groups you are interested in.

Additional information can be retrieved from the following Web site:
`http://www.grasshopper.de`

# 2 Introduction

This document describes how to implement mobile and stationary agents on top of the Grasshopper platform. Every fundamental implementation aspect is handled, such as mobility, local/remote communication, agent localization, CORBA support, and security.

Simple example agents are introduced and enhanced step by step throughout the whole document, starting as 'minimal agents' and ending as rather complex agents that make use of most of the Grasshopper functionality.

### Installation Requirements

We recommend that you take a look at the examples provided in this manual. The following requirements have to be fulfilled in order to run them.

1. The Java Development Kit JDK 1.2 (or higher) as well as the Grasshopper platform V2.2 is needed for all examples. Concerning JDK 1.3, Grasshopper takes advantage of improved reflection mechanisms: With JDK 1.3, it is not anymore required to manually create proxy classes for using the Grasshopper communication service.
2. In order to compile and run examples that are accessing CORBA functionality, a CORBA environment must have been installed. Please refer to the User's Guide to get information about the CORBA environments that are supported by Grasshopper. If your installed CORBA implementation is not initially supported by Grasshopper, please refer to the User's Guide in order to see how to adapt Grasshopper to your CORBA implementation.
3. In order to compile and run the examples that make use of the security features of Grasshopper, a security add-on must have been installed. For detailed information about this add-on, please refer to the User's Guide.

### API Specification

Grasshopper provides a large set of classes and interfaces that can be used by agents for accessing the functionality of the platform. This document focuses on the introduction of the platform functionality in terms of textual descriptions and programming examples. It does not provide an entire specification of the platform API. If a specific method is mentioned in this document, its concrete parameters are usually left out. For instance, the method

```
AgentInfo createAgent (
        java.lang.String className,
        java.lang.String codeBase,
        java.lang.String placeName,
```

```
          java.lang.Object[] arguments)
```

is simply mentioned as

```
createAgent(...)
```

An entire API specification is provided in HTML format as part of your Grasshopper installation (see directory `<GH_HOME>/doc/api`, where `<GH_HOME>` is the root directory of your Grasshopper installation).

**Example Scenarios**

The examples that are introduced in the following chapters are part of the Grasshopper release. Their source code can be found in the directory `<GH_HOME>/examples/src`, and the corresponding class files are stored in the directory `<GH_HOME>/examples/classes` of your Grasshopper installation.

The examples are arranged in Java packages. The root package is named `examples`, and the inner packages correspond to the sections of this document:

- Package `examples.simple`

  This package comprises all examples that consist of single agents:

  - `HelloAgent` (see Example 1 on page 19). This example represents the minimal Grasshopper agent that just prints a variation of the probably most famous example message.
  - `PrintStringAgent` (see Example 2 on page 27). This example shows how to provide creation arguments to an agent and how to remove an agent.
  - `PrintInfoAgent` (see Example 3 on page 45). This example describes the set of information that characterizes an agent, and explains the way how an agent can get access to this information.
  - `BoomerangAgent` (see Example 4 on page 55). This example agent migrates to a user-defined location and returns back to its home location.
  - `ActionAgent` (see Example 5 on page 61). This example shows how a user can trigger an agent via the agency's user interfaces.
  - `CopyAgent` (see Example 6 on page 70). This example agent produces copies of itself inside all running agencies.
  - `AgencyClientAgent` (see Example 21 on page 152). This example shows how an agent gets access to the functionality of local and remote agencies. It needs the following additional classes from the package examples.util: `GHListener`, `IListeningAgent`, `IListenin-`

gAgentP.

- `RegionClientAgent` (see Example 24 on page 175). This example shows how an agent gets access to a Grasshopper agency domain service. It needs the following additional classes from the package examples.util:
  `GHListener`, `IListeningAgent`, `IListeningAgentP`.
- `SleepyAgent` (see Example 33 on page 220): This example shows the usage of the persistence service.

- Package `examples.simpleCom`

  This package comprises the agents belonging to the simple communication scenario as described in Section 9.4 on page 83. The scenario describes the communication basics of the Grasshopper platform. The package consists of the following classes:

  - `ServerAgent` (see Example 7 on page 84)
  - `IServerAgent` (see Example 8 on page 84)
  - `IServerAgentP`. This class has been generated by using the Grasshopper stub generator. The class is needed by the ClientAgent for creating proxies of the ServerAgent, if a JDK 1.2 environment is used. Please refer to Section 9.2 for detailed information about Grasshopper proxy objects and the stub generator.
  - `ClientAgent` (see Example 9 on page 85 )

- Package `examples.asyncCom`

  This package comprises the agents belonging to the asynchronous communication scenario as described in Section 9.6 on page 97. The scenario explains how an agent can use the communication service for asynchronous method invocations and describes different result handling mechanisms. The package consists of the following classes:

  - `AsyncServerAgent` (see Example 10 on page 98)
  - `IAsyncServerAgent` (see Example 11 on page 100)
  - `IAsyncServerAgentP`. This class has been generated by using the Grasshopper stub generator. The class is needed by the AsyncClientAgent for creating proxies of the AsyncServerAgent, if a JDK 1.2 environment is used. Please refer to Section 9.2 for detailed information about Grasshopper proxy objects and the stub generator.
  - `AsyncServerException` (see Example 12 on page 100)
  - `AsyncClientAgent` (see Example 13 on page 104)

- Package `examples.dynamicCom`

  This package comprises the agents belonging to the dynamic communication scenario as described in Section 9.8 on page 114. The scenario shows how a client agent can contact a server agent without having access to the server proxy. The package consists of the following classes:

  - `DynamicServerAgent` (see Example 14 on page 115)
  - `IDynamicServerAgent` (see Example 15 on page 116)
  - `IDynamicServerAgentP`. This class has been generated by using the Grasshopper stub generator. The class is needed by the DynamicClientAgent for creating proxies of the DynamicServerAgent, if a JDK 1.2 environment is used. Please refer to Section 9.2 for detailed information about Grasshopper proxy objects and the stub generator.
  - `TestDataPacket` (see Example 16 on page 117)
  - `DynamicClientAgent` (see Example 17 on page 122)

- Package `examples.multicastCom`

  This package comprises the agents belonging to the multicast communication scenario as described in Section 9.10 on page 131. The scenario shows how a client agent can add a set of server agents to a multicast group and invoke methods on this group. The package consists of the following classes:

  - `MulticastServerAgent` (see Example 18 on page 131)
  - `IMulticastServerAgent` (see Example 19 on page 132)
  - `IMulticastServerAgentP`. This class has been generated by using the Grasshopper stub generator. The class is needed by the MulticastClientAgent for creating proxies of the MulticastServerAgent, if a JDK 1.2 environment is used. Please refer to Section 9.2 for detailed information about Grasshopper proxy objects and the stub generator.
  - `MulticastClientAgent` (see Example 20 on page 133)

- Package `examples.migratingCom`

  This package comprises the agents belonging to the migrating communication scenario as described in Section 9.15 on page 189. The scenario shows how the communication service forwards method calls and results to migrating server and client agents. The package consists of the following classes:

  - `MigratingServerAgent` (see Example 25 on page 189)
  - `IMigratingServerAgent` (see Example 26 on page 191)

  - `IMigratingServerAgentP`. This class has been generated by using the Grasshopper stub generator. The class is needed by the MigratingClientAgent for creating proxies of the MigratingServer-Agent, if a JDK 1.2 environment is used. Please refer to Section 9.2 for detailed information about Grasshopper proxy objects and the stub generator.
  - `MigratingClientAgent` (see Example 27 on page 194)

- Package `examples.externalCom`

  This package comprises the agents belonging to the external communication scenario as described in Section 9.17. The scenario shows how an external application can interact with Grasshopper components (agents, agencies, and region registries) by using the Grasshopper communication service. The external application acts as communication server and client.

- Package `examples.corbaCom`

  This package comprises the agents belonging to the CORBA communication scenario as described in Section 13.1.1. The scenario shows how to implement Grasshopper agents as CORBA server objects and CORBA clients. Particular focus lies on issues concerning the migration of CORBA-enhanced mobile agents.

- Package `examples.util`

  This package comprises classes that are commonly used by different examples agents mentioned above:

  - `GHListener` (see Example 23 on page 157). This class realizes a listener object that is able to monitor the events of an agency or a region registry. It is implemented by the following agents: `examples.simple.AgentSystemClientAgent`, `examples.simple.RegionClientAgent`.
  - `IListeningAgent` (see Example 22 on page 155). This class represents an interface that is contacted by `GHListener` objects in order to forward event notifications to listening agents. It is implemented by the following agents: `examples.simple.AgentSystemyClientAgent`, `examples.simple.RegionClientAgent`.
  - `IListeningAgentP`. This class has been generated by using the Grasshopper stub generator. The class is needed by `GHListener` objects for creating server proxies of the AgencyClientAgent, if a JDK 1.2 environment is used. Please refer to Section 9.2 for detailed information about Grasshopper proxy objects and the stub generator.

### Running the Examples

Each example consists of one or more agents. To run the examples, at least one agency must have been started. Some examples require two agencies and eventually an agency domain service (i.e., a region registry or X.500 directory service). For each example, these requirements are mentioned in this document below the corresponding source code. The corresponding paragraphs or sections are titled „Running the Example“ for examples consisting of a single agent and „Running the Scenario“ for examples consisting of more than one agent. For detailed information about how to start an agent, an agency, or an agency domain service, please refer to the User's Guide.

**Example output**

The example agents display the progress of their execution in terms of textual messages or graphical components. Note that textual messages, initiated by `log(String)` and `log(String, Throwable)` statements, are by default displayed in the terminal window (e.g., XTerm or MS-DOS console window) in which the agency has been created. If the agency's GUI is active, a message console can be activated via the menu item Tools -> Console. In this case, all outputs from `stdout` and `stderr` are printed in this console. The log(...) methods are provided by the class `de.ikv.grasshopper.agency.Agent` (see Chapter 3).

Some of the example agents create a simple GUI. This GUI realizes a *modal dialog* which blocks the whole agency GUI. Note that this is not a Grasshopper-specific problem, but a general characteristic of modal dialogs. The internal execution of the agency is not influenced. That means, all internal agency threads as well as threads of other running agents are not blocked. The only impact is that the agency GUI does not accept any user inputs until the GUI of the agent has been closed.

**Known problem**

In some cases the GUI of an example agent appears behind the agency GUI. In this case, the agent GUI is active but cannot be accessed, since the agency GUI (residing in front of the agent GUI) is blocked by the agent GUI. This problem could be solved by implementing a more advanced agent GUI. However, the intention of the example agents is to focus on a specific set of Grasshopper functionality and not on GUI design. Thus, the graphical components have been realized as simple as possible. *In order to avoid all associated problems, it is recommended that you start the agencies with their textual interface (TUI) instead of their GUI if you want to run an example agent that provides an own GUI.*

**Fault tolerance**

Note that all examples are meant to show a specific set of Grasshopper functionality. In order to focus on the essential parts of the code and to keep the code as simple and short as possible, no effort has been spent in making the example agents stable and fault tolerant. Thus, it may be that an example

crashes if it has not been started exactly in the described way.

**Class loading and caching**

In order to start an agent, the hosting agency must have access to all class files that are required by the agent. There are two possibilities to grant this access: Either the agent's code base must be included in the CLASSPATH environment variable of the underlying operating system, or the code base must be explicitly typed in when you start an agent. Note that the above choice influences the *class caching mechanism* of the Java virtual machine (JVM) on which the agency is running:

- If the agent's code base is included in the CLASSPATH environment variable, the agent's classes are loaded only once during the entire life time of the JVM (which is equal to the life time of the agency running on the JVM). If more than one instance of the same agent is started on one agency, the JVM uses its internally cached classes for the creation of the second and all subsequent instances.
- If the agent's code base is not included in the CLASSPATH environment variable, you have to provide it when starting a new agent. (For this purpose, a Grasshopper agency offers adequate graphical and textual input facilities whose usage is explained in the User's Guide.) In this case, the agency does not cache the agent code. If more than one instance of the same agent is started on one agency, the JVM loads the required code from the specified code base for each new agent.

The explanations above may be of interest for the development of agents:

During the development phase of a new agent, it may be desirable *not* to include its code base in the CLASSPATH environment variable. In this case it is possible to create an instance of the new agent, to modify the agent's code afterwards, and to create another instance of the agent inside the same agency by using the modified code *without re-starting the agency*. This may be advantageous especially if more than one agency is involved in a scenario, since the re-start of several agencies (which eventually run on different hosts) is rather time consuming.

After sufficiently testing an agent, it may be desirable to include its code base in the CLASSPATH environment variable. In this case, it is not necessary for you to explicitly type in the code base when creating the agent, since the agency's JVM by default accesses the system's Java classpath.

# 3 Hello Agent!

From a static point of view, a Grasshopper agent is realized by means of a Java class or a set of classes. Each agent has exactly one *agent class* which characterizes the agent and which must be derived from one of the superclasses `MobileAgent`, `StationaryAgent`, `PersistentMobileAgent`, or `PersistentStationaryAgent`, all provided by the Grasshopper platform.

**Stationary vs. mobile agents**

When implementing a Grasshopper agent, the first decision that must be made is whether the agent is to be stationary or mobile. Mobile agents are able to migrate autonomously from one agency to another, whereas stationary agents do not have this ability. Besides, it is not possible for a user to move a stationary agent via the graphical or textual user interfaces (UI) of the hosting agency. The reason for the separation between mobile and stationary agents is that the migration of specific agents may cause failures.

For instance, consider an agent that has references or native access to local resources. In this case, the agent may only be able to run on a specific agency where the required resources are available. If the agent is moved to another location where the required environmental conditions are not provided, the agent is not able to run properly. In order to avoid this situation, such an agent should be stationary.

Usually, Grasshopper mobile agents are derived from the class `de.ikv.grasshopper.agent.MobileAgent`, while stationary agents are derived from `de.ikv.grasshopper.agent.StationaryAgent`. Special cases are *persistent agents* which are mentioned in the following paragraph.

**Persistent vs. non-persistent agents**

The second decision when implementing a Grasshopper agent is whether the agent shall be recoverable after a system crash. Grasshopper provides a persistence service in order to enable agencies to persistently store an agent's data state within the local file system. In case of a system crash or simply the termination of an agency, all persistently stored agents can be recovered when the agency is restarted. In order to achieve this, two preconditions must be fulfilled:

1. The agency must have been started with activated persistence service. Please refer to the User's Guide for detailed information about how to start an agency.
2. The agents that are to be recoverable must have been implemented with enabled persistence. This is achieved by deriving the agent class from one

of the superclasses `de.ikv.grasshopper.agent.Persistent-MobileAgent` or `de.ikv.grasshopper.agent.Persistent-StationaryAgent`.

Note that the persistence service stores all persistence-supporting agents (i.e., agents derived from one of the classes `PersistentMobileAgent` or `PersistentStationaryAgent`). The more persistence-supporting agents are running, the more processing power is required by the hosting agency for maintaining their internal information. Even an agency without any running agents is slower if the persistence service is active. This fact should be considered for evaluating if an agent needs persistence-support or not. Further information about the persistence service can be found in Chapter 10.

Summarizing, Grasshopper supports four kinds of agents:

**Grasshopper agent types**

- *Mobile agents*: instances of classes that are derived from `de.ikv.grasshopper.agent.MobileAgent`
- *Stationary agents*: instances of classes that are derived from `de.ikv.grasshopper.agent.StationaryAgent`
- *Persistent mobile agents:* instances of classes that are derived from `de.ikv.grasshopper.agent.PersistentMobileAgent`. In contrast to usual mobile agents, persistent mobile agents are able to use the Grasshopper persistence service (see Chapter 10).
- *Persistent stationary agents:* instances of classes that are derived from `de.ikv.grasshopper.agent.PersistentStationary-Agent`. In contrast to usual stationary agents, persistent stationary agents are able to use the Grasshopper persistence service (see Chapter 10).

As shown in Figure 1, all Grasshopper agents have the common superclass `de.ikv.grasshopper.agency.Agent`.

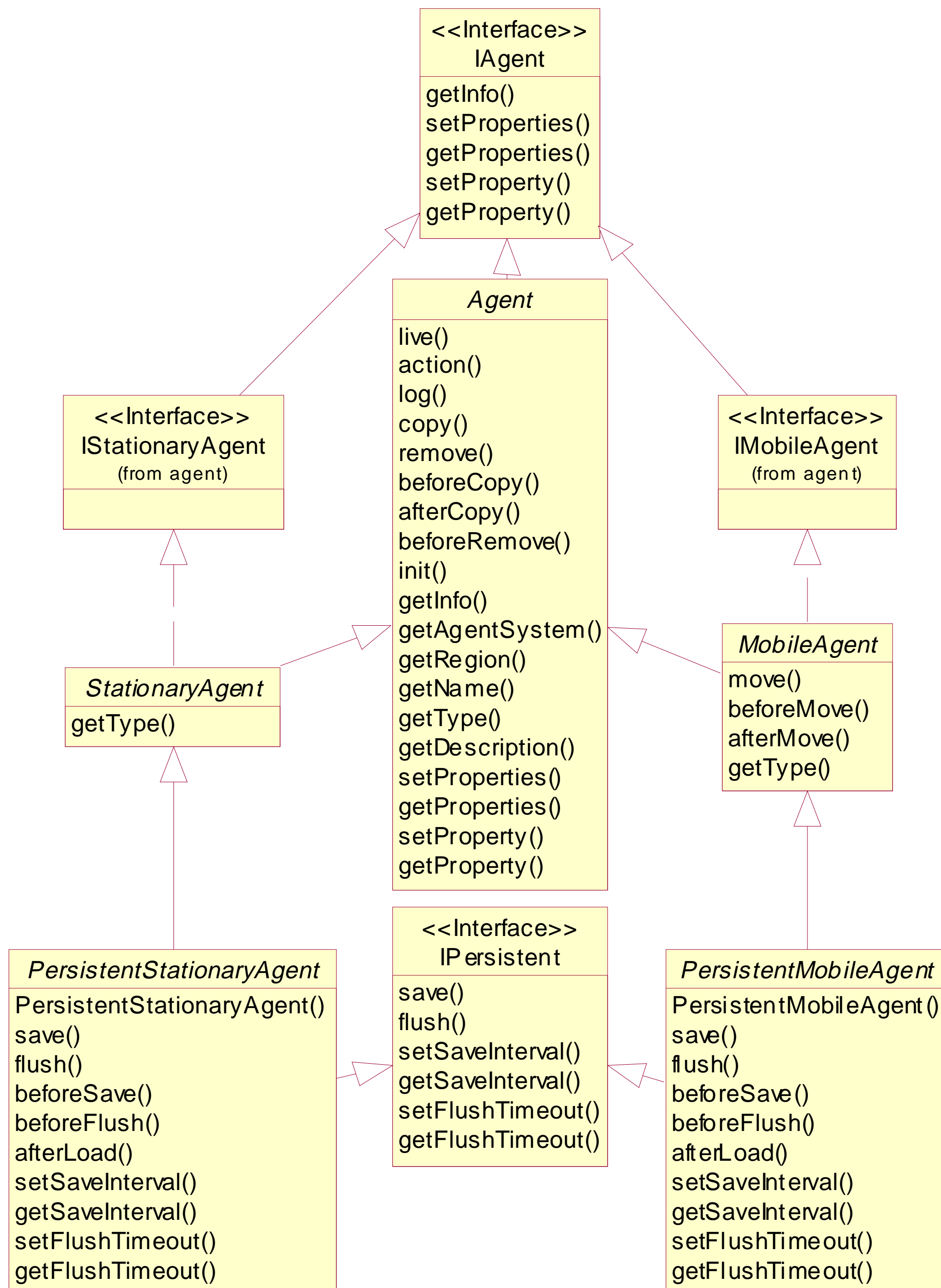



Figure 1: Agent Class Diagram

**class Agent**

Via its superclass `Agent`, each Grasshopper agent has access to the following methods:

- `action()`: This method is automatically invoked by the agency if a user performs a double-click on the corresponding agent entry in the agency GUI or types in the 'invoke' command in the TUI. For example, this

method can be used to activate an agent's own GUI on request of a user. Please refer to Chapter 7 for more information.

- `afterCopy(...)`: This method is automatically invoked by the hosting agency on the copy of an agent right after its creation. The method may be overridden by agent programmers when implementing an agent class. Please refer to Chapter 8 for more information.
- `beforeCopy(...)`: This method is automatically invoked by the hosting agency before an agent is copied and may be overridden by agent programmers when implementing an agent class. Its purpose is to enable the agent to prepare its copying, e.g., by performing specific initializations. By throwing a `VetoException` inside the `beforeCopy()` method, an agent may even prohibit its copying. Please refer to Chapter 8 for detailed information.
- `beforeRemove(...)`: This method is automatically invoked before an agent is removed. Its purpose is to enable the agent to prepare its removal, e.g., by releasing resources and references. Please refer to Section 4.2 for detailed information.
- `copy(...)`: This method enables an agent to create a copy of itself. Detailed information about its usage is provided in Chapter 8.
- `getAgentSystem(...)`: This method enables the agent to access the functionality of the local agency by returning the interface `IAgentSystem`. For detailed information about the concept of Grasshopper proxy objects, please refer to Chapter 9. Please refer to Section 9.11 in order to learn about an agency's API and about how an agent can access this API.
- `getDescription()`: This method returns the textual description of the agent which can be specified by the agent programmer during the implementation phase or by the user when creating the agent, provided that this is supported by the agent implementation (see Section 5.2). The purpose of the description is to provide information about the agent's capabilities to the user.
- `getInfo(...)`: This method returns a set of information that is associated with the agent. Among others, this set of information comprises the agent's identifier, type, and name. For detailed information, please refer to Chapter 5.
- `getName()`: This method returns the name of the agent. In contrast to the unique agent identifier which is automatically generated by an agency during the creation of an agent, the name can be specified by the agent programmer during the implementation phase or by the user when creating the agent, provided that this is supported by the agent implementation (see

Section 5.2).

- getPlace(): As explained in Chapter 11, an agency user can allocate additional functionality to specific places. With this method, an agent can get access to the functionality of the place in which the agent is currently running.
- `getProperties()`: Optionally, a set of properties can be provided to an agent, either during the agent's creation, or during its runtime. This method returns the complete list of an agent's properties. Please refer to Chapter 5 for further information about agent properties.
- `getProperty(...)`: Optionally, a set of properties can be provided to an agent, either during the agent's creation, or during its runtime. This method returns the value of one single property, specified by the property key. Please refer to Chapter 5 for further information about agent properties.
- `getRegion(...)`: This method enables the agent to access the functionality of an agency domain service by returning the interface `IRegion`. For detailed information about the concept of Grasshopper proxy objects, please refer to Chapter 9. Please refer to Section 9.12 in order to learn about an agency domain service's API and about how an agent can access this API.
- `getType(...)`: This method returns the type of the agent. Please refer to Chapter 5 for detailed information.
- `init(...)`: This method is automatically called by the hosting agency when an agent is created. It offers the possibility to provide creation arguments to the agent. For detailed information, please refer to Chapter 4.
- `live(...)`: This is the most fundamental method of each Grasshopper agent, since its implementation realizes the agent's active, autonomous behavior. Note that this is the only method that *must* be overridden by an agent programmer when implementing an agent class. A first implementation of this method is provided by the example in Section 3.1. A detailed guideline for the design of the `live(...)` method for mobile agents is given in Chapter 6.
- `log(...)`: Two `log(...)` methods are provided by the superclass `Agent` in order to enable an agent to print textual messages onto the text console of the local agency. The method `log(String)` directs outputs to `stdout`, while the method `log(String, Throwable)` directs outputs to `stderr`.
- `remove()`: This method allows an agent to remove itself. Please refer to

Section 4.2 for detailed information.

- `setProperties()`: Optionally, a set of properties can be provided to an agent, either during the agent's creation, or during its runtime. This method sets the complete list of an agent's properties. Please refer to Chapter 5 for further information about agent properties.
- `setProperty(...)`: Optionally, a set of properties can be provided to an agent, either during the agent's creation, or during its runtime. This method sets the value of one single property, specified by the property key. Please refer to Chapter 5 for further information about agent properties.

**class MobileAgent**

Via its superclass `MobileAgent`, each Grasshopper mobile agent has access to the following methods:

- `afterMove(...)`: This method is automatically invoked by the hosting agency after an agent has arrived in a new agency after a migration procedure. The method may be overridden by agent programmers when implementing an agent class. For instance, the agent may want to adapt itself to the new environment, e.g., by allocating new references or resources. Please refer to Section 6.2 for more information.
- `beforeMove(...)`: This method is automatically invoked by the hosting agency before an agent is moved. This method is of particular interest, if the agent's migration is not triggered by the agent itself, but by some other software component. In this case, the agent may want to prepare its migration, e.g., by releasing references or resources. By throwing a `VetoException` inside the `beforeMove()` method, an agent may even prohibit its migration. Please refer to Section 6.2 for more information.
- `getType(...)`: This method returns the type of the agent. Please refer to Chapter 5 for detailed information.
- `move(...)`: Via this method, an agent is able to migrate to another agency/place. Please refer to Chapter 6 for more information.

**class StationaryAgent**

Via its superclass `StationaryAgent`, each Grasshopper stationary agent has access to the following methods:

- `getType(...)`: This method returns the type of the agent. Please refer to Chapter 5 for detailed information.

**Persistent agents**

For information about the classes `PersistentMobileAgent` and `PersistentStationaryAgent`, please refer to Chapter 10 where the persistence service is described.

**Remotely accessible agent methods**

Beside the classes mentioned above, Figure 1 shows four Java interfaces: `IAgent`, `IMobileAgent`, `IStationaryAgent`, and `IPersistent`. These interfaces cover those methods that are accessible locally and remotely by other software components, whereas the remaining class methods are only accessible for the derived agent class itself. A detailed explanation of the external access of agent methods (which is realized with the Grasshopper communication service and proxy objects) is given in Chapter 9.

**Agents as threads**

By definition, a software agent is an active, autonomously acting component. In Grasshopper, this fundamental characteristic is released by defining an agent as Java thread. The entire thread handling is performed by the hosting agency (i.e., the Java process in which multiple agents may run concurrently).

Usually, the active behavior of a Java thread is specified inside the thread's `run()` method which is declared in the `java.lang.Runnable` interface.

Concerning Grasshopper, the `run()` method of the agent threads is declared final inside the superclass `Agent`. The reason is that the agency has to perform several checks and operations before an agent is allowed to start its actual task.

**live()**

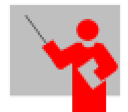
Instead of the original `run()` method, each Grasshopper agent has to implement a method named `live()`. This method (which is declared abstract in the agent's superclass) defines the active behavior of the agent, i.e., the control flow that is performed inside the agent's own thread. The `live()` method is the only method that is mandatory for each agent.

## 3.1 Example: HelloAgent

With the knowledge that has been provided so far, you are able to implement your first Grasshopper agent:

The HelloAgent has a rather short life which ends right after a single print statement. After performing this statement, the `live()` method as well as the agent thread's `run()` method terminate.

Example 1: HelloAgent



```
package examples.simple;

import de.ikv.grasshopper.agent.MobileAgent;

public class HelloAgent extends MobileAgent
  {
  public void live() {
```

```
      log("Hello Agent!");
    }
  }
```

Requirements:

- One running agency

Running the example:

Create the HelloAgent inside the running agency via the agency's UI and have a look at the agency's console window.

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.simple.HelloAgent
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

Once the `live()` method of an agent has terminated, also the agent's thread terminates, and the agent remains inside the hosting agency just as a passive Java object. The methods of the agent can still be accessed by other objects/ agents, but the agent itself is not active anymore. The only possibility to reanimate the agent is to move it to another agency. In this case, the agent's thread is re-started at its new location.

In order to keep the agent living, the termination of the `live()` method must be avoided. This can be achieved for example via endless loops or `wait` statements. However, in some cases it may be desired that an agent migrates actively to another agency and after this just acts as a passive server object for other entities. In this case, it is advisable to terminate the `live()` method in order to save resources.

## 3.2 Summary

- A Grasshopper agent is statically represented by a Java class that is either derived from one of the classes `MobileAgent`, `StationaryAgent`, `PersistentMobileAgent`, or `PersistentStationaryAgent`.
- A Grasshopper agent is realized as Java thread. The `run()` method of the agent's thread is not accessible for agent programmers. The `live()` method has to be used instead.
- Each Grasshopper agent has to implement the `live()` method whose

control flow is performed inside the agent's thread and thus defines the agent's active behavior.

- When the `live()` method terminates, the agent is not removed, but remains in the hosting agency as passive object.

# 4 Creation and Removal of Agents

This section describes how to create and remove a Grasshopper agent. Concerning the creation, special emphasis is given to the provision of creation parameters.

## 4.1 Agent Creation

The first method that runs when a usual Java object is created is the object's constructor. Beside the default constructor, it is possible to specify additional constructors in order to deliver initial parameters to the object.

Also a Grasshopper agent may require initial input parameters. However, in contrast to a usual Java object, these arguments cannot be delivered via a specific constructor. In order to explain the reason for this, the agent creation process has to be described in more detail:

**Agent creation**

Grasshopper agents are generally created inside an agency. A Grasshopper agency supports two different ways for creating agents: a user interface and a programming interface:

**...via the UI**

Human users can create agents via the graphical or the textual user interface (UI) of an agency. (Please refer to the User's Guide for detailed information about the GUI and TUI usage.)

**... via the API**

Software components, such as other agents/objects, can create an agent via the application programming interface (API) of an agency. *In contrast to the creation of usual Java objects, this cannot be achieved properly via a* `'new'` *statement(!)*, such as `'HelloAgent hAgent = new HelloAgent()'`.

**new vs. createAgent()**

Instead, an agency provides the method `createAgent(...)` that is offered by the interface `de.ikv.grasshopper.agency.IAgentSystem` and that has to be used for creating agents via the API.

The reason for this way of agent creation (which may seem to be a bit strange for Java programmers) is that an agency has to perform several internal tasks during the creation procedure:

- The agency registers the agent in the agency's internal database.
- The agency registers the agent inside the agency domain service, if this component is available.
- The agency enables the agent to access the agency's functionality by delivering fundamental object references.

- The agency creates important information objects that are containing data associated with the agent and that are required for management purposes.
- The agency initiates the agent's thread handling, among others by starting the agent's thread inside an own thread group. Reason: Additionally to its initial thread, an agent may create further threads. An own thread group is created for each agent in order to ensure that all threads created by the agent terminate when the agent is removed.

The execution of all these procedures is triggered by the invocation of the `createAgent(...)` method. By using a simple `'new'` statement instead, the agent is not able to run.

**createAgent(...) usage**

The `createAgent(...)` method requires the following parameters:

- the name of the agent class, i.e., the Java class that contains the `live()` method of the agent.
- the code base from which the agent classes can be retrieved (see Section 5.3). Note that this value can be initialized with an empty string if the agent classes are maintained in the Java classpath of the local agency.
- the name of the place in which the agent shall be created. If this value is initialized with an empty string, the agent is created in the default place InformationDesk which exists by default in every Grasshopper agency.
- a set of creation arguments, represented as Object[]. This array is used as parameter of the agent's `init(...)` method which is explained below.
- optionally, a set of properties. These properties are stored in the agent's AgentInfo structure and can be read and modified during the agent's runtime. Note that properties can only be provided to an agent if the agent is created via the agency's API. The agency's UI does not offer this possibility.

**About agent constructors**

When creating a new agent, an agency always invokes the agent's *default* constructor, i.e., a constructor without arguments. Thus, any additional constructor that may have been defined by the agent programmer will never be invoked. If creation arguments are to be provided to the agent, the `init(Object[])` method has to be used instead of a constructor.

**init(...)**

Tasks that have to be performed by the agent only once, i.e., right after the agent's initial creation, should be implemented inside the agent's `init(Object[])` method. This method is defined in the agent's superclass `Agent` and may be overridden by any agent subclass. It is automatically invoked by the agency right after creating the agent and can be considered as substitute of the agent's constructor.

**Delivering creation parameters to an agent...**

The `init(Object[])` method can be used by the agent to receive and analyze initial parameters. In this way, this method can be handled by agent programmers as replacement of a specific constructor. The only thing that has to be considered by the programmer is that the `Object[]` arguments have to be converted to appropriate types inside the `init` method.

**...via the API**

When an agent is created by a software component via the `createAgent(...)` method of the agency's API, the `Object[]` arguments are provided as one parameter of this method. If no parameters are to be provided, the `Object[]` value must be set to `null` inside the parameter list of the `createAgent(...)` method.

**...via the UI**

When an agent is created by a user, the creation parameters are typed in either via a graphical dialog window when using the GUI, or via a command line when using the TUI. If more than one parameter is provided, a blank character has to be used as separator. If one single parameter contains a blank character, the complete parameter has to be included in quotation marks. (Please refer to the User's Guide for detailed information).

Note that, when an agent is created by a human user, the agency always invokes the agent's `init(Object[])` method by using `String[]` as parameter type! Thus, whenever an agent is implemented that requires input parameters and that is to be created via the UI, the `init(Object[])` method has to expect `String[]` as parameter type and perform the cast to actually needed types internally.

Never invoke the agent's `move(...)`, `copy(...)`, or `remove(...)` method inside the `init(...)` method. This may lead to an unpredictable behavior.

## 4.2 Agent Removal

Three possibilities exist for the removal of an agent:

- The agent can remove itself by invoking the method `remove()` of its superclass `Agent`.
- The agent can be removed by external software entities via the API of the agency that is hosting the agent (method `removeAgent(...)` of the agency proxy `IAgentSystem`). Please refer to Section 9.11 where the functionality of agencies is explained.
- The agent can be removed by a user via the agency's UI. Please refer to the User's Guide for information about the usage of an agency.

A Grasshopper agent is realized as a Java thread that runs inside its own thread group. That means, if an agent itself creates several threads, they are also running in the agent's thread group. Since the removal of an agent includes the removal of its thread group, all threads that have been created by the agent are also removed.

**before Remove()**

After receiving the request to remove an agent, the hosting agency automatically invokes the agent's method `beforeRemove()` which is defined in the superclass `Agent`. An agent programmer may override this method in order to enable the agent to prepare its removal, e.g., by releasing occupied references. This is of particular importance if the agent is removed by other entities instead of initiating its removal by itself.

## 4.3 Example: PrintStringAgent

The agent in the following example expects two creation arguments: a string to be printed, and an integer number specifying how often the string has to be printed. These parameters are provided to the agent by the hosting agency via the agent's `init(Object[])` method.

**init(...)**

Note that the parameters are to be provided via the agency's UI. Since the user interfaces interpret every input as `String`, the agent has to handle the `Object[]` arguments of the `init(...)` method as `String[]` (see the conversion `...(String)creationArgs...` in the example code). Besides, the second parameter has to be converted to `int`.

After converting the creation arguments to the appropriate data types, the agent checks whether it has a property with the key „generation". Initially, this is not the case, and thus the agent adds this property with the value „first".

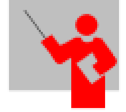
**live()**

If the „generation" property has the value „first" (which is the case if you have created the agent via the agency's UI), the PrintInfoAgent creates a new instance of itself, using the `createAgent(...)` method that has been explained in Section 4.1. As creation arguments (see the fourth parameter of the method), the agent specifies its own creation arguments, i.e., the string to be printed and the number of repetitions. Additionally, the PrintInfoAgent provides a „generation" property with value „second" to the `createAgent(...)` method (see fifth parameter). This property value causes the newly created agent not to create further agents.

After printing the string, the agent removes itself. Note that invoking the `remove()` method causes the agency to automatically invoke the method `beforeRemove()`. The purpose of this method is to enable the agent to

prepare its removal, e.g., in order to release occupied resources. This is of particular importance if the agent is removed by other entities instead of initiating its removal by itself.

Example 2: PrintStringAgent



```
package examples.simple;

import de.ikv.grasshopper.agent.MobileAgent;
import de.ikv.grasshopper.agency.*;

// This class realizes an agent that prints a
// user-defined string.
// Its purpose is to show how creation arguments can
// be provided to an agent.
// After printing the user-defined string,
// the agent removes itself.
public class PrintStringAgent extends MobileAgent
{
  String printThis;
  int max;

  // Creation argument:
  //    args[0] = User-defined string
  //    args[1] = Number of print repetitions
  public void init(Object[] creationArgs) {
    if (creationArgs == null ||
        creationArgs.length < 2) {
      log("Creation arguments needed: \\
      <string> <number>");
      log("Exiting.");
      throw new RuntimeException();
    }
    // Parameters are provided via GUI as Strings.
    // Thus, (String) casting is required.
    printThis = new String((String)creationArgs[0]);
    max = Integer.parseInt((String)creationArgs[1]);

    // define a new property
    if (getProperty("generation") == null)
      setProperty("generation", "first");
  }

  public void beforeRemove() {
    log("(" + getProperty("generation") +
    "): Removing...");
  }
```

```
  public void live() {
    for (int count = 0; count < max; count++)
      log("(" + getProperty("generation") + "): " +
      printThis);

    if (getProperty("generation").equals("first")) {
      java.util.Properties newProps =
        new java.util.Properties();
      // Initialize properties for new agent
      newProps.setProperty("generation", "second");
      Object newArgs[] = new Object[2];
      // Initialize creation arguments for new agent
      newArgs[0] = printThis;
      newArgs[1] = new Integer(max).toString();
      try {
        // Create new agent
        log("(" + getProperty("generation") +
        "): Creating new agent.");
        getAgentSystem().createAgent(
          "examples.simple.PrintStringAgent",
          getInfo().getCodebase(),
          "",
          newArgs,
          newProps);
      }
      catch (AgentCreationFailedException e) {
        log("(" + getProperty("generation") +
        "): Could not create new agent. ", e);
      }
    }
    try {
      remove();
    }
    catch (Exception e) {
      log("(" + getProperty("generation") +
      "): Removal failed. ", e);
    }
  }
}
```

Requirements:

- One running agency

Running the example:

Create the PrintStringAgent inside the running agency via the agency's UI.

Type in the string to be printed and the number of repetitions as creation arguments.

If you are using the textual user interface of the agency, please create the agent by means of the following command (which is meant only as an example, concerning the creation arguments):

```
cr a examples.simple.PrintStringAgent „Hello User!" 7
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

## 4.4 Summary

- A Grasshopper agent is always created inside an agency.
- Using the agency's API, a Grasshopper agent must be created via the method `createAgent(...)` which is part of the agency's API. Never directly use a `new` statement for creating an agent!
- An agency uses the agent's default constructor for the creation. This constructor is also invoked after each migration and copying procedure of the agent. In contrast to this, the agent's `init(Object[])` method is performed only once.
- The agent's `init(Object[])` method, provided by the superclass `Agent`, can be overridden by agent programmers. This method is automatically invoked by an agency when creating a new agent. Via this method, initial creation parameters can be delivered to the agent.
- The creation of an agent can be triggered in two different ways: by human users via the agency's UI and by software entities via the `createAgent(...)` method of the agency's API. Both possibilities allow the provision of creation parameters.
- When an agent is created via the agency's UI, all provided creation parameters are transferred to the agent in terms of a `String` array. Thus, the agent has to convert the string values to data types that are actually required.
- Never invoke the agent's `move(...)`, `copy(...)`, or `remove(...)` method inside the `init(...)` method. This may lead to an unpredictable behavior.

# 5 Agent Related Information

Every Grasshopper agent carries information about itself that may be accessed by itself or by other entities. This information is maintained by an instance of the class `de.ikv.grasshopper.type.AgentInfo`. When a new agent is created, the creating agency initializes a new `AgentInfo` instance and transfers it to the agent. The agent may access this instance by invoking the method `getInfo()` that is implemented in its superclass `Agent`.

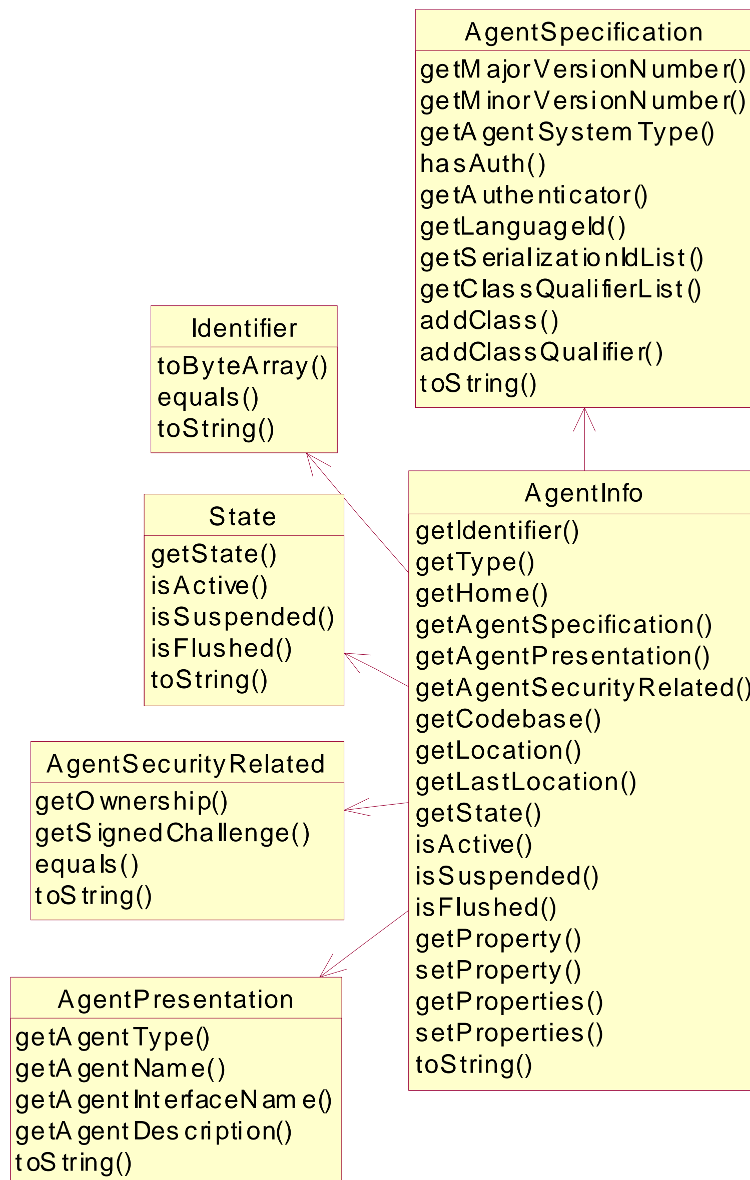



Figure 2: AgentInfo Class Diagram

**AgentInfo**

Note that the `AgentInfo` object is part of the *data state* of every Grasshop-

per agent. That means, a mobile agent carries the content of this object with it when the agent migrates to a new location. Please refer to Section 6.3 in order to learn about the data state.

The AgentInfo class covers the following components:

- *Code base:* This component maintains the code base of the agent. The code base can be provided as http address in the form http://<domain-name>/<path> or as file address in the form file:/<drive>/<directory>. Detailed information about supported code base is given in Section 5.3. (Java type: java.lang.String)
- *Home location:* The Grasshopper address of the agency in which the agent has been created. The home address has the following form: <protocol>://<hostName>:portNumber/<agencyName>. A place name is not included. Detailed information about addressing agencies and agents can be found in Section 5.4. (Java type: de.ikv.grasshopper.communication.GrasshopperAddress)
- *Identifier:* Each Grasshopper agent has its own unique identifier that is generated by the hosting agency during the agent's creation. Detailed information about the identification of Grasshopper agents and agencies is given in Section 5.1. (Java type: de.ikv.grasshopper.type.Identifier)
- *Last location:* The Grasshopper address of the agency that the agent has visited right before the current one. Note that this information is only valid for mobile agents, since stationary agents are not able to change their location. The last location has the following form: <protocol>://<hostName>:portNumber/<agencyName>/<placeName>. Detailed information about addressing agents can be found in Section 5.4. (Java type: de.ikv.grasshopper.communication.GrasshopperAddress)
- *Location:* The Grasshopper address of the agency in which the agent is currently residing. The location information has the following form: <protocol>://<hostName>:portNumber/<agencyName>/<placeName>. Detailed information about addressing agents can be found in Section 5.4. (Java type: de.ikv.grasshopper.communication.GrasshopperAddress)
- *Properties:* An agent can maintain a set of properties, e.g., in order to provide information about its individual characteristics and capabilities. The properties field can be initialized with any set of desired key/value pairs. If a new property is specified whose key already exists in the agent's properties field, the old value is replaced with the new one. Note that an initial set

of properties can be provided to an agent via the `createAgent(...)` method, as explained in Chapter 4. Example 2 in Section 4.3 and Example 6 in Section 8.1 show a possible usage of the properties. (Java type: `java.util.Properties`)

- *Agent presentation:* This component comprises descriptive information about an agent. The class holds the following information: the agent's name, type, textual description, and interface name. (Java type: `de.ikv.grasshopper.type.AgentPresentation`)
  - *Agent name:* Since an agent identifier is a bit uncomfortable to read and interpret for human users, a programmer can define an individual name for an agent that may refer to the agent's characteristics. Note that, in contrast to the identifier, there is no guarantee for the agent name to be unique in the entire Grasshopper environment. Please have a look at Section 5.2 for information about how to set an agent name. (Java type: `java.lang.String`)
  - *Agent type:* This component specifies the type of the agent. The following agent types are defined as constants in the class `de.ikv.grasshopper.util.GrasshopperConstants`: `StationaryAgentType`, `MobileAgentType`.
  - *Textual description:* In the first place, this component is meant for human users in order to get information about the capabilities of the corresponding agent. Please have a look at Section 5.2 for information about how to set an agent description. (Java type: `java.lang.String`)
  - *Interface name:* This component maintains the full qualified name of the agent class, i.e., the class name prefixed with the complete package structure. (Java type: `java.lang.String`). The agent class is the class implementing the agent's `live()` method.
- *Agent security related information:* This component maintains a certificate which defines the owner of a specific agent. Detailed information about this class can be found in Chapter 12 which describes the security features of Grasshopper. (Java type: `de.ikv.grasshopper.type.AgentSecurityRelated`)
- *Agent specification:* This class is just defined in order to support the Mobile Agent System Interoperability Facility (MASIF) standard of the Object Management Group (OMG)[1]. All comprised components have

1. The MASIF specification is available for download on the OMG FTP server. ftp://ftp.omg.org/pub/docs/orbos/. Please look for the ORBOS document with the number 97-10-05.

been developed in the context of MASIF. For detailed information, please refer to the MASIF specification. (Java type: `de.ikv.grasshopper.type.AgentSpecification`)

Note that, in contrast to previous releases of Grasshopper, the current release does not inherently support MASIF. Instead, a MASIF add-on can be downloaded from the IKV web site in order to enhance Grasshopper with MASIF functionality.

- *Class qualifier list:* A class qualifier allows the unique identification of a class. Apart from the class name itself, additional information, such as its version, can be provided by means of the discriminator. For detailed information, please refer to the MASIF specification. (Java type: `de.ikv.grasshopper.type.ClassQualifier[]`)
- *Agent system type:* The type of the MASIF-compliant agent system that is able to create the agent. For detailed information, please refer to the MASIF specification. (Java type: `short`)
- *Authenticator:* This component specifies the authenticator of the agent. For detailed information, please refer to the MASIF specification. (Java type: `short`)
- *Language identifier:* This component contains the identifier of the programming language in which the agent has been implemented. For detailed information, please refer to the MASIF specification. (Java type: `short`)
- *Major version:* Major version of the agent system that is able to create the agent. For detailed information, please refer to the MASIF specification. (Java type: `short`)
- *Minor version:* Minor version of the agent system that is able to create the agent. For detailed information, please refer to the MASIF specification. (Java type: `short`)
- *Serialization identifier list:* This component specifies the kinds of serialization that can be applied to serialize the agent. For detailed information, please refer to the MASIF specification. (Java type: `short[]`)

- *State:* The following states are defined for Grasshopper agents: active, suspended, and flushed. During its life time, an agent's state may change numerous times. Please refer to Section 5.5 for more information about states and life cycles of agents. (Java type: `de.ikv.grasshopper.type.State`).

The `AgentInfo` structure is created and initialized during the creation of the corresponding agent. Some components of `AgentInfo`, such as the agent's code base and class name, are provided by the agent creator via the agency's UI or as parameters of the agency's `createAgent(...)` method. Other components, such as the agent identifier, are automatically generated by the agency that creates the agent. A third set of components can be initialized inside the agent's `init(...)` method. These components are the agent's name and its textual description. Please refer to Section 5.2 for information about how to initialize these components.

## 5.1 Identification

A fundamental management requirement of each agent platform is to enable the unique identification of its distributed components (i.e., agencies and agents). The identifier of an agent is generated by the hosting agency during the agent's creation and afterwards maintained inside the superclass `Agent`.

**Identifier structure**

A Grasshopper identifier consists of the following components:

- a prefix, describing the kind of component that is associated with the identifier. The prefix has one of the following values:
  - *Agent*: This value indicates that the identifier belongs to a mobile or stationary agent.
  - *AgentSystem*: This value indicates that the identifier belongs to an agency.
  - *Listener*: This value is associated with listeners. Note that this prefix is reserved for internal usage only.
- the Internet address of the host on which the identifier has been created
- the date on which the identifier has been created: "yyyy-mm-dd"
- the time on which the identifier has been created: "hh-mm-ss-msms"
- the number of copies of the corresponding agent

A Grasshopper identifier is maintained by an instance of the class `de.ikv.grasshopper.type.Identifier`. Converted into its string representation, an identifier has the following form:

<prefix>#<ip-address>#<date>#<time>#<copy-number>

Example of an agent identifier:

"Agent#123.456.789.012#1999-11-19#15:59:59:0#0"

The first copy of the second copy of the original agent has the following identifier:

"Agent#123.456.789.012#1999-11-19#15:59:59:0#0.2.1"

A comparison between the two example identifiers shows that the only difference is the copy number. That means, the copy of an agent gets the same identifier as the original agent, suffixed by a new copy number. Detailed information about copying agents is given in Chapter 8.

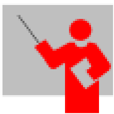
## 5.2 Names and Descriptions

As explained above, most of an agent's information is either generated automatically by the creating agency (e.g., the identifier) or provided to the agency by an external entity (human user or software component) via the agency's UI or API (e.g., the code base).

In contrast to this, the agent name and textual description may be defined by the agent itself. This is achieved by overriding the methods `getName()` and `getDescription()`, respectively.

The methods `getName()` and `getDescription()` can be overridden by agent programmers. These methods return an agent name or textual description, respectively, in form of a Java string. The purpose of both components is to provide information about the characteristics of a specific agent to human users. If the methods are not overridden, they both return the default string „Grasshopper Agent“.

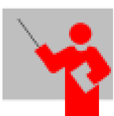
The simplest way to provide an individual name and description is to override the methods with those returning a *constant string value*.



```
public String getName() {
  return „MyAgentName“;
}
```

Concerning this example, it is obvious that all instances of the corresponding agent class have the same name.

If an agent programmer wants to be able to provide different names to different instances of one agent class, the name can be provided as parameter of the agent's `init(...)` method. Inside the `init(...)` method, an instance variable of the agent class has to be initialized with this parameter, and this variable has to be used as return value of the agent's `getName()` method.



```
public class Agent extends MobileAgent {
  String agentName;
  public void init(Object[] creationArgs) {
```

```
      agentName = (String)creationArgs[0];
    }
    public String getName() {
      return agentName;
    }
    ...
  }
```

Note that the initialization of the name has to be performed inside the `init(...)` method if the name is to be maintained by the `AgentInfo` structure. The reason is that only the agency which creates the agent is able to initialize the `AgentInfo` structure. After running the `init(...)` method, the creating agency reads the agent name by invoking the `getName()` method and writes this name into the corresponding component of the `AgentInfo` structure. Any further modifications of the name variable will not be stored within the `AgentInfo` structure. The `AgentInfo` structure is used for registering the agent inside the local agency and the agency domain service. An agent's name can be used as search key in order to enable entities to look for a specific agent. If the searching entity expects another name than the one maintained inside the `AgentInfo` object, the searched agent will not be found. *The name that is used for registering the agent is equal to the name that has been specified inside the init(...) method, independent whether the name has been modified afterwards or not.* Please have a look at the example in Section 5.6 where this mechanism is applied.

The provision of a textual description via the method `getDescription()` can be handled in a similar way as the provision of a name as explained above.

Note that an agency automatically invokes an agent's `getName()` and `getDescription()` methods during the agent's creation. These method invocations are performed *before* the agent's security policies have been completely initialized. Thus, please do not implement any security-sensitive operations, such as the access of a file system or system properties, inside the methods `getName()` and `getDescription()`. Otherwise, if the agency's security service is active, the agent's creation may fail due to a security exception. Please refer to Chapter 12 for further details about the Grasshopper security service.

## 5.3 Code base

In order to create an agent or to re-instantiate a mobile agent after its arrival in a new destination agency, the agency must have access to the agent's class code. For this purpose, an agent code base can be specified when creating an agent via the agency API or UI. If no code base is explicitly provided (or if the

demanded agent classes are not included in an explicitly provided code base), the agency looks for the demanded classes in the directories maintained by the Java CLASSPATH environment variable.

Grasshopper supports two different kinds of code base:

- *File systems*

  The classes of an agent may be maintained in the file system of an agency. In this case, the code base, represented as String, must have the following format:

  `file:/<directory-path>`

  where `<directory-path>` represents a path that leads to the directory in which the agent's class files are stored. Single directories of the path are separated with slash ('/') characters. Note that on Windows machines, the letter of the maintaining device has to be specified:

  `file:/<driveLetter>:/<directory-path>`

- *Http servers*

  The classes of an agent may be maintained on an Http server. In this case, the code base, represented as String, must have the following format:

  `http://<domain-name>/<path>`

  where `<domain-name>` and `<path>` are structured in the usual way (i.e., domain components separated with a dot ('.') character, and path components separated with a slash ('/') character).

There are different possibilities to grant an agency access to an agent's code base:

**Local file system**

- *Class code is maintained by all agencies:*

  The agent's class files are initially stored in the file system of every potential destination agency, and additionally these class files are included in the CLASSPATH environment variable of the running Java environments. In this case, if an agent migrates to a new agency, the agency already has access to the agent's classes without the need of contacting a remote codebase.

  Note: Agent classes that are stored in the system's classpath are cached by the agency for its entire runtime. The reason is that the classes are not only maintained by the agent's own class loader, but also by the JVM's system class loader. That means, the agent classes are loaded only during the first creation of the agent. When the agent is created for the second time, the agency uses the internally cached classes instead of accessing the file sys-

tem again. This is also true if the agent's class files have been re-compiled before the agent's second creation.

This caching behavior may cause problems if an agent migrates from agency A to agency B where both agencies maintain different versions of the agent class. In this case, the agent will not be able to migrate, since agency B considers its maintained agent class as different from the agent class maintained by agency A.

**Home agency**

- *Class code is only maintained by the agent's home agency:*

  Initially, the agent's classes are only stored in the file system of the agent's home agency, i.e., on the host of the agency where the agent was created. If the agent migrates, each new destination agency has to request the class code from the home agency.

  Note: In this case, a destination agency caches the agent's classes only inside the agent's own class loader (and not in the system class loader). Thus, if two agents of the same class migrate to the same destination agency, the agency retrieves the classes both times. That means, the cached classes of the firstly arrived agent are not used for creating the secondly arriving agent. Instead, they are loaded again from the agent's home agency.

**Http server**

- *Class code is only maintained by a central Http server:*

  Initially, the agent's classes are only stored on a central Http server. In this case, even the agent's home agency has to retrieve the classes from this code base in order to create the agent.

  Note: In this case, a destination agency caches the agent's classes only inside the agent's own class loader (and not in the system class loader). Thus, if two agents of the same class migrate to the same destination agency, the agency retrieves the classes both times. That means, the cached classes of the firstly arrived agent are not used for creating the secondly arriving agent. Instead, they are loaded again from the agent's home agency.

**Previous agency**

- *Class code is only maintained by the previously visited agency:*

  In certain scenarios, the home agency of an agent is only temporarily connected to the network. For instance, an agent may be created on a notebook which is to be disconnected from the network right after sending the agent to another host. Supposed that no central Http server has been specified as code base and that the agent's classes are not maintained by the file systems of potential destination agencies, the agent's code can be forwarded from one agency to the next at each time the agent migrates. That

means, the agent's code base is always represented by the previously visited agency, and it changes with every migration.

Note: In this case, a destination agency caches the agent's classes only inside the agent's own class loader (and not in the system class loader). Thus, if two agents of the same class migrate to the same destination agency, the agency retrieves the classes both times. That means, the cached classes of the firstly arrived agent are not used for creating the secondly arriving agent. Instead, they are loaded again from the agent's home agency.

An agency accesses the different code bases in the following order:

1. System class loader of currently visited agency (maintaining classes loaded from the classpath of the local agency)
2. Previously visited agency
3. All locations (file system and/or Http server) specified in the agent's code base
4. Home agency

## 5.4 Grasshopper Addresses and Locations

In the context of Grasshopper, the term *location* or *address* specifies an agency or place. Every agency contains one or more *places* in which agents can run. Each place may have specific characteristics (defined by the agency administrator), such as an own security policy. (Please refer to the User's Guide for more information.)

In order to migrate (or to establish a communication connection as explained in Chapter 9), an agent has to provide information about the desired destination location. The agent specifies this information in terms of a Grasshopper *address* which is an instance of the class
`de.ikv.grasshopper.communication.GrasshopperAddress`

A Grasshopper address refers to a communication server of the desired destination agency, region registry, or external object. If an agency is addressed, a place name can optionally be specified.

A Grasshopper address covers the following components:

**Protocol types**

- *protocol type*: Type of the protocol to be used for the migration. The following protocols are supported:
  - *socket*: plain socket protocol

  - *rmi*: Java Remote Method Invocation (RMI) protocol
  - *iiop*: CORBA's Interoperable Inter-ORB Protocol (IIOP). This protocol is only available if a CORBA runtime environment has been installed. (Please refer to the User's Guide for more information.)
  - *socketssl*: plain socket protocol, protected via SSL. This protocol is only available if the security packages have been installed. (Please refer to the User's Guide for more information.)
  - *rmissl*: Java Remote Method Invocation (RMI) protocol, protected via SSL. This protocol is only available if the security packages have been installed. (Please refer to the User's Guide for more information.)
  - *grasshopperiiop*: In contrast to the previously mentioned protocol types, `grasshopperiiop` is a *meta protocol* that has to be mapped onto a concrete protocol type. For instance, an agent can try to establish a communication connection with a remote Grasshopper component by specifying the remote address in terms of a `grasshopperiiop` address of the form `grasshopperiiop://<hostName>/<agencyName>`.
    In this case, an agency domain service is required in order to determine the concrete address of the server side, including the real protocol type and port number.

- *host name*: Name or IP address of the destination host
- *object name*: Name of the destination agency, region registry, or external object
- *port numbe*r: Number of the port at which the communication server of the destination agency is listening.
- *place name*: Name of the destination place. This component is optional. If no place name is specified, the agent migrates to the default place „InformationDesk“ which exists in every Grasshopper agency.

The initialization of a `GrasshopperAddress` instance can be performed either by separately specifying the single components or by specifying all components in terms of a single `String` object. In the latter case, the address string has the following syntax:

**complete address**

```
protocol://hostName:portNumber/agencyName/placeName
```

Note that, in certain cases, a subset of the address components is sufficient. The following examples explain all possible cases:

- If an agency domain service is running and both the source and destination agencies are registered at this service, the minimal address consists of the

host name and agency name:

**minimal address**

`hostName/agencyName`

In this case, the agency domain service determines all communication servers of the specified destination agency, automatically selects one of them, and completes the address. Therefore, this minimal address should only be used if the migrating agent does not require a specific (e.g., secure) protocol. Usually, the plain socket protocol is selected.

Note that the agent will migrate to the default place InformationDesk of the destination agency. If the agent wants to migrate to a specific place, the place name must be appended to the address:

`hostName/agencyName/placeName`

- If an agency domain service is running and the migrating agent wants to be transferred via a specific protocol, the following address syntax can be used:

  `protocol://hostName/agencyName`

  In this case, the agency domain service checks if the specified destination agency provides a communication server that uses the desired protocol. If this is true, the domain service determines the port number and completes the address.

  Note that the agent will migrate to the default place InformationDesk of the destination agency. If the agent wants to migrate to a specific place, the place name must be appended to the address:

  `protocol://hostName/agencyName/placeName`

- If no agency domain service is running, the agent has to specify at least the protocol type, host name, port number, and agency name:

  `protocol://hostName:portNumber/agencyName`

  Note that the agent will migrate to the default place InformationDesk of the destination agency. If the agent wants to migrate to a specific place, the place name must be appended to the address:

  `protocol://hostName:portNumber/agencyName/placeName`

## 5.5 States and Life Cycles

At any point of its life time, a Grasshopper agent resides in a well-defined *state*. In each state, an agent has certain characteristics. Grasshopper defines the following state values:

- *active*: Immediately after its creation, an agent is transferred into the active

state. In this state, the agent is executing its task as specified inside the `live()` method, i.e., the agent's thread is running. When the agent's `live()` method has been finished, the agent remains in the active state and is still accessible by other entities as passive object. That means, other software components may invoke public methods of an agent via its proxy if the agent resides in the active state.

- *suspended*: Suspending an agent means suspending the agent's thread and thus interrupting the agent's active task execution. An agent can be suspended via the hosting agency's API (method `suspendAgent(...)`) or via the agency's UI. In order to transfer a suspended agent back into its active state, the agent can be resumed via the agency's API (method `resumeAgent(...)`) or via the UI. Note that, in contrast to the active state, the accessible methods of a suspended agent cannot be invoked by other components.
  Note that a suspended agent is not able to resume itself. Instead, it has to wait for being resumed by other entities. Thus, the `suspendAgent()` method should be handled with care inside an agent's code.
- *flushed*: This state is controlled by the agency's persistence service. When an agent is flushed, its data state is locally stored, and its instance is removed from the agency. Usually, the purpose of flushing an agent is to save system resources in times when the agent's existence is not required. A flushed agent is reactivated when another component tries to invoke any of its accessible methods. Detailed information about flushing agents and the persistence service in general can be found in Chapter 10.
  Note that a flushed agent is not able to re-activate itself. Instead, it has to wait for being re-activated by the hosting agency. Thus, the `flushAgent()` method should be handled with care inside an agent's code.

The following figure shows the state diagram of a Grasshopper agent.

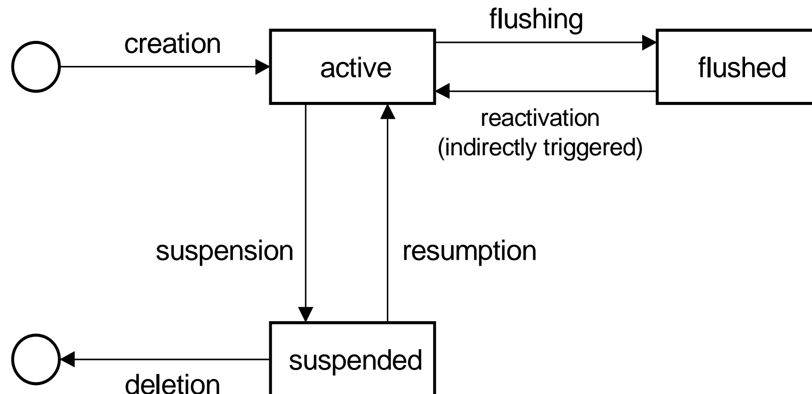



Figure 3: Agent State Diagram

In order to change the state of an agent, the hosting agency provides the following methods via its interface `de.ikv.grasshopper.agency.IAgentSystem`:

`flushAgent(...)`, `flushAgentAfter(...)`, `reloadAgent(...)`, `resumeAgent(...)`, `saveAgent(...)`, `saveAgentEvery(...)`, `suspendAgent(...)`.

Detailed information about these methods can be found in Section 9.11.2 which describes the functionality of the interface `IAgentSystem`.

Never use the `setState(...)` method of the class `de.ikv.grasshopper.type.AgentInfo`! This method is just meant to be used internally by an agency. Invoking this method will not change an agent's state. Instead, it will lead to an unpredictable behavior!

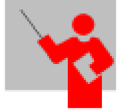
## 5.6 Example: PrintInfoAgent

The following example agent prints information about itself, retrieved from its `AgentInfo` instance. Note that a name and a textual description can be specified in terms of creation parameters.

As described in Section 5.2, the agent's name has to be set inside the `init(...)` method in order to be inserted into the `AgentInfo` instance. This fact may be of particular interest since the agent's `AgentInfo` instance is used for registering the agent inside the local agency and the agency domain service. An agent's name can be used as search key by other entities. If the searching entity expects another name than the one maintained inside the

`AgentInfo` object, the searched agent will not be found. As shown by the last lines of Example 3, a modification of the agent's name outside the `init(...)` method does not modify the corresponding name value that is maintained by the `AgentInfo` object. Thus, the name that is used for the agent's registration remains the same.

Example 3: PrintInfoAgent



```
package examples.simple;

import de.ikv.grasshopper.agent.MobileAgent;
import de.ikv.grasshopper.type.AgentInfo;

// This class realizes an agent that prints
// information about itself.
// It shows how to modify an agent's name and
// textual description, if desired.
public class PrintInfoAgent extends MobileAgent
{
  String agentName, agentDescription;

  // Creation arguments:
  //     args[0] = New agent name
  //     args[1] = New agent description
  public void init(Object[] creationArgs) {

    agentName = "PrintInfoAgent";
    agentDescription =
      "This agent tells you about its secrets.";
    if (creationArgs != null) {
      if (creationArgs.length > 0)
        agentName = (String)creationArgs[0];
      if (creationArgs.length > 1)
        agentDescription = (String)creationArgs[1];
    }
  }

  public String getName() {
    return agentName;
  }

  public String getDescription() {
    return agentDescription;
  }

  public void live() {
    AgentInfo myInfo = this.getInfo();
```

```
      log("My name: " +
      myInfo.getAgentPresentation().getAgentName());
      log("My id: " +
      myInfo.getIdentifier().toString());
      log("My type: " + myInfo.getType());
      log("My home: " + myInfo.getHome());
      log("My description: " + getDescription());

      // Further modifications of the variable
      // 'agentName' do not have any influence on the
      // AgentInfo structure. Therefore, the
      // initialization of this variable has to be
      // performed inside the init(...) method.
      agentName = "New name";
      log("My registered name remains the same: " +
      myInfo.getAgentPresentation().getAgentName());
    }
  }
```

Requirements:

- One running agency

Running the example:

Create the PrintInfoAgent inside the running agency via the agency's UI. The first creation parameter is interpreted as agent name and the second one as agent description. If less then two parameters are specified, the missing information is initialized with default values.

If you are using the textual user interface of the agency, please create the agent by means of the following command (which is meant only as an example, concerning the creation arguments):

```
cr a examples.simple.PrintInfoAgent InfoAgent „Want
some info?"
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

## 5.7 Summary

- Every Grasshopper agent maintain a set of information that describes its characteristics. This information can be accessed by the agent itself or by other entities.

- Every Grasshopper agent is uniquely identifiable inside the entire Grasshopper environment. An identifier describes the type of the associated component as well as the date, time, and location of the components's creation. Exceptions are agent copies which maintain the same identifier as their original instances, suffixed by a copy number.
- Every Grasshopper agent has a name and a textual description. In contrast to the identifier which is generated automatically and whose uniqueness is guaranteed, the name and textual description can be specified by the agent programmer or by the user (if this has been intended by the programmer during the implementation) (see Section 5.2).
- During its life time, an agent can reside in different states. In the *active* state, an agent is performing its task and is accessible by other entities. In the *suspended* state, an agent's thread is suspended. In the *flushed* state, an agent does not exist any more as runtime instance. Instead, the agent's data state is persistently stored in a local database, and the runtime instance is removed. In this case, the agent is automatically re-instantiated when another entity tries to access it (see Chapter 10). Note that the ability of persistently storing agents is only available if the hosting agency runs with an activated persistence service *and* if the agents are derived from one of the classes `de.ikv.grasshopper.agent.PersistenMobileAgent` or `de.ikv.grasshopper.agent.PersistentStationaryAgent`.

# 6 Move Me!

Grasshopper provides three possibilities for moving agents.

- *Via the agency's UI:* An agency administrator may move an agent via the graphical or textual user interface of an agency. This mechanism is associated with the agency usage rather than with agent programming, and thus it is not described in this context. Detailed information about the functionality of the agency's user interfaces can be found in the User's Guide.
- *Via the agency's API:* A Grasshopper mobile agent can be moved by another agent or object via the `moveAgent(...)` method of the hosting agency. This mechanism is explained in Section 9.11 which deals with the functionality provided by agencies.
- *Via the agent's API:* A Grasshopper mobile agent may move itself by invoking the `move(...)` method of its own superclass `MobileAgent`. *Note that this is the usual way for an agent to actively migrate through the distributed environment.* In contrast to this, the two possibilities described above just enable external entities to move an agent.

This section only deals with the third possibility of the above list.

## 6.1 Strong vs. Weak Migration

A mobile agent is able to perform different parts of its task on different network locations. For example, an agent may need to gather information from different databases. In order to take advantage of local interactions instead of remote procedure calls, the agent visits the database hosts one after the other, accesses the databases, filters the information locally, and just maintains the most interesting subset of the information before migrating to the next host.

Figure 4 shows this general procedure.

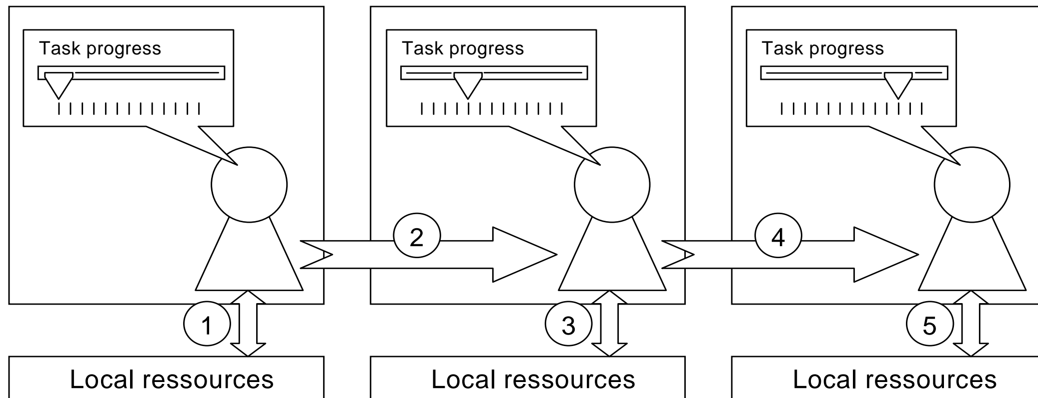



Figure 4: Agent Migration

Two different kinds of migration can be separated:

**Strong migration; execution state**

Strong migration means that an agent migrates together with its whole *execution state*. An agent's execution state contains all stack information that is required to characterize the point of execution that the agent has currently reached. After a strong migration, the agent continues processing its task exactly at the point at which it has been interrupted before the migration.

**Weak migration; data state**

Weak migration means that an agent just maintains its *data state* when travelling from one location to another. An agent's data state consists of internal variable values that are serialized at the agent's old location, transferred across the network, and provided to the agent again at the new location. The agent programmer has to decide which variables are to be part of the data state.

**Restrictions of Java**

The Java programming language does - by default - not offer the possibility to capture the execution state of a process or thread. The only possibility to achieve this is to modify the Java Virtual Machine. However, since one objective of Grasshopper is to be as open as possible concerning its underlying software environment, this possibility was not considered during the development of the platform. Since Grasshopper has to be executable on all JVMs that are compliant to the Java specifications of Sun Microsystems, Grasshopper agents use weak migration for travelling across the network.

The following section shows how strong migration can - with certain restrictions - be „simulated" by using weak migration. This principle is fundamental for developing mobile agents on top of Grasshopper.

## 6.2 The Migration Procedure

When we speak of agent migration, we mean the travel of an agent's code and data state from one agency or place to another. However, from an implementation-related point of view, an agent is not really travelling. Instead, after each migration, a *new agent instance* is created at the destination agency, and the old agent instance is removed at the source agency. By supplying the new instance with the data state of the old one (including among others the old agent's identifier), the agent seems to remain the same.

The „migration" consists of the following execution steps:

**The migration procedure**

1. The agent's migration is initiated. This can be done either by the agent itself (via its own `move(...)` method), by other software components (via the `moveAgent(...)` method offered by the agency's API), or by human users (via the agency's UI).

**beforeMove()**

2. The agent's `beforeMove()` method is automatically called by the agency in order to enable the agent to prepare its migration, e.g., by releasing occupied resources or removing references. This method may be of particular importance if the agent's migration is triggered by external entities (software components or human users), because in this case the migration request is usually not expected by the agent. If the agent itself triggers its migration, it has the possibility to prepare the migration already before invoking its `move(...)` method.

   **VetoException**

   An agent may prohibit its own migration. If the agent does not want to be moved, the agent can throw the `de.ikv.grasshopper.agent.VetoException` inside its `beforeMove()` method. If the move request has been initiated via the agency's UI, the user is informed about the migration rejection via the user interface. If the move request has been initiated via the agency's API, the agency forwards the `VetoException` to the triggering software component.

3. The agent's execution is interrupted by stopping the agent thread. Since each agent is created inside its own thread group, additionally all threads are stopped that have been created by the agent itself.

4. The agent's data state is serialized. That means, all instance variable of the agent that are *not* declared as *transient*, are put into a data stream. Please refer to Section 6.3 for detailed information about the data state.

5. The agent's serialized data state as well as additional information are transferred to the destination agency. Among others, the additional information covers the agent class name and its code base. This is required by the destination agency to create a new instance of the agent.

6. The destination agency creates a new instance of the agent and provides the agent with its transferred data state. If the agent's class code is not initially maintained by the destination agency, it is retrieved via Java class loading mechanisms by accessing the code base that has been delivered by the source agency.
7. The destination agency informs the source agency about the successful creation of the agent. Now the source agency removes the old agent instance. (Exactly speaking, the agency removes its references to the agent and thus enables the Java garbage collector to release the agent's occupied resources.)

**afterMove()**

8. The destination agency automatically calls the agent's `afterMove()` method. In this way, the agent is able to prepare the resumption of its task execution, e.g., by allocating references and resources. (Note that the `afterMove()` method is also called after an agent's copying.)
9. The destination agency starts the thread of the agent. Now the agent is able to continue its task execution.

## 6.3 The Data State: Mobile Information

When implementing a mobile agent, the developer has to determine which parts of the agent's internal data has to be maintained when the agent migrates.

**Definition: data state**

An agent's data state consists of all non-transient instance variables of the agent class, i.e., the class that is derived from one of the agent super classes `MobileAgent`, `StationaryAgent`, `PersistentMobileAgent`, or `PersistentStationaryAgent`[1]. Note that the data state also comprises all non-transient instance variables of the super classes. When a mobile agent migrates, its data state is serialized at the source location, transferred across the network, and provided to the migrated agent instance at the destination location.

**AgentInfo**

One part of the data state of *every* Grasshopper agent is the `AgentInfo` structure which maintains important information of the agent, such as its identifier, name, type, and properties. Please refer to Chapter 5 for further details about the AgentInfo structure.

**class Serializable**

A general prerequisite for the serialization of Java objects is that the corresponding Java class (or any of its superclasses) implements the `java.io.Serializable` or `java.io.Externalizable` interface.

1. Although *every* Grasshopper agent maintains a data state, the data state is only of importance for *mobile* agents, since it's purpose is to preserve data during an agent's migration.

Please refer to the Java documentation for more information about object serialization.

Each Grasshopper agent is by default serializable. However, an agent may instantiate objects that do not fulfill the serialization criteria. Since the serialization of an agent covers all non-transient instance variables of the agent, all objects belonging to the agent's data state have to be serializable.

**Defining the data state**

When implementing a mobile agent, the programmer has to evaluate which of the agent's instance variables are to be part of the data state. Since the size of the data state has a high impact on the migration duration, only a minimal set of instance variables should be included. The following examples are meant to explain how a programmer can evaluate which variables have to be included into the data state:

1. If, after each migration, the value of an instance variable is modified before the variable is read by the agent or other components, it is not necessary to transfer the old (and not anymore required) value to the new location. In this case, the variable should be transient in order to exclude it from the data state.
   Especially if an agent creates its own GUI, the GUI should be declared transient (or defined inside a method instead as instance variables). Beside the reduction of the agent's data state, the main reason for this is that several Java GUI classes are dependent on a specific operating system. Thus, if an agent migrates between agencies that run on different operating systems and if this agent carries a set of GUI classes, failures may occur due to incompatibility problems associated with these classes.
2. If an instance variable is semantically bound to the current local system environment, its value may become invalid after the agent's migration. This problem often occurs in a CORBA environment with CORBA objects referenced by the agent. In this case, the variable should be transient, and the agent should re-instantiate and initialize it anew after each migration.
3. If an agent instantiates objects that are (entirely or partly) not serializable, the agency's attempt to serialize the agent fails. That means, an agent that has allocated references to non-transient, non-serializable objects can neither migrate nor be stored by the agency's persistence service. To avoid this, non-serializable objects should be transient.

## 6.4 Structuring an Agent's Life

Due to the explanations in Section 6.1, Grasshopper agents use weak migration for travelling from one agency to another. Parts of the agent's data state can be used in order to enable an agent to *continue* its task processing after a migration instead of *restarting* its task from the beginning on. For this purpose, the agent's `live()` method can be separated into different *execution blocks*, each one covering a set of operations that has to be performed at a single location. After completing the execution of one block, the agent migrates to the next location, triggered by a `move(...)` method at the end of the performed block, and starts executing the next block after arriving at the new location (Figure 5).

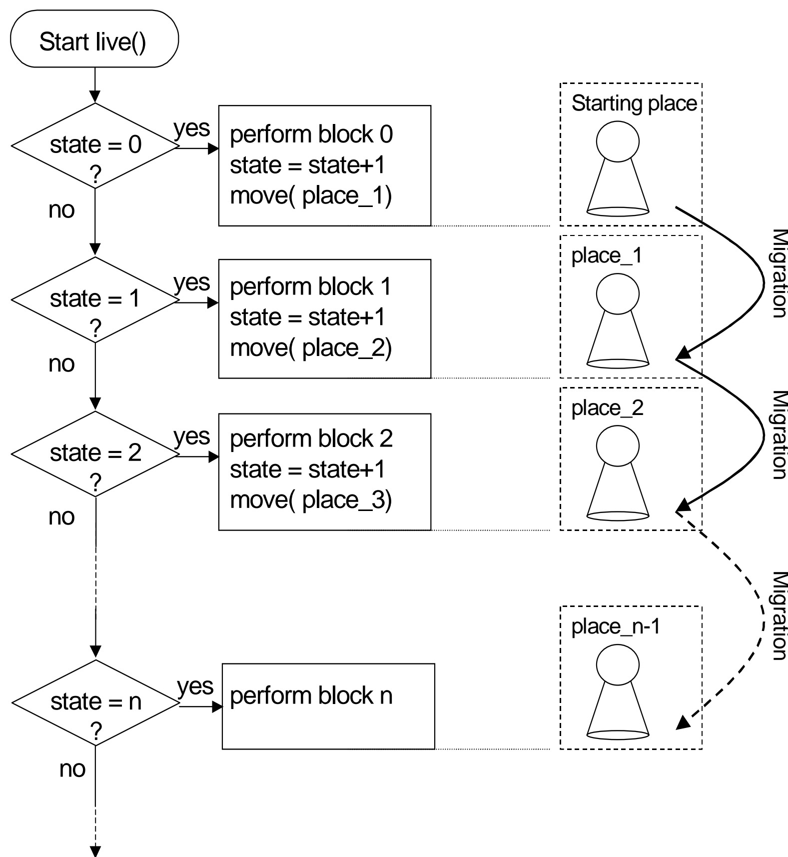



Figure 5: Structure of an Agent's live() Method

In Figure 5, `'state'` is a non-transient instance variable, declared in the agent class. In this way, `state` becomes part of the data state. By analyzing the value of this variable, the agent determines which execution block has to be performed. Inside the performed execution block, the agent sets `state` to a new value before invoking the `move(...)` method. During the migration,

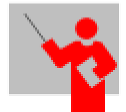
the new `state` value is serialized, transferred to the new location, and provided to the migrated agent instance. Please have a look at Example 4 in Section 6.5 which shows an agent that makes use of its data state.

## 6.5 Example: BoomerangAgent

The following example is our first mobile agent that makes use of its mobility and of its data state. After starting the agent, a small window appears, requesting a new destination address from the user. When pressing the OK button, the agent migrates to the specified location. After its arrival, the agent asks for the permission to migrate back to its home location. The different behavior of the agent, dependent on its current location, is realized by means of the agent's data state that is represented in terms of a single integer variable named `state`:

- `state = 0`: This is the initial state, set in the agent's `init(...)` method. In this state, the first execution block inside the `live()` method is performed. The agent asks for a new location, increments the state variable, and migrates.
- `state = 1`: In this state, the agent does not request the input of a new location, but just asks for the permission to migrate back to its home location. Before the agent travels home, `state` is set back to 0.

Example 4: BoomerangAgent



```
package examples.simple;

import de.ikv.grasshopper.agent.MobileAgent;
import de.ikv.grasshopper.communication.*;
import javax.swing.*;
import java.awt.*;

// This class realizes an agent that moves to a remote
// agency and, after this, returns to its origin.
public class BoomerangAgent extends MobileAgent
{
  // A little data state.
  int state;

  // No creation arguments needed.
  public void init(Object[] creationArgs) {
    // Initialize data state
    state = 0;
```

```
  }

  public String getName() {
    return "BoomerangAgent";
  }

  public void live() {
    String location;

    switch(state) {
      case 0:
        log("Waiting for new location...");
        location = JOptionPane.showInputDialog(
          null, "Where shall I go?");
        if (location != null) {
          state = 1;
          log("Trying to move...");
          try {
            // Go away!
            move(new GrasshopperAddress(location));
          }
          catch (Exception e) {
            log("Migration failed: ", e);
          }
          // The next statement is only reached
          // if the migration failed!!!
          state = 0;
        }
        break;
      case 1:
        log("Arrived at destination!");
        JOptionPane.showMessageDialog(
          null, "Let me go home!");
        state = 0;
        log("Trying to move...");
        try {
          // Come home!
          move(getInfo().getHome());
        }
        catch (Exception e) {
          log("Return trip failed: ", e);
        }
        // The next statement is only reached
        // if the migration failed!!!
        break;
    }
    log("Terminating my life.");
  }
```

```
}
```

Requirements:

- Optionally a running agency domain service. Note that the domain service has to be started before the agencies, and the domain service's address has to be specified when starting the agencies in order to register them. Please refer to the User's Guide for information about how to start agencies and agency domain services.
- At least two running agencies.
  *Since the BoomerangAgent creates an own GUI that may block the agency GUI, it is recommended that you do not activate the agency GUI. Instead, start the agencies just with their textual interface (command option -tui). Please refer to the paragraphs titled „Running the Examples" at the beginning of Chapter 2 in order to get a detailed explanation about the possibly occurring GUI problems.*

Running the example:

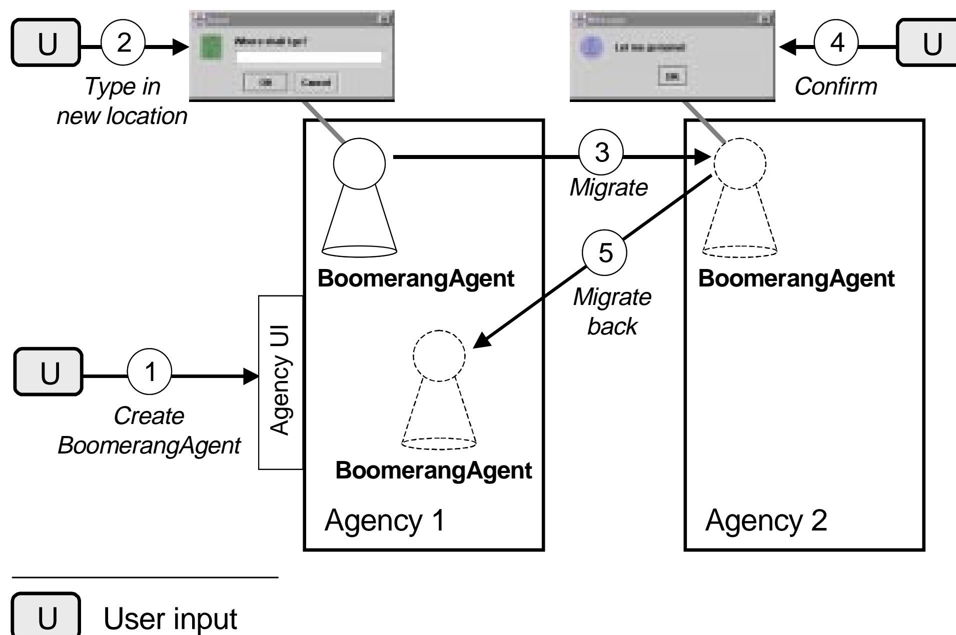



Figure 6: BoomerangAgent Scenario

Create the BoomerangAgent inside one of the running agencies via the agency's UI (1).

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.simple.BoomerangAgent
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

Type in the address of one of the other running agencies and press OK (2). If no agency domain service is running, you have to specify the complete address as explained in Section 5.4. You can determine the running communication servers of each agency via their GUI (Menu: File -> Preferences -> Servers) or TUI (Command: 'status').

Note: If you press the Cancel button, the agent's GUI disappears, and there is no possibility to re-activate it. Please refer to Chapter 7 in order to find a solution for this problem.

After pressing the OK button, the agent migrates to the specified agency (3) and creates another graphical dialog. Once arrived, the agent asks for the permission to return to its home agency. Press the OK button (4) in order to confirm the migration request. After its migration back to its original agency (5), the agent again creates its initial GUI, and the scenario proceeds with step (2).

Example variations:

Run the example with and without a running agency domain service. Each time, try different addresses by specifying or not specifying single address components, and see what happens. All possible address syntaxes are described in Section 5.4.

Create new communication servers of different protocols and use these new servers for moving the agent around. The kinds of supported protocols depend on your Grasshopper installation. For instance, SSL-protected protocols require the security packages, and the CORBA protocol IIOP requires a CORBA runtime environment. Detailed information about the installation of these extensions can be found in the User's Guide.

## 6.6 Summary

- An agent's migration can be triggered by the agent itself (via the agent's API), by other software components (via the agency's API), or by users (via the agency's UI).
- An agent may prohibit its migration by throwing a `VetoException` from inside its `beforeMove()` method.
- Grasshopper uses weak migration. That means, the agent's data state (i.e.,

all non-transient instance variables) is transferred to the destination agency. Strong migration (i.e., the migration of the agent's execution stack) is not realized, since this is not supported by standard Java Virtual Machines.

- Parts of the agent's data state can be used to enable an agent to continue its task execution after a migration (instead of starting its execution from the beginning on). For this purpose, the agent's `live()` method can be separated into different execution blocks where one block is completely performed within a single location. See Section 6.4 for a detailed description.
- Grasshopper mobile agents are able to migrate from one place to another. The source and destination places may be hosted by the same or by different agencies. The location of a place is specified in terms of a Grasshopper address, i.e., an instance of the class `de.ikv.grasshopper.communication.GrasshopperAddress`. A complete address (represented as string) has the following format:

  protocol://hostName:portNumber/agencyName/placeName

  Under certain conditions, some of the address components are not required. Please refer to Section 5.4 for detailed information.

# 7 Action!

The class `Agent` (which is the superclass of all Grasshopper agents, as explained in Chapter 3) provides a method named `action()`. This method is automatically invoked by the agency if a user performs one of the following actions:

- double-click on the corresponding agent entry in the agency GUI
- selection of an agent in the agency GUI, right-click on the selected agent entry and left-click on the invoke command of the appearing menu
- left-click on the invoke icon inside the icon bar of the agency GUI
- left-click on the menu item Object->Invoke
- typing the invoke command of the agency TUI

Beside the user interfaces, the `action()` method of an agent can also be triggered by another software entity via the `invokeAgentAction(...)` method of the agency's programming interface `de.ikv.grasshopper.agency.IAgentSystem`. Please refer to Section 9.11.2 where the API of agencies is explained.

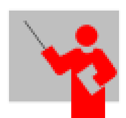
The intention of the `action()` method is to enable a user to trigger a certain action of an agent via the agency's user interface. For instance, a double-click on the agent entry may start an agent-specific GUI, as shown in the following example. Another possibility is shown in Example 6 (Section 8.1) where the `action()` method is used to print an agent's properties.

## 7.1 Example: ActionAgent

The following example is an extension of the BoomerangAgent introduced in Section 6.5. The enhancement is that the action method reactivates the agent's `live()` method. Thus, in contrast to the BoomerangAgent, the GUI can be re-activated by a user via one of the actions listed above.

Note that re-activating an agent's `live()` method via its `action()` method does not re-activate the agent's thread (whose execution stopped when the `live()` method terminated for the first time). Instead, the re-activated agent runs in a thread of the communication service of the local agency.

Example 5: ActionAgent



```
package examples.simple;
```

```
import de.ikv.grasshopper.agent.MobileAgent;
import de.ikv.grasshopper.communication.*;
import javax.swing.*;
import java.awt.*;

// This class realizes an agent that moves to a remote
// agency and, after this, returns to its origin.
// The agent can be re-activated via its 'action'
// method.
public class ActionAgent extends MobileAgent
{
  // A little data state.
  int state;

  // No creation arguments needed.
  public void init(Object[] creationArgs) {
    // Initialize data state
    state = 0;
  }

  public String getName() {
    return "ActionAgent";
  }

  public void action() {
    // Re-animate the agent
    log("Re-animated!");
    live();
  }

  public void live() {
    String location;

    switch(state) {
      case 0:
        log("Waiting for new location...");
        location = JOptionPane.showInputDialog(
          null, "Where shall I go?");
        if (location != null) {
          state = 1;
          log("Trying to move...");
          try {
            // Go away!
            move(new GrasshopperAddress(location));
          }
          catch (Exception e) {
            log("Migration failed: ", e);
```

```
          }
          // The next statement is only reached
          // if the migration failed!!!
          state = 0;
        }
        break;
      case 1:
        log("Arrived at destination!");
        JOptionPane.showMessageDialog(
          null, "Let me go home!");
        state = 0;
        log("Trying to move...");
        try {
          // Come home!
          move(getInfo().getHome());
        }
        catch (Exception e) {
          log("Return trip failed: ", e);
        }
        // The next statement is only reached
        // if the migration failed!!!
        break;
    }
    log("Terminating my life.");
  }
}
```

Requirements:

- Optionally a running agency domain service. (Note that the agency domain service has to be started before the agencies, and the domain service's address has to be specified when starting the agencies in order to register them. Please refer to the User's Guide for information about how to start agencies and agency domain services.
- At least two running agencies
  *Since the ActionAgent creates an own GUI that may block the agency GUI, it is recommended that you do not activate the agency GUI. Instead, start the agencies just with their textual interface (command option -tui). Please refer to the paragraphs titled „Running the Examples" at the beginning of Chapter 2 in order to get a detailed explanation about the possibly occurring GUI problems.*

Running the example:

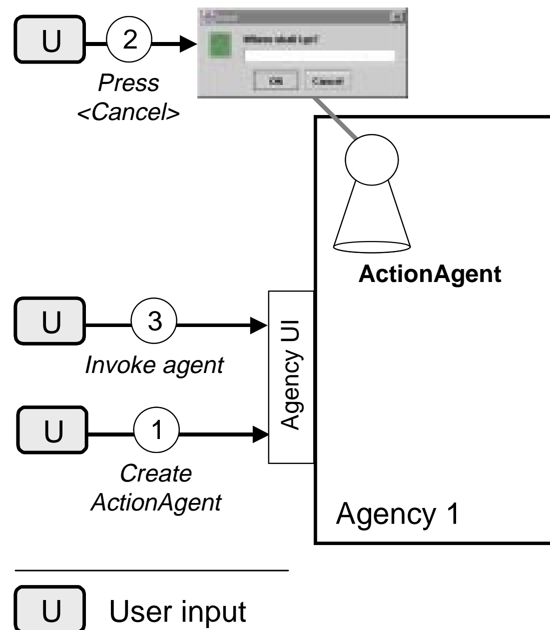



Figure 7: ActionAgent Scenario

Create the ActionAgent inside one of the running agencies via the agency's UI (1).

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.simple.ActionAgent
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

Press the Cancel button (2), and the GUI disappears. Now perform a double-click on the agent entry in the agency GUI or perform the 'invoke' command of the agency TUI (3). The agent's GUI appears again, and you can continue moving the agent around, as explained in the BoomerangAgent example (see Example 4 in Section 6.5).

## 7.2 Summary

- The `action()` method of an agent is automatically invoked by the hosting agency on behalf of a user (via the agency's UI) or on behalf of another software entity (via the agency's API).

- The agent programmer can override the `action()` method in order to enable a user to trigger certain actions of the corresponding agent.

# 8 Clones and Copies

**copy(...)**

Sometimes, it may be desirable for an agent to create a copy of itself and to order this copy to perform one specific task, while the original instance is occupied with another task. For this purpose, the agent's superclass provides the method `copy(...)`. By invoking this method, an agent creates another instance of its own agent class. The location in which the new instance is to be created has to be provided as parameter of the `copy(...)` method.

The copy procedure is realized similar to an agent's migration. That means, the new agent instance is created at the desired location, and the data state of the original instance is transferred (in serialized form) to the new instance. The important difference is that, in contrast to the migration, the original agent instance is not removed, but continues executing its task.

Note: The previous releases of Grasshopper provided two separate methods: `clone` and `copy`. The difference was that, in contrast to `copy`, the `clone` method did not require any parameter, and the new instance was created in the same location as the original instance. To achieve this with the new Grasshopper release, just specify the current location of the original instance as parameter of the `copy(...)` method, or simply set the value to `null`.

**The copy procedure**

1. The agent's copying is initiated. This can be done either by the agent itself (via its own `copy(...)` method), by other software components (via the `copyAgent(...)` method offered by the agency's API), or by human users (via the agency's UI).

2. The agent's `beforeCopy()` method is automatically called by the agency in order to enable the agent to prepare its copying. This method may be of particular importance if the agent's copying procedure is triggered by external entities (software components or human users), because in this case the copy request is usually not expected by the agent. If the agent itself triggers its copying, it has the possibility to prepare the copying already before invoking its `copy(...)` method. **before-Copy()**

   **VetoException** An agent may prohibit its own copying. If the agent does not want to be copied, the agent can throw the `de.ikv.grasshopper.agent.VetoException` inside its `beforeCopy()` method. If the copy request has been initiated via the agency's UI, the user is informed about the copy rejection via the user interface. If the copy request has been initiated via the agency's API, the agency forwards the `VetoException` to the triggering software component.

3. The agent's data state is serialized. That means, all instance variable of the

agent that are *not* declared as *transient*, are put into a data stream. Please refer to Section 6.3 for detailed information about the data state.

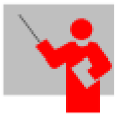
4. The agent's serialized data state as well as additional information are transferred to the destination agency. Among others, the additional information covers the agent class name and its code base. This is required by the destination agency to create a new instance of the agent.
5. The destination agency creates a new instance of the agent and provides the agent with its transferred data state. If the agent's class code is not initially maintained by the destination agency, it is retrieved via Java class loading mechanisms by accessing the code base that has been delivered by the source agency.

**afterCopy()**

6. The destination agency automatically calls the agent's `afterCopy()` method. In this way, the agent is able to prepare the start of its task execution, e.g., by allocating references and resources.
7. The destination agency starts the thread of the agent. Now the agent is able to start its task execution.

**Identifier handling**

Note that the copied agent instance gets a new identifier. This identifier is composed of the identifier of the original instance, suffixed by a period and the copy number, starting with '1'.



Example: The identifier of the original agent instance is

```
Agent#192.168.100.31#1999-09-28#09:51:13:453#0
```

The first copy of this agent gets the following identifier:

```
Agent#192.168.100.31#1999-09-28#09:51:13:453#0.1
```

The fifth copy of this *new* agent gets the following identifier:

```
Agent#192.168.100.31#1999-09-28#09:51:13:453#0.1.5
```

## 8.1 Example: CopyAgent

Inside its `init(...)` method, the CopyAgent creates two properties:

- Property 1: key = „generation"; value = „parent"
- Property 2: key = „copyPermission", value = „true"

(As explained in Chapter 5, an agent may have a set of properties that is maintained by the agent's `AgentInfo` object. Initially, no property is defined, but the agent may set and modify its properties at any time. All properties are automatically part of the agent's data state, i.e., they remain valid when the agent migrates. If an agent creates a copy of itself, the copy gets the same properties

as the original agent.)

The agent's `action()` method enables a user to watch the agent's properties at any time.

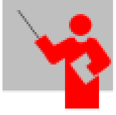
The `beforeCopy()` method of the CopyAgent reads the „copyPermission" property value. If this property is set to „true", the agent allows its copying. If the „copyPermission" property is set to „false", the agent prohibits its copying by throwing the `VetoException`. (As explained above, the general purpose of the `beforeCopy()` method is to enable an agent to react on a copy request. The method is automatically invoked by the local agency after receiving a copy request by the agent itself or by another entity.)

The CopyAgent's `afterCopy()` method just notifies the user about the successful arrival of a copy. (As explained above, the general purpose of the `afterCopy()` method is to enable an agent to react on its arrival in a new agency after a migration or copy procedure. The method is automatically invoked by the local agency before starting the agent's `live()` method.)

The CopyAgent uses the „generation" property inside its `live()` method in order to determine whether it is still the original instance (i.e., the „parent") or a copied instance (i.e., a „child"). In this way, the „generation" property represents the agent's data state, similar to the `state` variable of the BoomerangAgent (see Example 4 in Section 6.5).

The agent's `live()` method is divided into two parts:

- If the agent's generation value equals „parent", the agent requests a list of all agencies that are registered at the agency domain service. After this, the agent changes its „generation" value to „child" and sends one copy of itself to every available agency. Finally, the agent resets the generation value to „parent", and the `live()` method terminates. Note that the method `listAgencies(...)` has not been explained yet. This example does not focus on describing how to contact the agency domain service, but this functionality is needed to determine all registered agencies. Please refer to Section 9.12 for detailed information about contacting a domain service.
- If the agent's generation value equals „child" (which is true for all copies of the parent agent that have been created in the first part of the `live()` method), the agent changes its „copyPermission" property value to „false". After this, it is not possible for a user or software component to create copies of this agent, since the `beforeCopy()` method throws a `VetoException`.



Example 6: CopyAgent

```
package examples.simple;

import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.util.*;
import java.util.Properties;

// This class realizes an agent that produces copies of
// itself. The agent uses its internal properties in
// order to prohibit the further copying of its
// children.
public class CopyAgent extends MobileAgent
{
  transient AgentSystemInfo availableAgencies[];

  public void init(Object[] creationArgs) {
    setProperty("generation", "parent");
    setProperty("copyPermission", "true");
  }

  public void action() {
    log("Generation = " + getProperty("generation") +
    ", copyPermission = " +
    getProperty("copyPermission"));
  }

  public String getName() {
    return "CopyAgent";
  }

  public void beforeCopy()
        throws VetoException {
    if (getProperty("copyPermission").equals("false"))
    {
      log("Sorry, copying not allowed.");
      throw new VetoException();
    }
  }

  public void afterCopy() {
    log("Child has arrived.");
  }

  public void live() {
    String generation = getProperty("generation");
```

```
    if (generation.equals("parent")) {
      // Get a list of all available agencies
      availableAgencies = getRegion().listAgencies(
        null, new SearchFilter());
      // Create properties  for the copies
      Properties childProps = new Properties();
      childProps.setProperty("generation","child");
      childProps.setProperty("copyPermission",
        "false");
      // Send a copy to each agency
      log(availableAgencies.length +
        " agencies found");
      for (int i = 0; i < availableAgencies.length;
      i++) {
        log("Sending one copy to agency " +
        availableAgencies[i].getLocation());
        try {
          copy(availableAgencies[i].getLocation(),
            childProps);
        }
        catch (Throwable e) {
          log("Copy to location " +
          availableAgencies[i].getLocation() +
          " failed. ", e);
        }
      }
    }
  }
}
```

Requirements:

- A running agency domain service. Note that the domain service has to be started before the agencies, and the domain service's address has to be specified when starting the agencies in order to register them. Please refer to the User's Guide for information about how to start agencies and agency domain services.
- At least two running agencies. (The originally created agent will create a copy of itself in every available agency, so that more than one agency should be started. However, the example also runs with a single agency. In this case, a single copy will be created.)

Running the example:

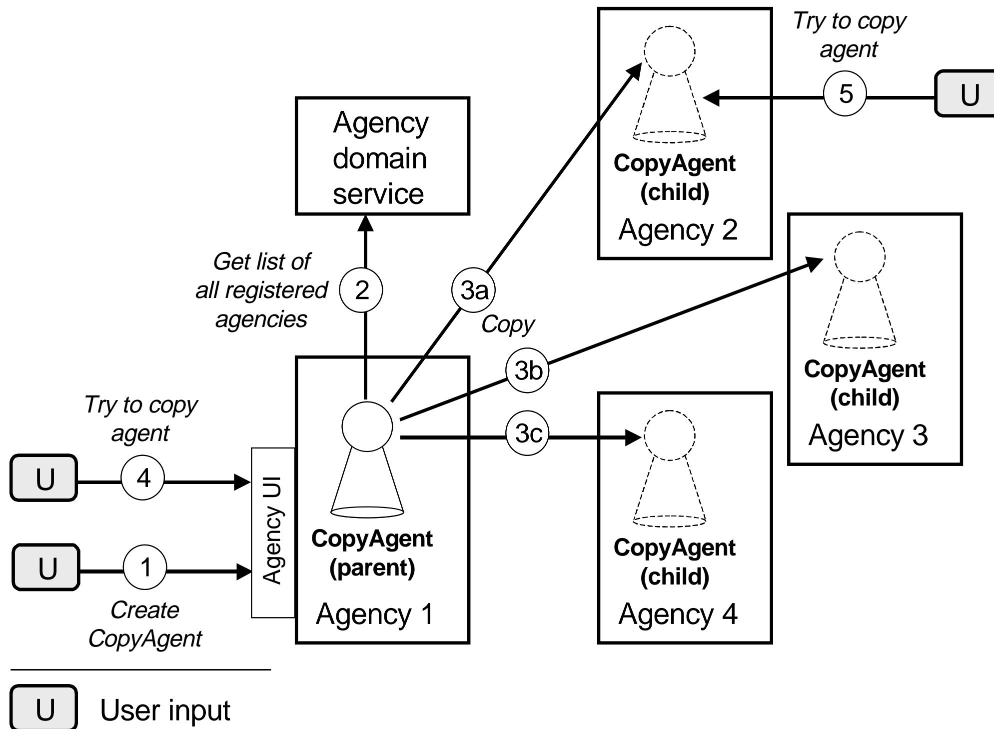



Figure 8: CopyAgent Scenario

Create the CopyAgent inside one of the running agencies via the agency's UI (1).

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.simple.CopyAgent
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

The agent contacts the agency domain service in order to retrieve a list of all registered agencies (2). After this, the agent creates one copy of itself in every agency contained in the retrieved list (3). These copies in turn will not create any further copies. They just print a message onto the console window of their local agency and terminate afterwards.

Try to copy the original agent (4) as well as the copied agents (5). You will find out that the original agent (i.e., the parent agent) can be copied, whereas the copied agents (i.e., the child agents) do not allow the user to copy them. A copy of the original agent behaves exactly as the original

agent itself, i.e., the copy will again produce „children“.

Look at the identifiers of all agents. As explained in Section 5.1, the copy number is appended to the original identifier.

## 8.2 Summary

- Copying an agent means to create a new instance of an existing agent and to transfer a copy of the current data state to the new instance. The new instance can be created in the same location as the original instance or in a remote agency.
- An agent may prohibit its copying by throwing a `VetoException`.
- The identifier of the new instance is composed of the complete identifier of the original instance, suffixed by the copy number.

# 9 The Communication Service

One of the most significant benefits of mobile agent technology is the ability of active service agents to migrate to different network locations in order to access static software components locally instead of interacting via remote procedure calls. However, in several application scenarios the traditional client/server paradigm using RPC still provides efficient solutions.

**Proxy objects**

By means of its communication service, Grasshopper achieves an integration of both, agent migration combined with local interactions, and remote interactions across the network. When using the communication service, clients do not have direct references to the corresponding servers. Instead, an intermediate entity is introduced, named *proxy object* or simply *proxy*. In order to establish a communication connection with a server, a client creates a proxy that corresponds to the desired server. This proxy in turn establishes the connection to the actual server. In the context of Grasshopper, three different kinds of servers are possible: agencies, agents, and agency domain services.

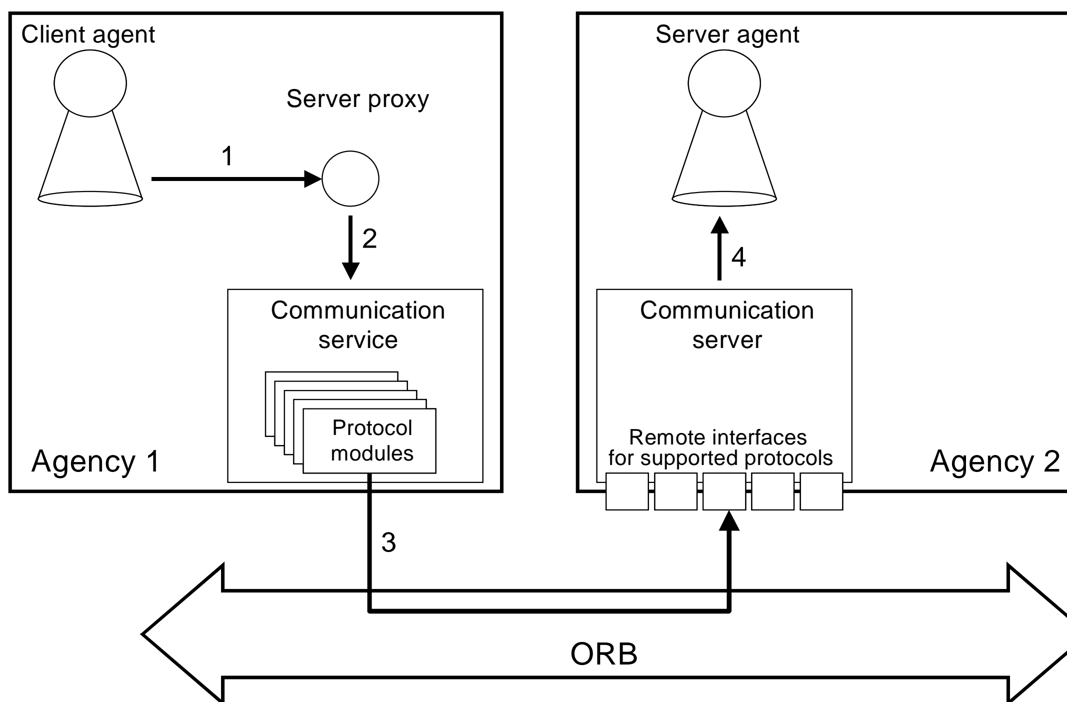



Figure 9: Communication via Proxies

**Location transparency**

If an agency domain service is running and both the client's and the server's agency are registered at this service, a location-transparent communication session can be established. The client simply provides the identifier of the demanded server, and the proxy object automatically contacts the agency domain service in order to determine the server's location. If the server is realized as mobile agent that moves to another location the proxy at the client side keeps track of the server by requesting its new location from the agency domain service.

In order to use the communication service, the following steps have to be performed:

1. Implementation of the server side (see Section 9.1)
2. Generation of a server proxy (only required for Java releases prior to JDK 1.3, see Section 9.2)
3. Implementation of the client side (see Section 9.3)

A client can access a server in different ways. Depending on the concrete application scenario, the programmer has to select between

- Synchronous and asynchronous communication (see Section 9.5)
- Static and dynamic communication (see Section 9.7)
- Unicast and multicast communication (see Section 9.9)

By default, each Grasshopper agency as well as the agency domain service are accessible via proxies. Agents can retrieve the proxy of the *local* agency via the method `getAgentSystem()` which is provided by the agents' superclass `Agent` (see Section 9.11.3). Besides, agents can create proxies of remote agencies (see Section 9.11.4). Finally, agents can get the proxy of an *agency domain service* via the method `getRegion()` which is provided by the agents' superclass `Agent` (see Section 9.12).

The main objective of an agency domain service is to enable users and software entities to search for specific components (i.e., agencies, places, and agents). Demanded components can be described in terms of filters (see Section 9.13) in order to facilitate the search.

Since clients as well as servers may be realized by mobile agents, the Grasshopper communication service has been designed to handle their migration during a communication session. Please refer to Section 9.14 for detailed information about migrating servers and clients.

In several application scenarios, an integration of Grasshopper into existing applications is required.

## 9.1 Implementing the Server Side

The server side of a Grasshopper communication connection is realized in terms of a Java object that provides at least one public method to the communication service. All methods that are to be accessible via the communication service have to be included in a Java interface that is implemented by the server object. In detail, the following steps have to be performed:

1. Implementation of a server (agent) class
2. Selection of those server methods that are to be accessible via the communication service
3. Definition of a Java interface that includes the previously selected methods, called *server interface* in the scope of this chapter. This interface has to be implemented by the server class.

   If also methods of the agent's superclasses are to be accessible, the corresponding interfaces (e.g., `IAgent` or `IMobileAgent`) have to be inherited by the newly defined server interface. Note that these superclass interfaces only offer a subset of the methods that are provided by the superclasses themselves. For instance, the interface `IAgent` only covers the method `getInfo()`, while the superclass `Agent` provides a large set of methods to its subclasses.

   If a server interface method uses user-defined Java classes as parameters and/or return types, these classes have to be serializable, i.e., they have to implement the interface `java.io.Serializable`. The reason is that the Grasshopper communication service uses the serialization mechanism for transferring the information that is associated with a remote method invocation.

## 9.2 Creating Proxy Objects

As explained above, a client accesses a locally created proxy object in order to communicate with a server. This proxy provides all public methods that have been defined in the server interface.

The way of creating a proxy object depends on the used Java release. If JDK 1.3 or higher is used, the proxy creation can be performed dynamically by a client agent during its runtime, just by using the class of the server interface (see Section 9.2.2). If an earlier release of Java is used, the agent programmer has to generate *proxy classes* manually by using the Grasshopper *stub generator* (see Section 9.2.1). In both cases, the *usage* of the proxies, i.e., the imple-

mentation of the client agent, is exactly the same.

### 9.2.1 Manual Proxy Generation

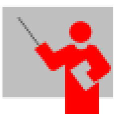
**Stubgen**

If a Java release prior to JDK 1.3 is used for running Grasshopper, *proxy classes* have to be created and made accessible for the client agent. For this purpose, Grasshopper provides a *stub generator*, realized in terms of a batch/shell script named `Stubgen`.

In order to use the stub generator, please perform the following steps:

1. Implement your server interface.
2. Compile the server interface.
3. Invoke the stub generator by providing the class file of the server interface as parameter of the `Stubgen` batch/shell script. Type in the *full qualified class name*, i.e., the classname prefixed by the complete package structure. Single package names have to be separated by a dot („.“) character. Avoid the suffix „.class“ at the end of the class name.

   Earlier releases of Grasshopper required the *class of the server agent* as input, and the generated proxy contained all public methods of this server class. The introduction of server interfaces has been realized in Grasshopper 2.0 in order to enable agent programmers to distinguish between agent-internal public methods and those public methods that are to be accessible via the communication service.
   The stub generator produces a Java source file with the same name as the server interface, suffixed with the letter 'P' (indicating proxy classes).
4. Compile the generated proxy source file.
5. Insert the compiled proxy class file into your Java classpath.



Considering the server agent described in Section 9.4.1, the creation of a server proxy can be achieved in the following way:

1. Compile the source file of the server interface, i.e., the file `IServerAgent.java`. The output will be a Java class file named `IServerAgent.class`.
2. Invoke the stub generator by using the full qualified class name of the server interface as parameter:

   ```
   Stubgen examples.simpleCom.IServerAgent
   ```

   The result will be a Java source file named `IServerAgentP.java`. This is the source file of the server proxy.

3. Compile the generated source file. The result will be a Java class file named `IServerAgentP.class`.

   If you discover problems when compiling proxy classes with JDK 1.3, please use the compiler option '`-target classic`'. If you are using JDK 1.2, no problems should occur during the compilation.

   Please note that, as explained in Section 9.2.2, no manual proxy generation via `Stubgen` is required if a JDK 1.3 environment is used. Even if a manually generated proxy class exists, JDK1.3 will not use it.

#### 9.2.1.1 Usage of the Stub Generator

The Grasshopper stub generator is realized as a batch/shell script named `Stubgen`. The following line gives an overview of all supported parameters:

```
Stubgen   [-h|--help] [-classpath <addClasspath>]
          [-d <stubDir>] [--compile] [--noSource]
          <classname>
```

Invoking `Stubgen` without any parameters prints out the list of available parameters, including a short description of their purpose. The same result can be achieved by using the parameter `-h` or `--help`, respectively.

`-classpath <addClasspath>`:

This optional parameter allows the user to add a set of directories to the existing Java CLASSPATH environment setting. `<addClasspath>` has to be substituted by the directory path(s) that is/are to be added.

`-d <stub_dir>`:

This optional parameter allows the user to specify a directory into which the generated Java source file of the proxy class has to be written.

`--compile`:

When this optional parameter is used, the stub generator automatically compiles the generated Java source file of the proxy class. Note that this option only works in a JDK 1.2 environment. In a JDK 1.3 environment, the Java compiler must be explicitly invoked with the option '`-target classic`'.

`--no_source`:

This optional parameter can only be applied together with the `--compile` option. In this case, no source file of the proxy class is generated.

`<classname>`:

This is the only mandatory parameter and must be substituted with the full qualified class name of the server interface class file. Full qualified means

that the complete package structure must be specified where two single package names are separated by a dot characters. The suffix „.class“ has to be avoided.

### 9.2.2 Dynamic Proxy Generation

Due to the enhanced *reflection* capabilities of JDK 1.3, it is possible for a Grasshopper (client) agent to create server proxies dynamically at runtime without the requirement to have access to a previously compiled proxy class, such as the class `IServerAgentP.class` mentioned in Section 9.2.1. This means that the manual proxy class generation, performed by the agent programmer by using the Grasshopper stub generator, is not necessary anymore. Even if a manually generated proxy class is available, Grasshopper will not use it in a JDK 1.3 environment.

Note that the client implementation is independent of the used Java environment, since it does not refer to the manually generated proxy class (i.e., `IServerAgentP.class` in the example below), but instead to the server interface class (i.e., `IServerAgent.class` in the example below). The interface class is required in any case, independently of the used Java environment.

### 9.2.3 Issues of Mixed JDK Environments

If the Grasshopper environment consists of agencies running on JDK1.2 Java Virtual Machines (JVMs) and other agencies running on JDK1.3 JVMs, problems may occur in cases where agents migrate while maintaining a proxy in their data state. A migration from a JDK1.2 agency to a JDK1.3 agency causes no problems. The proxy remains valid after the migration and can be used by the migrated agents for accessing the associated server. Concerning the opposite direction, i.e., a migration from a JDK 1.3 agency to a JDK 1.2 agency, the client agent should exclude all proxies from its data state before migrating. As described in Chapter 6, this can be achieved either by declaring the proxy as transient instance variable or, in case of a non-transient proxy, by setting the proxy to null before the migration. In both cases, the agent has to re-create the proxy after the migration.

## 9.3 Implementing the Client Side

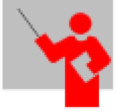
Note that the implementation of a client agent is exactly the same, independent whether the proxy class has been created manually via the stub generator (as described in Section 9.2.1) or whether the server proxy is dynamically generated by the client agent during its runtime, as described in Section 9.2.2. The internal proxy handling is transparently performed inside the `newInstance(...)` method of the class `de.ikv.grasshopper.communication.ProxyGenerator` by analyzing the used Java runtime environment. If a Java release prior to JDK 1.3 is detected, the `newInstance(...)` method automatically tries to access the manually created proxy class (`IServerAgentP.class` in the example, see Section 9.4), while in the case of JDK 1.3 the reflection mechanism is used to generate a proxy dynamically out of the server interface, (i.e., `IServerAgent.class` in the example).

The prerequisites that must be fulfilled by the client agent are:

- access to the server interface class (`IServerAgent.class` in the accompanying example, see Section 9.4)
- access to the server proxy class, *only required if a Java release previous to JDK 1.3 is available and the proxy class thus had to be manually generated via the stub generator* (`IServerAgentP.class` in the accompanying example, see Section 9.4)
- knowledge about the identifier of the server agent (specified either as instance of `de.ikv.grasshopper.type.Identifier` or as instance of `java.lang.String`)
- If the client and server agents are not registered at the same agency domain service and both agents are running in different agencies, the client agent must provide the current location of the server agent (either as instance of `de.ikv.grasshopper.communication.GrasshopperAddress` or as instance of `java.lang.String`). If both agents are running inside the same agency, the location need not be specified even if no agency domain service is running.
- If the client agent wants to invoke the server methods in an asynchronous way, this must be specified by a `byte` parameter, set to `ProxyGenerator.ASYNC`. If synchronous method invocation is to be performed, this additional parameter may be avoided or set to `ProxyGenerator.SYNC`. *Note that a single proxy supports either synchronous or asynchronous method invocation. If a client agent wants to use both mechanisms on the same server, two proxies must be created. Please refer*

*to Section 9.5 for detailed information.*

A client agent creates a server proxy by invoking the `newInstance(...)` method of the class `de.ikv.grasshopper.communication.ProxyGenerator`. Depending on the running Grasshopper environment and the requirements of the client agent, the `newInstance(...)` method can be invoked with different parameters.



Considering the server agent described in Section 9.4.1, the creation of a server proxy can be achieved with the following lines of code:

```
Identifier serverIdentifier = ...;
IServerAgent serverProxy =
  (IServerAgent) ProxyGenerator.newInstance(
                        IServerAgent.class,
                        serverIdentifier);
```

Note that `IServerAgent` represents the interface of the server agent.

The code above requires both client and server agent to be registered at the same agency domain service or to reside inside the same agency, in order to enable the proxy to locate the server agent. If both prerequisites are not fulfilled, the `newInstance(...)` method must be enhanced with the current location of the server agent:

```
String serverLocation = ...;
IServerAgent serverProxy =
  (IServerAgent) ProxyGenerator.newInstance(
                        IServerAgent.class,
                        serverIdentifier,
                        serverLocation);
```

The examples above create a server proxy that supports synchronous communication. If the client agent wants to invoke the server methods asynchronously, this has to be specified by a `byte` variable, as shown below:

```
IServerAgent serverProxy =
  (IServerAgent) ProxyGenerator.newInstance(
                        IServerAgent.class,
                        serverIdentifier,
                        serverLocation,
                        ProxyGenerator.ASYNC);
```

In Grasshopper releases 1.x, a single proxy was able to handle both synchronous and asynchronous method invocation. In these early Grasshopper releases, the stub generator inserted additional methods into the proxy class. The result was a proxy class covering two methods for each public method of the server agent. One of these methods was meant for synchronous communication, and the second one (with an additional parameter for maintaining the asynchronously arriving method result) for asynchronous communication.

In the current Grasshopper release, it is not anymore required to use the stub

generator, supposed that a JDK 1.3 runtime environment is used. In order to enable a client implementation to be independent of the fact whether a manually generated proxy class exists or whether the Java reflection mechanism is used for the dynamic proxy generation, no additional methods are created by the stub generator. *Thus, if a client agent wants to access a server agent synchronously and asynchronously, the client agent has to create two proxies, i.e., one proxy for each communication mechanism*. Please refer to Section 9.5 for detailed information about asynchronous communication.

Note that a proxy object is always created as instance of a server interface and not of the corresponding server class. Do not try to convert the proxy down to the server class (neither when creating the proxy nor when invoking a method on the proxy), since this will raise a `ClassCastException`. Concerning the example in Section 9.4, the interface `IServerAgent` has to be used for proxy-related stuff, and not the class `ServerAgent`.

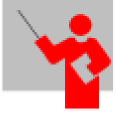
## 9.4 Simple Communication Scenario

The following scenario consists of the following three classes/interfaces, covered by the package `examples.simpleCom`:

- `ServerAgent` (see Example 7 in Section 9.4.1): An agent that provides one method to the communication service. By means of this method, a (remote) client can order the ServerAgent to migrate to another location.
- `IServerAgent` (see Example 8 in Section 9.4.1): The server interface that contains the method which has to be accessible for client agents. This interface is the basis for the generation of server proxies.
- `ClientAgent` (see Example 9 in Section 9.4.2): The agent that invokes the accessible method on the ServerAgent. In the context of this scenario, the ClientAgent remains at its initial location and in this way realizes a stationary user interface for the ServerAgent. By means of the ClientAgent's GUI, a user can move the ServerAgent remotely from one agency to another.

### 9.4.1 Example: ServerAgent

The agent shown in Example 7 provides the public method `go(...)` to the communication service. When this method is invoked, the agent tries to migrate to the location that has been provided as method parameter.

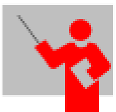


Example 7: ServerAgent

```
package examples.simpleCom;

import de.ikv.grasshopper.agent.*;
import
de.ikv.grasshopper.communication.GrasshopperAddress;

// This class realizes the server agent of the simple
// communication scenario.
public class ServerAgent  extends MobileAgent
                          implements IServerAgent
{
  public String getName() {
    return "ServerAgent";
  }

  // This method is accessible via the communication
  // service.
  public void go(String location) {
    log("Roger, moving to " + location);
    try {
      move(new GrasshopperAddress(location));
    }
    catch (Exception e) {
      log("Migration failed. Exception = ", e);
    }
  }

  public void live() {
    log("ready.");
  }
}
```

In order to make the `go(...)` method accessible via the communication service, a *server interface* has to be defined that contains the method. This server interface (see Example 8) must be implemented by the server agent.

Note: As explained in Section 9.2, a special proxy class has to be created with the Grasshopper stub generator if a Java runtime environment previous to JDK 1.3 is used. If JDK 1.3 is used, this step is not required, since in this case the proxy is dynamically created by using the Java reflection mechanism.



Example 8: IServerAgent

```
package examples.simpleCom;
```

```
public interface IServerAgent
{
  public void go(String location);
}
```

Note: If the ServerAgent has to provide also the methods of its superclasses, the interface `IMobileAgent` has to be extended by the interface `IServerAgent`.

A description about how to run the example is given in Section 9.4.3.

### 9.4.2 Example: ClientAgent

The agent below acts as a client that uses the communication service in order to access the server agent introduced in Section 9.4.1.

Inside its `init(...)` method, the `ClientAgent` creates an instance of the `ServerAgent` as well as a server proxy, i.e., an instance of the server interface `IServerAgent` (see Example 8). After this, the client agent is able to invoke the server agent's `go(...)` method.

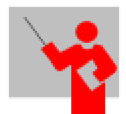
Note that the client uses a second proxy, i.e., the proxy of the local agency, in order to create the server agent. The local agency proxy is available for each agent via the method `getAgentSystem()` of the agent's superclass `Agent`. Detailed information about possible interactions between agents and their local agency is given in Section 9.11.

The created server proxy is meant for synchronous communication, since the parameter `ProxyGenerator.ASYNC` mentioned in Section 9.3 is not set. The client just specifies the server interface as well as the server identifier.

A running agency domain service is required for running the example, since the client moves the server to other locations. In order to maintain the connection to the server, the proxy has to contact the agency domain service for requesting the server's new location. This is done transparently for the client.

The client agent creates a GUI in order to ask the user for a new location for the server agent. After the user has pressed the OK button, the client agent invokes the `go(...)` method of the server proxy, transmitting the previously specified location via the communication service to the server agent.

Example 9: ClientAgent



```
package examples.simpleCom;

import de.ikv.grasshopper.agent.*;
```

```
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.communication.*;
import javax.swing.*;
import java.awt.*;

// This class realizes the client agent of the simple
// communication scenario.
public class ClientAgent extends MobileAgent
{
  // Proxy of local agency = transient
  // (i.e., not part of the data state),
  // since it becomes invalid if the agent moves
  // to another location.
  transient IAgentSystem agencyProxy;

  // Data state of the agent, since not transient.
  // AgentInfo serverInfo;
  IServerAgent serverProxy;

  public void init(Object[] creationArgs) {
    // Get proxy of local agency.
    agencyProxy = getAgentSystem();
    // Create the server agent.
    try {
      serverInfo =
        agencyProxy.createAgent(
          "examples.simpleCom.ServerAgent",
          getInfo().getCodebase(),
          "InformationDesk", null);
    }
    catch (AgentCreationFailedException e) {
      log("Creation of server agent failed.");
    }
    // Create proxy of the server agent.
    if (serverInfo != null)
      serverProxy = (IServerAgent)
        ProxyGenerator.newInstance(
          IServerAgent.class,
          serverInfo.getIdentifier());
  }

  public String getName() {
    return "ClientAgent";
  }

  public void action() {
    live();
```

```
    }

    // This method requests user input via a graphical
    // component.
    // The user has to specify the new location to which
    // the ServerAgent shall migrate.
    public String requestLocation() {
      String location = null;
      log("Request location");
      location = JOptionPane.showInputDialog(null,
          "Where shall I send the server?");
      log("Moving the server to " + location);
      return location;
    }

    public void live() {
      String location;
      log("Starting life");
      location = requestLocation();
      while (location != null) {
        // Invoke method on server agent via proxy.
        serverProxy.go(location);
        location = requestLocation();
      }
    }
  }
```

A description about how to run the example is given in Section 9.4.3.

### 9.4.3 Running the Scenario

This section explains how to run the communication example whose parts (i.e., `ClientAgent` and `ServerAgent`) have been introduced in the previous sections.

Requirements:

- A running agency domain service. Note that this service has to be started before the agencies, and its address has to be specified when starting the agencies in order to register them. Please refer to the User's Guide for more information about how to start agencies and agency domain services.
- At least two running agencies
  *Since the ClientAgent creates an own GUI that may block the agency GUI, it is recommended that you do not activate the agency GUI. Instead, start the agencies just with their textual interface (command option -tui). Please*

*refer to the paragraphs titled „Running the Examples" at the beginning of Chapter 2 in order to get a detailed explanation about the possibly occurring problems.*

- If you are using a JDK 1.2 environment, you must have generated a proxy class (named `IServerAgentP`) by invoking the Grasshopper stub generator with the interface class `IServerAgent` as input parameter. The file `IServerAgentP.class` should be stored either in a directory belonging to the Java classpath or in the code base directory of the Client-Agent. In a JDK 1.3 environment, this class is not needed. Even if it is available, it will not be used. Instead, the proxy is dynamically generated by the ClientAgent at runtime.

<u>Running the Example:</u>

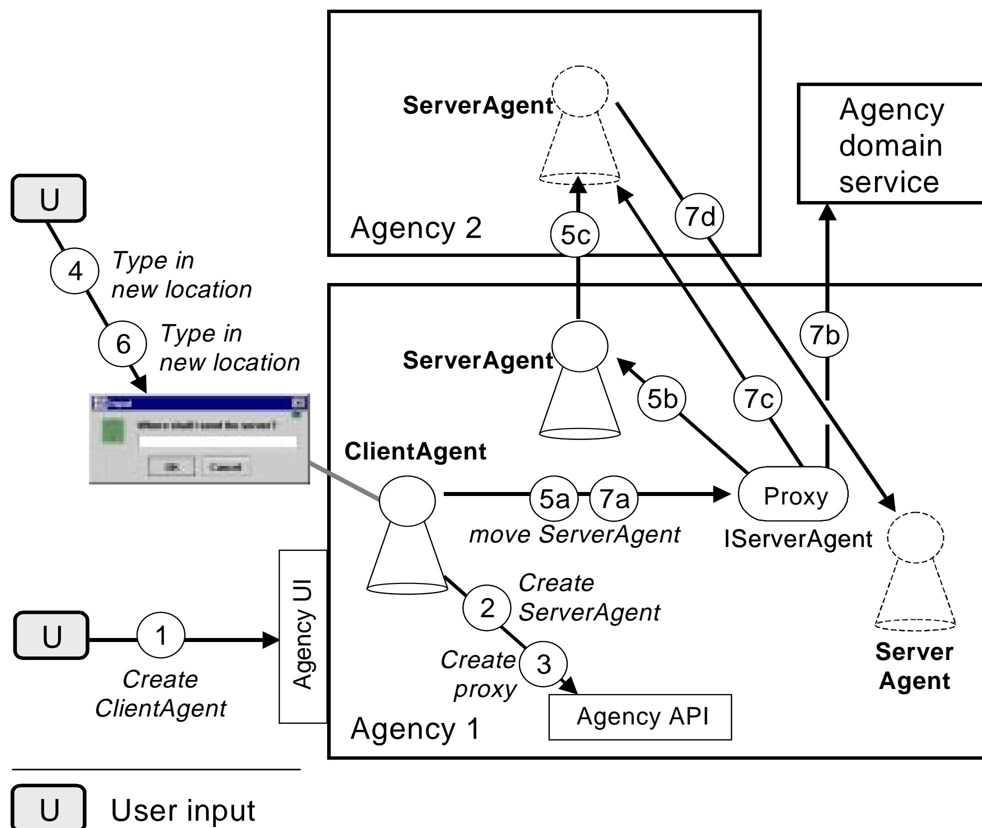



Figure 10: Simple Communication Scenario

Create the `ClientAgent` inside one of the running agencies via the agency's UI (1).

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.simpleCom.ClientAgent
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

At first, the client agent creates an instance of the server agent (2) and establishes a connection to this agent via a server proxy (3). The server agent will appear in the agency's GUI. (If you are using the TUI, please use the `l[ist]` command in order to list all agents running inside the agency. One of them should be the server agent.)

Use the client agent's GUI to type in the new location to which the server agent has to migrate (4). Note that, since an agency domain service is running (and connected with both agencies!), it is sufficient to specify the host name, agency name, and (optionally) a place name of the desired destination. As explained in Section 5.4, the required (simplified) location format is `<hostName>/<agencyName>/<placeName>` or `<hostName>/<agencyName>`. In the latter case, the agent will migrate to the default place `Informationdesk` of the destination agency.

After pressing the OK button, the ClientAgent uses the server proxy to trigger the ServerAgent's migration to the specified location (5).

**Location-transparent communication**

Note that the client does not know that the server agent has changed its location. Thus, after you have specified a new server location for the second time and pressed OK (6), the ClientAgent again contacts the server proxy (7a) which tries to contact the server agent at its former location. After noticing that the server agent is not accessible anymore, the proxy automatically contacts the agency domain service in order to determine the server agent's new location (7b). Since each agency automatically registers all hosted agents at the agency domain service (if existent), this service is aware of the locations of all agents at any time. In this way, the re-establishment of the connection between the proxy and the server agent is performed completely transparent for the client agent. The proxy forwards the migration order from the ClientAgent to the new location of the ServerAgent (7c), and the ServerAgent migrates to its new destination (7d).

### 9.4.4 Summary

- The purpose of the Grasshopper communication service is to enable local and remote interactions between Grasshopper components (agents, agencies, region registries).
- A communication session is initiated by a client (agent) that establishes a

connection to a server (agent) via a so called proxy object or proxy.

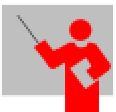
- A Grasshopper proxy (object) represents the server at the client side. A client always contacts a proxy locally, and the proxy in turn contacts the (locally or remotely residing) server. A proxy is based on a Java interface that defines those server methods which are to be accessible via the communication service.
- Concerning JDK 1.2 environments, a proxy object has to be generated by the programmer by using the Grasshopper stub generator. In contrast to this, JDK 1.3 environments allow a dynamic proxy generation during the runtime of the communicating components.

## 9.5 Sync. vs. Async. Communication

As described in Section 9.3, two different server proxies have to be created by a client agent, depending on whether the client agent wants to communicate with the corresponding server agent in a synchronous or asynchronous way.

Synchronous communication means that, after invoking a method of a server agent via its proxy, the client agent is blocked until the invoked method returns. In contrast to this, an asynchronously invoked method does not block the invoking client.

The decision between synchronous and asynchronous communication is made when the client agent creates a server proxy by invoking the `newInstance(...)` method of the class `de.ikv.grasshopper.communication.ProxyGenerator`. The default communication mode is synchronous communication. If a proxy has to support asynchronous communication, this has to be specified by means of an additional [`byte`] parameter.



- Creation of a proxy for synchronous communication. This is the default mode, i.e., no specification of the communication mode is required:

```
IServerAgent serverProxy =
(IServerAgent) ProxyGenerator.newInstance(
                    IServerAgent.class,
                    serverIdentifier);
```

- Creation of a proxy for synchronous communication with (optional) specification of the communication mode:

```
IServerAgent serverProxy =
(IServerAgent) ProxyGenerator.newInstance(
                    IServerAgent.class,
                    serverIdentifier,
                    ProxyGenerator.SYNC);
```

- Creation of a proxy for asynchronous communication. The communication mode must be specified:

```
IServerAgent serverProxy =
(IServerAgent) ProxyGenerator.newInstance(
                         IServerAgent.class,
                         serverIdentifier,
                         ProxyGenerator.ASYNC);
```

More information about the usage of the `newInstance(...)` method is provided in Section 9.3.

### 9.5.1 Asynchronous Provision of Results

Asynchronously invoked methods do not block the invoking client. That means, the client may continue its task after invoking the method, while the corresponding server performs the invoked method in parallel. This does not cause any problems for the client if the invoked server method neither returns a result nor throws an exception. However, if a result or an exception is to be transmitted by the server, the client must have the possibility to receive it.

**FutureResult**

For this purpose, a proxy that supports asynchronous communication implements the interface `de.ikv.grasshopper.communication.IFutureResult` which provides a method named `getFutureResult()`. This method, invoked by the client, returns an instance of the class `de.ikv.grasshopper.communication.FutureResult` which represents an intermediate storage for asynchronously arriving results. That means, when an asynchronously invoked method returns a result in terms of a return value or an exception, this result is transmitted to and maintained by the `FutureResult` object of the server proxy. The client may perform its own task in parallel without being influenced by the interactions between the server and its proxy.

When needed, the client can access the `FutureResult` object via the following methods:

- `getResult()`: This method returns the result that the server agent has sent to the proxy. If no result is available, the method waits for a result for a certain period of time which can be set via the `setTimeout(...)` method. If the server agent does not deliver a result to the proxy within the specified timeout period, a `de.ikv.grasshopper.communication.AsyncTimeoutException` is thrown by the method, and the client can continue its task.

**Return types derived from Object**

The default return type of the getResult() method is java.lang.Object. If the corresponding server method provides a return type that is derived from java.lang.Object, the client can retrieve the server's return value simply by converting java.lang.Object to the concrete return type of the server method, such as java.lang.String.

**Simple return types**

If a server method returns a primitive data type that is not derived from java.lang.Object, such as int, this return type cannot be retrieved directly from the getResult() method via converting. That means, the return type int of a server method can only be retrieved as Integer, since Integer is derived from java.lang.Object. An additional cast is required in order to get the primitive data type, e.g., by invoking a method like intValue(). Another possibility for retrieving primitive data type values is to use one of the methods get<Type>Result(). These methods are provided in order to simplify the retrieval of non-Object values.

**Exception handling**

If the asynchronously invoked server method throws an exception, this exception is forwarded to the getResult() method. Thus, the getResult() method has to catch all exceptions that may be thrown by the corresponding server method, as well as the Throwable exception in order to handle failures that may be associated with communication or platform related problems.

- get<Type>Result(): Several specialized getResult() methods are defined that provide primitive data types as return types. For instance, return values of the type double can be retrieved by invoking the method getDoubleResult(). Those specialized methods are defined for the following return types: <Type> = boolean, byte, char, double, float, int, long, short. If no result is available, the method waits for a result for a certain period of time which can be set via the setTimeout(...) method. If the server agent does not deliver a result to the proxy within the specified timeout period, a de.ikv.grasshopper.communication.AsyncTimeoutException is thrown by the method, and the client can continue its task.

**Exception handling**

If the asynchronously invoked server method throws an exception, this exception is forwarded to the getResult() method. Thus, the get<Type>Result() method has to catch all exceptions that may be thrown by the corresponding server method, as well as the Throwable exception in order to handle failures that may be associated with communication or system related problems.

- getTimeout(): If a get<...>Result() method is invoked before

a result is available, the method blocks for a certain period of time, waiting for the result. The `getTimeout()` method returns the blocking period that is specified for the `get<...>Result()` methods. The blocking period is given in milliseconds, and the default value is one minute. If the `get<...>Result()` method does not return a result during the specified timeout period, a `de.ikv.grasshopper.communication.AsyncTimeoutException` is thrown by the method, and the client can continue its task.

- `setTimeout(...)`: This method allows the client to specify the blocking period for the get<...>Result() methods. If a `get<...>Result()` method is invoked before a result is available, the method blocks for the specified period of time, waiting for the result. The blocking period is given in milliseconds, and the default blocking period is one minute. A value of '0' defines an infinite blocking period. If the `get<...>Result()` method does not return a result during the specified timeout period, a `de.ikv.grasshopper.communication.AsyncTimeoutException` is thrown by the method, and the client can continue its task.
- `isAvailable()`: This method informs the client whether an asynchronously invoked method has already returned a result or not. In contrast to the `get<...>Result()` methods, this method is non-blocking. If no result is available, the method returns at once.
- `isUserException()`: In case the method invocation ends with an exception (thrown by the `get<...>Result()` method), the `isUserException()` method indicates whether the exception has been thrown by the server agent (`true`) or due to other errors that may be associated with communication or system failures (`false`).
- `addResultListener(...)`: By adding a result listener to the `FutureResult` object, a client is automatically informed about incoming results from a server. If a result arrives, the method `resultHasArrived(...)` of the associated listener is invoked. Inside this method, the client can react on the result. For more information, please refer to the paragraphs on page 96 which are titled *3. notification*.
- `removeResultListener(...)`: This method removes an attached result listener from the `FutureResult` object.

The *only possibility* for a client to retrieve the result of an asynchronously invoked method is to invoke a `get<...>Result()` method of the `FutureResult` object that is associated with the server proxy. An asynchronous method call like

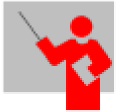
```
result = asyncServerProxy.serverMethod();
```

will *not* initialize the `result` variable! This is the reason why this method is called without allocating a return variable in the examples below. In contrast to this, the result of a synchronous method invocation can of course be retrieved in this way.

By using the `FutureResult` methods, a client can realize three different mechanisms for accessing an asynchronously arriving result:

**Blocking**

1. *Blocking*
   After the client has asynchronously invoked a server method, it continues its task, while the server performs the invoked method in parallel. At a certain point of execution, the client may require the result of the invoked method in order to continue. In this case, the client can set the `FutureResult` timeout to infinite (value = 0), and call one of the `get<...>Result()` methods.



```
// Invoke server method via asynchronous proxy.
try {
  asyncServerProxy.serverMethod();
catch ... // server & other exception
// Get FutureResult object
FutureResult futureResult =
  ((IFutureResult) asyncServerProxy).\\
    getFutureResult();
// Client performs its task.
...
// Client needs the result.
// Set infinite timeout
futureResult.setTimeout(0);
try {
  int result = futureResult.getIntResult();
} catch ... // server & other exceptions
...
```

**Exception handling is required two times**

The example code above contains two try/catch blocks. The first block may be surprising to you, since an asynchronous method call does neither directly return a result nor throw an exception. The reason for the need of this try/catch block is that Grasshopper generates proxies via the Java reflection mechanism. This mechanism uses the server interface as input for the proxy generation. If exceptions are specified inside this interface, these exceptions are automatically adopted by the proxy.

After each asynchronous method call, a new `FutureResult` object is created by the proxy. This object can be retrieved by the client by invoking the `getFutureResult()` method.

Note: It is required for the client to invoke the `getFutureResult()` method directly after performing an asynchronous method call. The reason is that the proxy only maintains a single `FutureResult` instance. Thus, if the client performs several asynchronous calls on the same proxy without requesting the `FutureResult` object after each call, the proxy only maintains the `FutureResult` object of the latest method call, and all previously created `FutureResult` objects are lost.

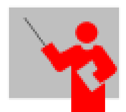
In the example code above, the client sets an infinite timeout for the retrieved `FutureResult` object. In this way, the subsequently invoked `getIntResult()` method will block until the server method has returned (either with a return value or an exception).

Usually, the client does not know if the server is still working on the invoked method, or if something unexpected has happened, such as a system crash on the server side. Thus, it is recommended for the client not to wait to the end of time, but to set a finite timeout that may be a bit longer than the expected duration of the server's method performance. If the timeout period is over and the server method has still not returned, a `de.ikv.grasshopper.communication.AsyncTimeoutException` is thrown by the `FutureResult` object.

As shown in the example code above, the second try/catch block is placed around the `getIntResult()` method. The purpose of this method is to transmit the server result to the client, either in terms of a return value or an exception. Thus, this method has to be handled in the same way as the real server method (i.e., `serverMethod()`) is handled in case of a synchronous invocation.

**Polling**

2. *Polling*
If the client is able to perform some tasks while waiting for the result of an asynchronously invoked server method, the client can periodically check whether a result has arrived or not. For this purpose, the `FutureResult` object provides the non-blocking method `isAvailable()` that returns a boolean value. If this method returns `true`, the client can retrieve the result via the `getIntResult()` method (which will not block in this case, independent of the defined timeout period, since the client has assured that the result has already arrived).



```
// Invoke server method via asynchronous proxy.
try {
  asyncServerProxy.serverMethod();
} catch ... // server & other exceptions
// Get FutureResult object
FutureResult futureResult =
  ((IFutureResult) asyncServerProxy).\\
```

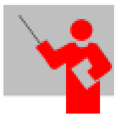
```
    getFutureResult();
// Start polling
while(!futureResult.isAvailable()) {
  // Client performs its task.
...
}
// Now the result is available.
try {
  int result = futureResult.getIntResult();
} catch ... // server & other exceptions
...
```

The first eight lines of this example code have already been explained above.

Concerning the example, the return value of the method `isAvailable()` is used for evaluating the condition of a `while` loop. The loop ends if the server method has returned, i.e., if the method `isAvailable()` returns the value `true`. After this, the client retrieves the result via `getIntResult()`.

**Notification**

3. *Notification*
   Beside the possibilities to perform a blocking call or to periodically check whether a result is available, a client can order to be notified when the called server method returns. This is achieved by adding a result listener to the `FutureResult` object:



```
// Instantiate listener
Listener listener = new Listener();
// Invoke server method via asynchronous proxy.
try {
  asyncServerProxy.serverMethod();
} catch ... // server & other exceptions
// Get FutureResult object
futureResult =
  ((IFutureResult) asyncServerProxy).\\
  getFutureResult();
// Add result listener
futureResult.addResultListener(listener);
index++;
```

The corresponding `Listener` class has to implement the `de.ikv.grasshopper.communication.ResultListener` interface. This interface defines the method `resultHasArrived(...)` which is called by the proxy when the asynchronous server method has returned.

```
class Listener implements ResultListener {
  public void resultHasArrived(ResultEvent e){
```

```
      // Get FutureResult object
      FutureResult fResult =
        (FutureResult) e.getSource();
      try {
        int result = fResult.getIntResult();
      } catch ... // server & other exceptions
      ...
    }
  }
```

Note that a client may initiate a listener-based call without waiting for the result of a previously performed call. In this case, a client must be able to associate an incoming result with the corresponding method call. Concerning the example above where the existence of only one `Listener` object is assumed, this can be achieved by creating a new `FutureResult` object for each method call. Inside the method `resultHasArrived(...)`, the `FutureResult` object that is associated with the result event can be compared with the `FutureResult` objects that are associated with the single method calls.

In some cases, a client may want to migrate to another location before the result of a previously initiated asynchronous call has arrived. In this case, the client agent has to add itself as listener to the `FutureResult` object. In order to do this, the client agent has to implement the `ResultListener` interface. If the result listener is not realized by the client agent class itself, incoming results will be lost if the client migrates. Please refer to Section 9.14 for more information about migrating clients and servers.

## 9.6 Asynchronous Communication Scenario

The example scenario for asynchronous communication consists of four classes/interfaces, covered by the package `examples.asyncCom`:

- `AsyncServerAgent` (see Example 10 in Section 9.6.1): An agent that provides one method to the communication service. For each single method call, the user can decide, whether the result is to be a regular return value or an exception. This is to show how the exception handling of asynchronously invoked methods can be achieved.
- `IAsyncServerAgent` (see Example 11 in Section 9.6.1): The server interface that contains the method which has to be accessible for the client agent. This interface is the basis for the generation of server proxies.
- `AsyncServerException` (see Example 12 in Section 9.6.1): This exception may be thrown by the server agent's accessible method.

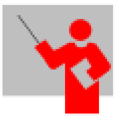
- `AsyncClientAgent` (see Example 13 in Section 9.6.2): The client agent that invokes the accessible method of the server. The user can decide between four invocation mechanisms: *synchronous*, *asynchronous blocking*, *asynchronous polling*, or *asynchronous notification*.

### 9.6.1 Example: AsyncServerAgent

The AsyncServerAgent implements the interface IAsyncServerAgent. This interface contains the method `requestConfirmation()` that is accessible via the communication service. When this method is invoked, a dialog appears, asking the user whether the method has to terminate regularly, or whether an exception is to be thrown. The purpose is to show how a client agent retrieves asynchronously arriving return values and how it handles exceptions.

In order to enable the user to associate a specific method call at the server side with the corresponding result arrival at the client side, the server agent increments the result value each time the `requestConfirmation()` method is invoked.

The source code of the corresponding client agent is described in Section 9.6.2.



Example 10: AsyncServerAgent

```
package examples.asyncCom;

import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.communication.*;
import javax.swing.*;
import java.awt.*;

// This class realizes the server agent of the async.
// communication scenario.
public class AsyncServerAgent extends MobileAgent
                    implements IAsyncServerAgent
{
  int result;

  // No creation arguments are required.
  public void init(Object[] creationArgs) {
    result = 1;
  }

  public String getName() {
```

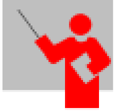
```
      return "AsyncServerAgent";
    }

    // This method requests user input via a graphical
    // component.
    // The user has to decide whether the server method
    // (invoked by the AsyncClientAgent) shall return
    // regularly or throw an exception.
    public int requestConfirmation()
      throws AsyncServerException {

      int yesOrNo = 0;

      result++;
      log("Client request arrived. Result will be = " +
        result);
      yesOrNo = JOptionPane.showConfirmDialog(
        null,
        "Throw exception to client?",
        "AsyncServerAgent",
        JOptionPane.YES_NO_OPTION);
      if (yesOrNo == JOptionPane.YES_OPTION) {
        log("No result. Throwing exception instead!");
        throw new AsyncServerException();
      }
      else
        return result;
    }

    public void live() {
      log("ready.");
    }
  }
```

The server interface just contains the method `requestConfirmation()`.

Note: In Section 9.5.1 it has been mentioned that, concerning asynchronous calls, a client has to perform an exception handling twice: at first when the server method is called, and the second time when the result is retrieved via one of the `get<...>Result()` methods. The reason for the first exception handling is that the server proxy is dynamically generated out of the server interface by using the Java reflection mechanism. Since all server exceptions are defined in the server interface (as shown in Example 11), the reflection mechanism creates a proxy that expects an exception handling. However, actually neither a server exception nor a return value can appear directly when a server method is invoked in an asynchronous way. Instead, both results arrive at the

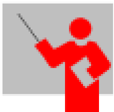
client side only when the client invokes a `get<...>Result()` method.

Note: As explained in Section 9.2, a special proxy class has to be created with the Grasshopper stub generator if a Java runtime environment previous to JDK 1.3 is used. If JDK 1.3 is used, this step is not required, since in this case the proxy is dynamically created by using the Java reflection mechanism.



Example 11: IAsyncServerAgent

```
package examples.asyncCom;

public interface IAsyncServerAgent
{
  public int requestConfirmation()
    throws AsyncServerException;
}
```



Example 12: AsyncServerException

```
package examples.asyncCom;

public class AsyncServerException extends Exception
{
  public AsyncServerException() {}
}
```

A description about how to run the example is given in Section 9.6.3.

### 9.6.2 Example: AsyncClientAgent

The AsyncClientAgent maintains the following instance variables:

- `regionProxy`: A proxy of the local agency, instantiated from the class `IRegion`. In contrast to the agency interface `IAgentSystem` that enables a client to look for agents inside the local agency, the interface `IRegion` provides access to the agency domain service and thus enables a client to look for agents (and agencies) inside a whole region/domain. Note that this proxy has to be transient in order to enable the agent to migrate.
- `syncServerProxy`: A variable maintaining a server proxy (i.e., an instance of `IAsyncServerProxy`) that is able to handle synchronous communication (referred to as *synchronous proxy* in the following paragraphs). This variable is created by setting the communication mode parameter to `ProxyGenerator.SYNC`. Since this variable is not

declared transient, it represents a part of the agent's data state and is maintained by the agent when the agent migrates.

- `asyncServerProxy`: A variable maintaining a server proxy (i.e., an instance of `IAsyncServerProxy`) that is able to handle asynchronous communication (referred to as *asynchronous proxy* in the following paragraphs). This variable is created by setting the communication mode parameter to `ProxyGenerator.ASYNC`. Since this variable is not declared transient, it represents a part of the agent's data state and is maintained by the agent when the agent migrates.
- `futureResult`: A variable for handling asynchronously arriving server results, i.e., return values or exceptions. Since this variable is not declared transient, it represents a part of the agent's data state and is maintained by the agent when the agent migrates.

The following paragraphs describe the functionality of each client method.

`init(Object[] creationArgs)`

**Get agency proxy**

In its `init(...)` method, the agent requests the `IRegion` interface of the local agency in order to look for the AsyncServerAgent. The interface is retrieved via the method `getRegion()` which is provided by the agent's superclass `Agent`.

**Look for server agent**

In order to find the AsyncServerAgent, the client sets a search filter by specifying the server agent's name, and invokes the `listAgents(...)` method on the `IRegion` interface. Invoking this method orders the local agency to contact an agency domain service in order to look for all agents matching the specified filter. All agencies that are registered at the contacted agency domain service are included in the search. Detailed information about functionality associated with the interface `IRegion` is given in Section 9.12.1.

The reason for the client to look for the server agent is that the client needs the server's identifier in order to create corresponding server proxies. The identifier is part of the `AgentInfo` object that is returned by the method `listAgents(...)`. (A detailed list of all components of `AgentInfo` is given in Chapter 5.) Note that the method `listAgents(...)` returns a set of `AgentInfos`, representing the set of all agents that match the specified filter criteria. The client agent simply selects the first agent from the list.

**Proxy generation**

The client agent creates two proxies of the server: one for synchronous and one for asynchronous communication. For this purpose, the client specifies the server interface, the server identifier, and a `[byte]` variable that defines the communication mode (SYNC or ASYNC). No provision of the

server location is needed, since an agency domain service is available. The proxy is able to locate the server agent transparently for the client by automatically contacting the agency domain service.

`requestCommunicationMode()`

This method, called from inside the agent's `live()` method, activates the client's GUI that enables the user to select between the following communication modes:

- synchronous communication
  (method `synchronousInvocation()`)
- asynchronous blocking communication
  (method `blockingResultHandling()`)
- asynchronous polling communication
  (method `pollingResultHandling()`)
- asynchronous notification-based communication
  (method `notificationResultHandling()`)

Concerning the invocation of the server method, the only difference between synchronous and asynchronous calls is that the synchronous call directly returns a result, while the result of asynchronous calls has to be requested by invoking a `get<...>Reault()` method.

There are no differences between the blocking, polling, and notification based server method calls. Concerning asynchronous calls, differences only exist in handling the result after invoking the method.

In order to keep the code as short as possible, the required exception handling is performed only once inside the `live()` method instead of inside each single method.

`synchronousInvocation()`

The synchronous method invocation is already known from the previous communication scenario, described in Section 9.4. The client uses the synchronous proxy for invoking the server method `requestConfirmation()`. After the invocation, the client is blocked until the server method returns.

`blockingResultHandling()`

In this method, the client uses the asynchronous proxy for invoking the server method. After requesting the `FutureResult` object, the client is free to perform its own task, represented by the `for` loop. After performing the loop, it is assumed that the client needs the result of the server method. Thus, the client sets the timeout to an appropriate value (the value

'0' represents an infinite timeout) and requests the server result by calling `getIntResult()`. If the result does not arrive within the specified timeout period, a `de.ikv.grasshopper.communication.AsyncTimeoutException` will be thrown. This mechanism is meant to prevent the client from waiting till the end of time if for instance the server or its hosting system has crashed.

`pollingResultHandling()`

After invoking the server method and requesting the `FutureResult` object, the client continues its own task, represented by a `while` loop. Inside this loop, the client periodically checks whether the server method has returned. If this is true, the client requests the result by invoking the proxy's `getIntResult()` method.

`notificationResultHandling()`

After invoking the server method and requesting the `FutureResult` object, the client adds a result listener to the proxy. Note that in the given example the client agent itself represents the result listener by implementing the `ResultListener` interface. (Another possibility would have been to define a separate listener class.)

After adding the listener, the client does not have to care about polling for the result. Instead, the client's method `resultHasArrived(...)` is automatically invoked by the proxy when a result has arrived.

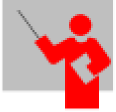
`live()`

The `live()` method of the client agent has two purposes:

- to delegate the invocation of the server method to that client method which corresponds to the communication mode selected by the user (via the GUI)
- to perform the exception handling that is required by the asynchronous invocation.

`resultHasArrived(ResultEvent e)`

If the client has added itself as listener to the `FutureResult` object (see method `notificationResultHandling()` above), the method `resultHasArrived(...)` is automatically called by the proxy as soon as the previously invoked server method has returned. In this case, the result handling is performed in the usual way, i.e., by invoking the `getIntResult()` method. Note that the `FutureResult` object on which the `getIntResult()` method is invoked is associated with the listener event (see statement `fResult = (FutureResult) e.getSource();`).



Example 13: AsyncClientAgent

```
package examples.asyncCom;

import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.communication.*;
import de.ikv.grasshopper.util.*;
import javax.swing.*;
import java.awt.*;

// This class realizes the client agent of the async.
// communication scenario.
public class AsyncClientAgent extends MobileAgent
                  implements ResultListener
{
  // Proxy of local agency = transient
  // (i.e., not part of the data state),
  // since it is not serializable.
  // A non-transient agency proxy would not
  // allow the agent to migrate.
  transient IRegion regionProxy;

  // Data state of the agent, since not transient
  IAsyncServerAgent syncServerProxy;
  IAsyncServerAgent asyncServerProxy;
  FutureResult futureResult;

  // No creation arguments are required.
  public void init(Object[] creationArgs) {
    AgentInfo[] serverInfos;

    // Get proxy of local agency
    regionProxy = getRegion();
    // Look for the server agent in the
    // agency domain service
    SearchFilter filter = new
      SearchFilter(
        SearchFilter.NAME+"=AsyncServerAgent");
    serverInfos =
      regionProxy.listAgents(null, filter);
    // Create proxies of the server agent
    // (One for sync. and one for async. communication)
    if (serverInfos != null) {
      syncServerProxy = (IAsyncServerAgent)
        ProxyGenerator.newInstance(
```

```
          IAsyncServerAgent.class,
          serverInfos[0].getIdentifier().toString(),
          ProxyGenerator.SYNC);
      asyncServerProxy = (IAsyncServerAgent)
        ProxyGenerator.newInstance(
          IAsyncServerAgent.class,
          serverInfos[0].getIdentifier().toString(),
          ProxyGenerator.ASYNC);
  }
}

public String getName() {
  return "AsyncClientAgent";
}

public void action() {
  live();
}

// This method requests user input via a graphical
// component.
// The user has to select one of the communication
// modes:
//   - synchronous
//   - asynchronous blocking
//   - asynchronous polling
//   - asynchronous notification-based
public String requestCommunicationMode() {
  String comMode = null;
  String options[] = {
    "sync.",
    "async. blocking",
    "async. polling",
    "async. notification"};
  comMode = (String)
    JOptionPane.showInputDialog(
      null, "Communication mode:",
      "AsyncClientAgent",
      JOptionPane.QUESTION_MESSAGE, null, options,
      options[0]);
  return comMode;
}

// This method performs a synchronous invocation of
// the method 'requestConfirmation',
// provided by the AsyncServerAgent.
public int synchronousInvocation() throws Throwable {
  int result = -1;
```

```
    log("Starting synchronous call");
    // Invoke server method synchronously
    // by using the sync. server proxy
    result = syncServerProxy.requestConfirmation();
    return result;
  }

  // This method performs an asynchronous blocking
  // invocation of the method
  // 'requestConfirmation', provided by the
  // AsyncServerAgent.
  public int blockingResultHandling()
                throws Throwable {
    int result = -1;

    log("Starting blocking call");
    // Invoke server method asynchronously
    // by using the async. server proxy
    asyncServerProxy.requestConfirmation();
    // Get futureResult object from the proxy
    futureResult = ((IFutureResult)
      asyncServerProxy).getFutureResult();
    // Perform some task until the result is required.
    for (int i = 0; i < 20; i++)
      log("I'm doing something serious!");
    log("Now I need the server's result.");
    // Set timeout of 10 seconds
    futureResult.setTimeout(10000);
    // getIntResult() will block until the server
    // method returns or until the timeout is over.
    log("Waiting for the result for 10 seconds");
    result = futureResult.getIntResult();
    log("Waiting time is over");
    return result;
  }

  // This method performs an asynchronous polling
  // invocation of the method
  // 'requestConfirmation', provided by the
  // AsyncServerAgent.
  public int pollingResultHandling() throws Throwable {
    int result = -1;

    log("Starting polling call");
    // Invoke server method asynchronously
    // by using the async. server proxy
    asyncServerProxy.requestConfirmation();
```

```
    // Get futureResult object from the proxy
    futureResult = ((IFutureResult)
      asyncServerProxy).getFutureResult();
    // Check periodically if the server method
    // has returned
    while (!futureResult.isAvailable())
      log("I'm doing something serious!\n");
    // Now a result is available
    log("Result has arrived");
    result = futureResult.getIntResult();
    return result;
  }

  // This method performs an asynchronous notification-
  // based invocation of the method
  // 'requestConfirmation', provided by the
  // AsyncServerAgent.
  public void notificationResultHandling()
                 throws Throwable {

    log("Starting notifying call");
    // Invoke server method asynchronously
    // by using the async. server proxy
    asyncServerProxy.requestConfirmation();
    // Get futureResult object from the proxy
    futureResult = ((IFutureResult)
      asyncServerProxy).getFutureResult();
    // The client agent adds itself as result
    // listener to the futureResult object
    futureResult.addResultListener(this);
    // Note: The result will be retrieved by the
    // method resultHasArrived
    // of the client's result listener.
    log("Listening for notification");
  }

  // The live method requests the desired communication
  // mode from the user and performs a remote method
  // call on the AsyncServerAgent, using the selected
  // mode.
  public void live() {
    String comMode;
    int serverResult;

    comMode = requestCommunicationMode();
    while (comMode != null) {
      serverResult = -1;
      try {
```

```
        if (comMode.equals("sync."))
          // Synchronous method incocation
          serverResult = synchronousInvocation();
        else if (comMode.equals("async. blocking"))
          // Asynchronous, blocking method invocation
          serverResult = blockingResultHandling();
        else if (comMode.equals("async. polling"))
          // Asynchronous, polling method invocation
          serverResult = pollingResultHandling();
        else if (comMode.equals(
                  "async. notification"))
          // Notif. based method invocation
          notificationResultHandling();
        log("Server result = " + serverResult);
      }
      catch (AsyncServerException e) {
        log("User exception caught: ", e);
      }
      catch (AsyncTimeoutException e) {
        log("Timeout! Server seems to be busy.");
      }
      catch (Throwable t) {
        log("Communication exception caught: ", t);
      }
      comMode = requestCommunicationMode();
    }
  }

  // This method is automatically called when a server
  // method, previously invoked in a notification-based
  // way, has returned. In this way, the
  // client agent is automatically notified about the
  // arrival of the method result.
  public void resultHasArrived(ResultEvent e){
    FutureResult fResult;
    int serverResult = -1;
    log("Listener notified.");
    fResult = (FutureResult) e.getSource();
    try {
      serverResult = fResult.getIntResult();
    }
    catch (Throwable t) {
      log("Exception caught: ", t);
    }
    if (serverResult != -1)
      log("Notified server result = " + serverResult);
  }
```

```
}
```

### 9.6.3 Running the Scenario

Requirements:

- A running agency domain service. Note that this service has to be started before the agencies, and the service's address has to be specified when starting the agencies in order to register them. Please refer to the User's Guide for information about how to start agencies and agency domain services.
- Two running agencies
  *Since the agents in this example create an own GUI that may block the agency GUI, it is recommended that you do not activate the agency GUI. Instead, start the agencies just with their textual interface (command option -tui). Please refer to the paragraphs titled „Running the Examples" at the beginning of Chapter 2 in order to get a detailed explanation about the possibly occurring GUI problems.*
- If you are using a JDK 1.2 environment, you must have generated a proxy class (named IAsyncServerAgentP) by invoking the Grasshopper stub generator with the interface class IAsyncServerAgent as input parameter. The file IAsyncServerAgentP.class should be stored either in a directory belonging to the Java classpath or in the code base directory of the AsyncClientAgent. In a JDK 1.3 environment, this class is not needed. Even if it is available, it will not be used. Instead, the proxy is dynamically generated by the AsyncClientAgent at runtime.

Running the Example:

Figure 11: Asynchronous Communication Scenario

Create the AsyncServerAgent inside one of the running agencies via the agency's UI (1). (This agent has to be created first, since the `AsyncClientAgent` tries to contact the `AsyncServerAgent` via the communication service.)

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.asyncCom.AsyncServerAgent
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

Create the AsyncClientAgent inside the other agency via the agency's UI (2). (You should not create both agents inside the same agency, since both agents provide modal dialogs that may block each other.)

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.asyncCom.AsyncClientAgent
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

After creating the AsyncClientAgent, this agent creates a proxy of the AsyncServerAgent (3) and asks you to select a communication mode that is to be used for contacting the server (4). After pressing the OK button, the client invokes the server method (5), and the call is forwarded by the proxy to the `AsyncServerAgent`. The server in turn starts its GUI, requesting the user to specify whether the method is to return regularly or whether an exception is to be thrown. In parallel, the client agent acts in one of the following ways, depending on the previously selected communication mode:

- If the server method has been invoked *synchronously*, the client agent is blocked until the server method has returned (9a), i.e., until the user has pressed one of the buttons of the server GUI (8). After this, the client's GUI appears again, and a new communication mode can be selected (4). Note that in this case, the remaining steps (6, 9b, and 10) of the above figure are not performed.
- If the server method has been invoked *asynchronously*, the client agent creates a `FutureResult` object via the server proxy (6) and is then free to perform its own tasks in parallel to the server agent (7). After you have pressed one of the buttons of the server agent's GUI (8), the result of the server method (i.e., a return value or an exception, depending on your previously made choice) is transferred to the server proxy (9a) which in turn forwards the result to the `FutureResult` object (9b). Now the client agent is able to retrieve the result via one of the `get<...>Result()` methods, to be invoked on the `FutureResult` object (10).

  Note that the steps (7) and (10) of the above figure depend on the selected kind of asynchronous behavior:

  - If the server method has been invoked *asynchronously and blocking*, the client prints some outputs in order to show that the server method is performed independently of the client's thread (7). After this, the client waits for the server's result for 10 seconds (10). If the server method does not return within this time frame, the client stops waiting, caused by an exception thrown by the proxy.
    Please check this behavior by selecting the asynchronous blocking call at the client side. At the server side, press one of the GUI buttons within the client's time frame, and the next time wait for more than 10 seconds before pressing the button.

- If the server method has been invoked *asynchronously and polling*, the client enters a loop that ends when the server method has returned (7). After this, the client retrieves the result (10).
- If the server method has been invoked *asynchronously notification-based*, the client's GUI re-appears at once after invoking the server method. When the server method has returned, the client's listener method (i.e., the method `ResultHasArrived(...)`) is invoked. (For the sake of simplicity, the notification-based behavior is not reflected in the above figure. Please have a look at the source code instead.)

### 9.6.4 Summary

- A Grasshopper proxy object is able to handle either synchronous or asynchronous communication. The communication mode has to be selected when the proxy is created, and it remains the same for the entire life time of the proxy. If a client wants to use synchronous as well as asynchronous communication on a single server, the client has to create two proxies.
- A proxy maintains the result of an asynchronous method call inside an object of the class `FutureResult`. A client can get a reference to this object in order to perform the asynchronous result handling.
- In order to retrieve the result of an asynchronously invoked method, the client can use one of the `get<...>Result()` methods provided by the FutureResult object which is associated with the method invocation.
- In order to catch exceptions, the get<...>Result() method has to be included in a try/catch block. The catch blocks must include all exceptions that may be thrown by the server method as well as the superclass `java.lang.Throwable`. The latter one is meant to catch possible communication exceptions/errors.
- A client can handle asynchronously arriving results in different ways: The client can block its own execution until the result has arrived, it can periodically poll for the result, or it can order the `FutureResult` object to be notified when the result has arrived.

## 9.7 Static vs. Dynamic Method Invocation

Concerning the communication scenarios that have been explained in the previous sections, the client agents must have access to the proxy class code of

the corresponding server agent. A proxy provides all methods of the server agent that are to be accessible via the communication service and that are therefore included in the server interface.

**Generic proxy**

In addition to this *static communication* where the concrete server methods are available for the client agents, Grasshopper supports *dynamic communication* where the clients invoke a *generic method* on a *generic proxy class*.

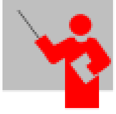
Dynamic communication is of particular importance if a client knows the method signatures of a server agent, but does not have access to the server's proxy code.

**Dynamic method calls**

In order to perform a dynamic method call on a server agent, a client invokes the static method `invoke(...)` of the class `de.ikv.grasshopper.communication.DynamicInvoker`. This method requires the following information about the server method:

- The identifier of the server agent (Java type: `de.ikv.grasshopper.type.Identifier`)
- The name of the server method that is to be invoked (Java type: `java.lang.String`)
- The classes of all method parameters in the order of their appearance in the concrete server method (Java type: `java.lang.Class[]`)
- The object values of all method parameters in the order of their appearance in the concrete server method (Java type: `java.lang.Object[]`)
- The current location of the server agent, only needed if no agency domain service is available. If no location is to be specified, this parameter has to be set to `null`. (Java type: `de.ikv.grasshopper.communication.GrasshopperAddress`)
- **User-defined classes** If the server method defines a return type that is not a standard Java type, the client has to specify the class loader that is responsible for retrieving the class of the return type. (Java type: `java.lang.ClassLoader`)
  As described in Section 9.1, user-defined classes have to be serializable if they are to be used as parameters and/or return types of server methods. In order to achieve this, the classes have to implement the interface `java.io.Serializable`.
- **Asynchronous dynamic call** If the dynamic method call is to be performed asynchronously, a `FutureResult` object has to be provided for maintaining the result of the invoked method. Please refer to Section 9.5 for detailed information about asynchronous communication. (Java type: `de.ikv.grasshopper.communication.FutureResult`)

The return type of the `invoke(...)` method is `java.lang.Object`. Thus, the client agent has to convert a retrieved return value to the concrete type of the real server method.



Consider the following method of a server proxy:

```
public int method1(int a, Integer b);
```

The following lines of code represent the corresponding dynamic method call performed at the client side:

```
Integer result = (Integer)DynamicInvoker.invoke (
  serverId, "method1",
  new Class[]
    {java.lang.Integer.TYPE, java.lang.Integer.class},
  new Object[]
    {new Integer(123), new Integer(456)},
  null);
```

**Handling primitive types**

Concerning this example, please note that the return type as well as the first parameter of the server method are primitive data types, not derived from `java.lang.Object`. Since the dynamic `invoke(...)` method is only able to return subtypes of `Object`, the client has to perform a cast to an `Object` type. In the example, the return value as well as the first method parameter are converted to `Integer`. In order to get the original simple type `int`, the client can perform the following additional call:

```
int simpleResult = result.intValue();
```

Beside the return value, also the first parameter of the server method is of the simple type `int`. Concerning the dynamic call, this simple type has to be considered when specifying the classes/types of the server method parameters. As shown in the example, the primitive data type is specified as `java.lang.Integer.TYPE`, while the Object type is specified as `java.lang.Integer.class`. The parameter values are both handled as `java.lang.Integer`.

## 9.8 Dynamic Communication Scenario

The example scenario for dynamic communication consists of four classes/interfaces, covered by the package `examples.dynamicCom`:

- `DynamicServerAgent` (see Example 14 in Section 9.8.1): An agent that provides four methods to the communication service. Each method has different parameter and return types which require a specific handling at the client side.

- `IDynamicServerAgent` (see Example 15 in Section 9.8.1): The server interface that contains the methods which have to be accessible for the client agent. This interface is the basis for the generation of server proxies.
- `TestDataPacket` (see Example 16 in Section 9.8.1): A class that is used as parameter as well as return type of one server method. Note that for handling this class, the client has to specify a class loader when dynamically invoking the server method.
- `DynamicClientAgent` (see Example 17 in Section 9.8.2): The client agent that invokes the accessible methods of the server.

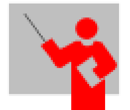
### 9.8.1 Example: DynamicServerAgent

The DynamicServerAgent implements the interface IDynamicServerAgent. This interface contains four methods that are to be accessible via the communication service. Each method has different parameter and return types which require a specific handling at the client side. Concerning the methods themselves, there is nothing more to say, since their internals are not very exciting.

Note that `method3(...)` tries to access the `String` array at an invalid index. Thus, an `ArrayOutOfBound` exception is thrown. This programming error is intended in order to show that the client retrieves this exception.

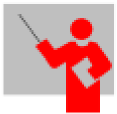
The source code of the corresponding client agent is described in Section 9.8.2.

Example 14: DynamicServerAgent



```
package examples.dynamicCom;

import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.communication.*;
import javax.swing.*;
import java.awt.*;

// This class realizes the server agent of the dynamic
// communication scenario
public class DynamicServerAgent extends MobileAgent
                  implements IDynamicServerAgent
{
  public String getName() {
    return "DynamicServerAgent";
  }

  // The following four methods are to be called by the
```

```
    // DynamicClientAgent by using
    // the dynamic communication mechanism provided by
    // Grasshopper.

    public Integer method1(int a, Integer b) {
      log("method1(" + a + ", " + b + ") called.");
      log("    returning 42.");
      return new Integer(42);
    }

    public int method2() {
      log("method2() called.");
      log("    returning -42.");
      return -42;
    }

    public void method3(String[] s) {
      log("method3(");
      for (int i = 0; i < s.length; i++)
        log("    " + s[i]);
      // The next line will cause an ArrayOutOfBound
      // exception.
      // Let's see if the client will notice this.
      log("    " + s[s.length] + ") called.");
    }

    public TestDataPacket method4(TestDataPacket s) {
      log("method4(" + s + ") called.");
      return s;
    }

    public void live() {
      log("ready.");
    }
  }
```



Example 15: IDynamicServerAgent

```
  package examples.dynamicCom;

  public interface IDynamicServerAgent
  {
    public Integer method1(int a, Integer b);
    public int method2();
    public void method3(String[] s);
    public TestDataPacket method4(TestDataPacket s);
  }
```

```
}
```

The DynamicServerAgent uses the class `TestDataPacket` as parameter as well as return type of `method4(...)`. Note that this class has to implement the interface `java.io.Serializable` in order to be usable for the communication service. The following listing defines this additional class. Please do not wonder about its semantic meaning. Its only purpose is to show how to handle user-defined classes in the context of the communication service.

Example 16: TestDataPacket

```
package examples.dynamicCom;

// This class is just meant to show how to use a 'self
// made' object type as parameter and return type of
// dynamically invoked methods.
// (See method4(...) of the DynamicServerAgent.)
public class TestDataPacket implements
                  java.io.Serializable {

  public TestDataPacket(){
  }

  public Integer i = new Integer(1);
  public char c = 'c';
  public Float f = new Float((float) 5.0);
  public Long l = new Long((long) 4.0);
  public String s = "I'm the outer class";

  public String toString(){
    StringBuffer b =
      new StringBuffer("Class TestDataPacket");
    return b.toString();
  }

  class AnInnerClass{
    String s = new String("I'm the inner class");
    public AnInnerClass(){
    }
    public String toString(){
      return new String("Class AnInnerClass\n" + s );
    }
  }
}
```

A description about how to run the example is given in Section 9.8.3.

### 9.8.2 Example: DynamicClientAgent

At the beginning of its `live(...)` method, the client agent requests the `IRegion` interface of the local agency in order to look for the DynamicServerAgent. The interface is retrieved via the method `getRegion()` which is provided by the agent's superclass `Agent`. In contrast to the agency interface `IAgentSystem` that enables a client to look for agents inside the local agency, the interface `IRegion` provides access to the agency domain service and thus enables a client to look for agents (and agencies) in a whole region/domain. The client uses the `IRegion` proxy in order to look for an agent with the name 'DynamicServerAgent'. If more than one agent is found, the client agent just takes the first one from the retrieved list. The reason for this lookup is that the client agent needs the identifier of the server agent in order to perform dynamic method calls.

After retrieving the server identifier, the DynamicClientAgent invokes the methods of the server agent. The following paragraphs explain each method call in detail.

**method1(...)**

Dynamic call of `method1(...)`:

The server method `method1(...)` is dynamically called with the following parameter values:

- `serverId`:
  The identifier of the server agent that is to be contacted.
- „`method1`":
  The name of the server method that is to be invoked.
- `new Class[]`
  `{java.lang.Integer.TYPE,`
  `java.lang.Integer.class}`:
  This server method requires parameters of the simple type `int` and of the Object type `Integer`. The simple type has to be considered when specifying the classes/types of the server method parameters. As shown in the source code, the primitive data type is specified as `java.lang.Integer.TYPE`, while the Object type is specified as `java.lang.Integer.class`.
- `new Object[]`
  `{new Integer(para01), para02)}`:
  The parameter values that are to be transferred to the server are both handled as `java.lang.Integer`, since the `invoke(...)` methods expects subclasses of `java.lang.Object`.
- `null`:

Via the last parameter of the `invoke(...)` method, the server's location can be specified. This parameter is set to `null`, since the example assumes a running agency domain service, so that the DynamicInvoker is able to locate the server agent by itself.

- `null`:
  Since a standard Java class is used as return type, no class loader has to be specified.

The return type of `method1(...)` is `java.lang.Integer`. Since this is a subclass of `java.lang.Object`, the client agent can directly convert the return value of the `invoke(...)` method from `Object` to `Integer` in order to retrieve the result.

Dynamic call of `method2()`:

**method2(...)**

The server method `method2(...)` is dynamically called with the following parameter values:

- `serverId`:
  The identifier of the server agent that is to be contacted.
- „`method2`":
  The name of the server method that is to be invoked.
- `new Class[0], new Object[0]`:
  The server method itself does not require any parameters, so that the third and fourth parameter of the `invoke(...)` method are initialized with an empty `Class` respectively `Object` array.
- `null`:
  Via the fifth parameter of the `invoke(...)` method, the server's location can be specified. This parameter is set to `null`, since the example assumes a running agency domain service, so that the DynamicInvoker is able to locate the server agent by itself.
- `null`:
  Since a standard Java class is used as return type, no class loader has to be specified.
- `futureResult2`:
  The dynamic invocation of `method2()` is performed asynchronously, so that a `FutureResult` object is provided as last parameter. The handling of the asynchronously arriving result is exactly the same as described in Section 9.5.

Note that the `invoke(...)` method is called without directly specifying a variable for retrieving the result. The reason is, as explained in Section 9.5, that the result of an asynchronously called method cannot be retrieved

directly by the method call itself. Instead, the `getResult()` method or one of the `get<Type>Result()` methods provided by the `FutureResult` object have to be invoked. Concerning the example, the client agent invokes the `getIntResult()` method, since the expected return value is of the simple Java type `int`.

**method3(...)** Dynamic call of `method3(...)`:

The server method `method3(...)` is dynamically called with the following parameter values:

- `serverId`:
  The identifier of the server agent that is to be contacted.
- `„method3"`:
  The name of the server method that is to be invoked.
- `classArray`:
  The required parameter of the server method is an array of `String`. Please have a look at the source code in order to see how this parameter has been constructed.
- `argumentArray`:
  This parameter represents the actual parameter of the server method, i.e., the `String` array. Please have a look at the source code in order to see how this parameter has been constructed.
- `null`:
  Via the fifth parameter of the `invoke(...)` method, the server's location can be specified. This parameter is set to `null`, since the example assumes a running agency domain service, so that the DynamicInvoker is able to locate the server agent by itself.
- `null`:
  Since a standard Java class is used as return type, no class loader has to be specified.

**method3(...) oneway** Dynamic oneway call of method3(...):

Dynamic method invocations can be performed *oneway*. That means, the client ignores possibly arriving return values and exceptions. Thus, the parameter `FutureResult` is not required. Besides, there is generally no need for specifying a class loader, even if the return value is realized in terms of a user-defined class, as in `method4(...)` of the current example.

For enabling oneway calls, the `DynamicInvoker` provides the method `invokeOneway(...)`. The parameters are the same as known from the usual `invoke(...)` method.

The server method `method3(...)` is dynamically called with the following parameter values:

- `serverId`:
  The identifier of the server agent that is to be contacted.
- „`method3`":
  The name of the server method that is to be invoked.
- `classArray`:
  The required parameter of the server method is an array of `String`. Please have a look at the source code in order to see how this parameter has been constructed.
- `argumentArray`:
  This parameter represents the actual parameter of the server method, i.e., the `String` array. Please have a look at the source code in order to see how this parameter has been constructed.
- `null`:
  Via the fifth parameter of the `invoke(...)` method, the server's location can be specified. This parameter is set to `null`, since the example assumes a running agency domain service, so that the DynamicInvoker is able to locate the server agent by itself.

Dynamic call of method4(...):

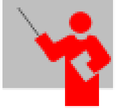
**method4(...)**

The server method `method4(...)` is dynamically called with the following parameter values:

- `serverId`:
  The identifier of the server agent that is to be contacted.
- „`method4`":
  The name of the server method that is to be invoked.
- `new Class[]{TestDataPacket.class}`:
  This method requires a user-defined Java type as parameter. However, the provision of the corresponding class is handled exactly as a standard Java class.
- `new Object[]{tdp}`:
  Also the provision of the parameter value corresponds to the handling of standard Java classes.
- `null`:
  Via the fifth parameter of the `invoke(...)` method, the server's location can be specified. This parameter is set to `null`, since the example assumes a running agency domain service, so that the Dynam-

icInvoker is able to locate the server agent by itself.

- `this.getClass().getClassLoader()`:
  Since a self-defined Java class (i.e., non-standard Java type) is used as return type of a dynamically invoked method, a class loader has to be specified which is responsible for retrieving the class.



Example 17: DynamicClientAgent

```
package examples.dynamicCom;

import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.communication.*;
import de.ikv.grasshopper.util.*;
import javax.swing.*;
import java.awt.*;

// This class realizes the client agent of the dynamic
// communication scenario
public class DynamicClientAgent
                  extends MobileAgent
{
  Identifier serverId;

  public String getName() {
    return "DynamicClientAgent";
  }

  public void action() {
    live();
  }

  // Inside the live() method, the agent sequentially
  // calls the four methods provided
  // by the DynamicServerAgent in a dynamic way.
  public void live() {
    IRegion regionProxy;
    AgentInfo serverInfos[];

    // Get proxy of local agency
    regionProxy = getRegion();
    // Look for the server agent in the
    // agency domain service
    SearchFilter filter = new
      SearchFilter(
        SearchFilter.NAME+"=DynamicServerAgent");
```

```
serverInfos = regionProxy.listAgents(
  null, filter);
serverId = serverInfos[0].getIdentifier();
log("Server located: " + serverId);

// method1:
int para01 = 123;
Integer para02 = new Integer(456);
log("Invoking method1(123, 456) synchronously.");
try {
  // Invoke method1 synchronously
  Integer result1 =
    (Integer)DynamicInvoker.invoke(
      serverId, "method1",
      new Class[]{java.lang.Integer.TYPE,
      java.lang.Integer.class},
      new Object[]{new Integer(para01), para02},
      null, null);
  log("Result of method1 = " + result1);
}
catch (Throwable e) {
  log("Exception caught: ", e);
}

// method2:
FutureResult futureResult2 = new FutureResult();
log("Invoking method2 asynchronously polling.");
try {
  // Invoke method2 asynchronously polling
  DynamicInvoker.invoke(
    serverId, "method2",
    new Class[0], new Object[0],
    null, null, futureResult2);
}
catch (java.lang.reflect.
  InvocationTargetException e) {
  log("Exception caught: ", e);
}
log("I'm doing something serious!");
while (!futureResult2.isAvailable())
  System.out.print(".");
System.out.println();
try {
  // Get asynchronous result.
  int result2 = futureResult2.getIntResult();
  log("Result of method2 = " + result2);
}
catch (Throwable t) {
```

```
  log("Exception caught: ", t);
}

// method3:
log("Invoking method3(\"I\", \"am\", \"the\",
  \"client\").");
String para03[] = {"I", "am", "the", "", "client"};

Class classArray[] = new Class[1];
Object argumentArray[] = new Object[1];
argumentArray[0] = para03;
classArray[0] = para03.getClass();

try {
  DynamicInvoker.invoke(
    serverId, "method3", classArray,
    argumentArray, null, null);
}
catch (java.lang.reflect.
  InvocationTargetException e) {
  log("Exception caught: ",
    e.getTargetException());
}
catch(Throwable e){
  log("Exception caught: ", e);
}

// method3 oneway:
log("Invoking method3(\"I\", \"am\", \"the\",
  \"Oneway\" \"client\").");
para03[3] = "oneway";
try {
  DynamicInvoker.invokeOneWay(
    serverId, "method3", classArray,
    argumentArray, null);
  log("Exceptions do not harm me...");
}
catch(Throwable e){
  log("Exception caught: ", e);
}

// method4:
log("Invoking method4()");
TestDataPacket tdp = new TestDataPacket();
try {
  tdp = (TestDataPacket)
    DynamicInvoker.invoke(
      serverId, "method4",
```

```
            new Class[]{TestDataPacket.class},
            new Object[]{tdp}, null,
            this.getClass().getClassLoader());
        log("What you send is what you get..." + tdp);
      }
      catch(Throwable e){
        log("Exception caught: ", e);
      }
    }
  }
```

### 9.8.3 Running the Scenario

Requirements:

- A running agency domain service. Note that this service has to be started before the agencies, and the service's address has to be specified when starting the agencies in order to register them. Please refer to the User's Guide for more information about how to start agencies and agency domain services.
- At least one running agency

Running the Example:

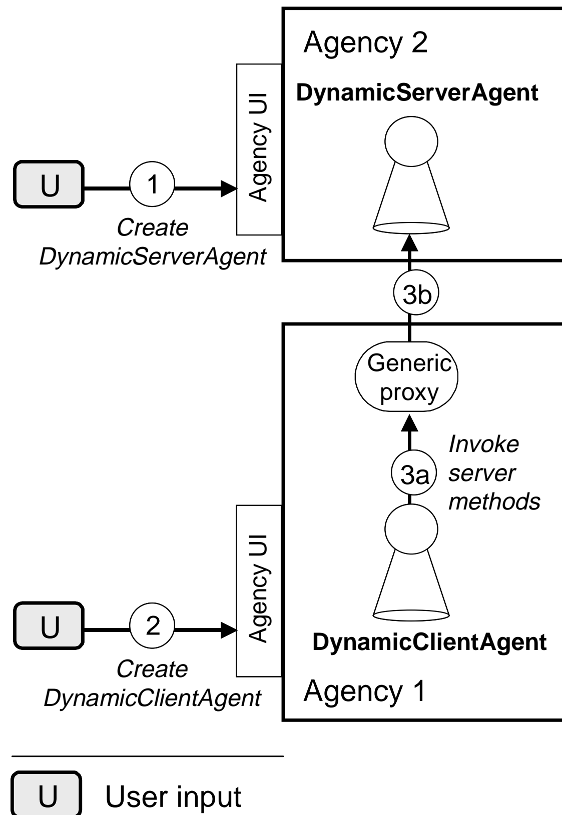



Figure 12: Dynamic Communication Scenario

Create the `DynamicServerAgent` inside a running agency via the agency's UI (1). (This agent has to be created first, since the `DynamicClientAgent` tries to contact the `DynamicServerAgent` via the communication service.)

> If you are using the textual user interface of the agency, please create the agent by means of the following command:
>
> ```
> cr a examples.dynamicCom.DynamicServerAgent
> ```
>
> If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

Create the `DynamicClientAgent` either in the same or a different agency (2).

> If you are using the textual user interface of the agency, please create the agent by means of the following command:
>
> ```
> cr a examples.dynamicCom.DynamicClientAgent
> ```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

After creating both agents in the order mentioned above, the client agent invokes the methods on the server agent by using a generic proxy (3). Both agents run without the need for any user interaction. Thus, there is no need for any further explanations. Just have a look at the agents' output and compare it with the agents' source code.

Note that the server throws an `ArrayOutOfBound` exception inside `method3(...)`. This is intended in order to show that the client receives this exception.

### 9.8.4 Summary

- In contrast to *static communication* where the concrete server methods are available for the client agents via a specific server proxy, Grasshopper supports *dynamic communication* where the clients invoke a *generic method* on a *generic proxy class*.
- Dynamic communication is of particular importance if a client knows the method signatures of a server, but does not have access to the server's proxy code.
- In order to perform a dynamic method call on a server, a client invokes the static method `invoke(...)` of the generic proxy class `de.ikv.grasshopper.communication.DynamicInvoker`.
  The following information of the server has to be provided: the server identifier, the name of the server method, and the classes and values of the method parameters. In specific cases, additional information is needed: the server location, a class loader, and a reference of a `FutureResult` object.
- The result of a dynamically invoked method is provided as an instance of the class `java.lang.Object`. The client has to convert this class to the actual result type of the invoked method.

## 9.9 Unicast vs. Multicast Communication

In specific scenarios, a client may want to invoke the same method successively on a set of server agents. Considering the communication mechanisms that have been explained in the previous sections, the client agent would require a

proxy of each server agent that is to be contacted, and the method calls would have to be performed sequentially on each proxy.

**Group proxy**

In order to simplify such a scenario for the client agent, Grasshopper supports *multicast communication*. Instead of creating a set of server proxies and sequentially invoking the same method on each single proxy, a client agent may create a *group proxy* and register a set of server agents at this proxy. When the client invokes a method on the group proxy, the proxy forwards the call to all registered agents. Internally, the group proxy contacts all registered server agents sequentially. Thus, only little advantage concerning the performance of the method invocations is achieved by using multicast communication. The main purpose of this mechanism is to facilitate the implementation at the client side.

**Group interfaces**

A Grasshopper group proxy is a Java object that implements two interfaces: the interface `de.ikv.grasshopper.communication.IGroup` and the server interface of those server agents that are to be added to the group, such as `IMulticastServerAgent`, concerning the example introduced in Section 9.10.

**Interface IGroup**

The interface `IGroup` is implemented by every group object and provides the following methods for group establishment and maintenance:

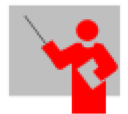
- `getMembers()`: This method returns the identifiers of all group members in form of String objects.
- `getResult()`: This method returns a `MulticastResult` object. The purpose of this object is similar to the purpose of a `FutureResult` object in asynchronous unicast scenarios (see Section 9.5): The `MulticastResult` object retrieves asynchronously arriving results of group members. Detailed information is provided below.
- `invoke(...)`: This method enables a client agent to perform a multicast call dynamically. Similar to dynamic unicast calls (see Section 9.8), the name of the server method as well as the classes and values of all method parameters have to be specified. However, please note that it is also possible to call the concrete server method directly, since the group proxy implements, besides the `IGroup` interface, also the server interface of the group members.
- `add(...)`: Via this method, a server agent can be added to a group proxy. For this purpose, the server agent's identifier has to be provided. Optionally, the server agent's location can be specified which is only required no agency domain service is available.
- `remove(...)`: This method enables the removal of a server agent from the group proxy. For this purpose, the server agent's identifier has to be

provided.

- `setType(...)`: This method sets the termination mode of the subsequent multicast calls to be performed on the group proxy. Grasshopper supports three types: AND termination, OR termination, and INCREMENTAL termination. Detailed information is provided below.

**Creating a group proxy**

A group proxy is created by calling the method `createGroup(...)` on the class `de.ikv.grasshopper.communication.ProxyGenerator`. Note that the group proxy should be of the generic Java class `Object`. The reason is that, as mentioned above, a group proxy implements two interfaces. By creating a group proxy as instance of the class `Object`, the proxy can be casted to both interfaces, depending on the method that is to be performed.



When creating a group proxy, the class of the server interface of the intended group members has to be provided as parameter.

```
Object serverGroup = ProxyGenerator.createGroup(
                         IMulticastServerAgent.class);
```

Via the interface `IGroup`, new members can be added to the group proxy:

```
((IGroup)serverGroup).join(
              serverAgentIdentifier);
```

Via the server interface, such as `IMulticastServerAgent` concerning the example introduced in Section 9.10, the methods of the group members can directly be invoked on the group proxy:

```
((IMulticastServerAgent)serverGroup).
     requestConfirmation("Client message");
```

### The termination mode

**Termination mode...**

Since a multicast call is usually sent to more than one server agent, the result of such a call is represented by a set of return values and/or exceptions. Due to the fact that a group proxy sequentially contacts all server agents, the multicast results do usually not arrive at the client side exactly at the same time. In order to fulfill the individual needs of the client concerning the retrieval of multicast results, the group proxy provides the following three mechanisms:

**...AND**

1. AND Termination

   The server method returns when *all* server results have arrived at the client side. Up to this point in time, the client agent is blocked.

**...OR**

2. OR Termination

   The server method returns when *the first* server result has arrived at the client side. Up to this point in time, the client agent is blocked.

**...INCREMENTAL**

3. INCREMENTAL Termination

   The server method returns at once. The client agent can request the results when they are needed.

The kind of termination can be set via the method `setType(...)` of the interface `IGroup`.

**Result handling**

The group proxy generally performs a multicast call *asynchronously*. Thus, the result handling is similar to asynchronous unicast invocations, as described in Section 9.5.1. After invoking a server method on the group proxy, the client agent has to call the method `getResult()` on the group proxy's interface `IGroup`. This method returns an instance of the class `de.ikv.grasshopper.communication.MulticastResult` which offers the following methods:

- `getFirst()`: This method returns the `FutureResult` object of the server agent whose result has arrived first.
- `getFutureResult(...)`: This method returns the `FutureResult` object of a specific server agent. The demanded server agent is selected by means of its identifier.
- `getNumberOfReturned(...)`: This method returns the number of server agents that have already returned a result.
- `getResult(...)`: This method returns the result of a specific server agent. The demanded server agent is selected by means of its identifier. Since the return type of this method is `java.lang.Object`, the client agent has to cast this type to the actual return type of the server method.
- `isAvailable(...)`: This method checks whether at least one of the contacted server agents has already returned a result.

Note that the handling of the `FutureResult` object is exactly the same as described in Section 9.5.1. By enabling the client agent to get a separate `FutureResult` object for each server agent of the contacted multicast group, a high degree of flexibility is provided, for instance by enabling the client to set different timeouts or to apply different result handling mechanisms (blocking, polling, notification) to different server agents.

Note that it is not necessary for a client agent to retrieve a `FutureResult` object of a specific server agent before retrieving the actual server result. Via the method `getResult()` of the class `de.ikv.grasshopper.communication.MulticastResult`, a client agent can retrieve the server result directly. However, in this case the client agent should verify that the result is already available. Please have a look at the following example for clarification.

# 9.10 Multicast Communication Scenario

The example scenario for multicast communication consists of three classes/ interfaces, covered by the package `examples.multicastCom`:

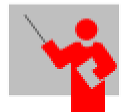
- `MulticastServerAgent` (see Example 18 in Section 9.10.1): An agent that provides one method to the communication service.
- `IMulticastServerAgent` (see Example 19 in Section 9.10.1): The server interface that contains the method which has to be accessible for the client agent. This interface is the basis for the generation of server proxies.
- `MulticastClientAgent` (see in Example 20 Section 9.10.2): The client agent that invokes the accessible method of the server.

## 9.10.1 Example: MulticastServerAgent

The MulticastServerAgent implements the interface IMulticastServerAgent. This interface contains one method that is to be accessible via the communication service. This method creates a modal dialog, requesting the user to press a button that terminates the method. Finally, the method returns the identifier of the server agent.

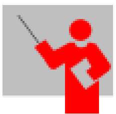
The source code of the corresponding client agent is described in Section 9.10.2.

Example 18: MulticastServerAgent



```
package examples.multicastCom;

import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.communication.*;
import javax.swing.*;
import java.awt.*;

// This class realizes the server agent of the multicast
// communication scenario.
public class MulticastServerAgent extends MobileAgent
                    implements IMulticastServerAgent
{
  int result;

  // No creation arguments needed.
  public void init(Object[] creationArgs) {
    result = 1;
  }
```

```
    public String getName() {
      return "MulticastServerAgent";
    }

    // This method requests user input via a graphical
    // component.
    // The user just has to confirm the dialog by clicking
    // the OK button.
    public String requestConfirmation(
                            String clientMessage) {

      log("Client request arrived. Returning my ID: " +
        getInfo().getIdentifier().toString());
      JOptionPane.showMessageDialog(
        null, clientMessage,
        "MulticastServerAgent",
        JOptionPane.PLAIN_MESSAGE);
      return getInfo().getIdentifier().toString();
    }

    public void live() {
      log("ready.");
    }
  }
```



Example 19: IMulticastServerAgent

```
  package examples.multicastCom;

  public interface IMulticastServerAgent
  {
    public String requestConfirmation(
      String clientMessage);
  }
```

A description about how to run the example is given in Section 9.10.3.

### 9.10.2 Example: MulticastClientAgent

**init(...)**

Inside its `init(...)` method, the MulticastClientAgent contacts the agency domain service via the `IRegion` interface of the local agency and requests a list of all agents with the name 'MulticastServerAgent'. This is required since the client agent needs the identifiers of the demanded server agents in order to add them to a group proxy. After retrieving a list of available server agents, the

client creates a group proxy, i.e. a proxy object for multicast communication, and adds all retrieved server agents to this proxy. This is done by providing the server agents' identifiers to the group proxy via the `join(...)` method.

**requestTermination-Mode()**

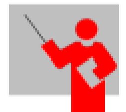
The method `requestTerminationMode()` is called from inside the agent's `live()` method and activates the client's GUI that enables the user to select between the following termination modes:

- *AND Termination*
  A multicast call returns after *all* server results have arrived. Up to this point in time, the client agent is blocked.
- *OR Termination*
  A multicast call returns after the first server results has arrived. Up to this point in time, the client agent is blocked.
- *INCREMENTAL Termination*
  A multicast call returns at once. The client can request the results when they are needed.

**live()**

Inside its `live()` method, the client requests the selection of a termination mode via its GUI and performs a multicast call by applying this termination mode to the method invocation. After this, the client prints out all retrieved results.

Example 20: MulticastClientAgent



```
package examples.multicastCom;

import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.communication.*;
import de.ikv.grasshopper.util.*;
import javax.swing.*;
import java.awt.*;

// This class realizes the client agent of the multicast
// communication scenario.
public class MulticastClientAgent extends MobileAgent
{
  // Proxy of local agency = transient
  // (i.e., not part of the data state),
  // since it is not serializable.
  // A non-transient agency proxy would not
  // allow the agent to migrate.
  transient IRegion regionProxy;
```

```
    // Data state of the agent, since not transient
    Object serverGroup;
    int numberOfServers;
    AgentInfo[] serverInfos;

    // No creation arguments needed.
    public void init(Object[] creationArgs) {

      // Get proxy of local agency
      regionProxy = getRegion();
      // Look for the server agent in the
      // agency domain service
      SearchFilter filter =
        new SearchFilter(
          SearchFilter.NAME+"=MulticastServerAgent");
      serverInfos = regionProxy.listAgents(
        null, filter);
      // Create multicast group for server agents that
      // implement the interface IMulticastServerAgent
      serverGroup = ProxyGenerator.createGroup(
        IMulticastServerAgent.class);
      numberOfServers = serverInfos.length;
      for (int i = 0; i < numberOfServers; i++)
        // Add all found MulticastServerAgents to the
        // group
        ((IGroup)serverGroup).join(
          serverInfos[i].getIdentifier());
    }

  public String getName() {
  return "AsyncClientAgent";
  }

    // This method requests user input via graphical
    // component.
    // The user has to select the termination mode of the
    // following multicast invocation.
    public String requestTerminationMode() {
      String termMode = null;
      String options[] = {
        "AND Termination",
        "OR Termination",
        "Incremental Termination"};
      termMode = (String) JOptionPane.showInputDialog(
        null, "Termination mode:",
        "MulticastClientAgent",
        JOptionPane.QUESTION_MESSAGE, null, options,
        options[0]);
```

```
  return termMode;
}

public void live() {
  String termMode;
  String serverId;
  String serverResult;
  MulticastResult mcResult = null;

  termMode = requestTerminationMode();
  while (termMode != null) {
    serverResult = null;
    if (termMode.equals("AND Termination")) {
      // Method incocation with AND termination
      log("I'm waiting for ALL results...");
      // Perform multicast call
      ((IGroup)serverGroup).setType(
        MulticastResult.AND);
      ((IMulticastServerAgent)serverGroup).
        requestConfirmation("Client message: AND");
      mcResult = ((IGroup)serverGroup).getResult();
      // All(!) results have arrived, or timeouts
      // have exceeded.
    }
    else if (termMode.equals("OR Termination")) {
      // Method incocation with OR termination
      log("I'm waiting for ONE results...");
      // Perform multicast call
      ((IGroup)serverGroup).setType(
        MulticastResult.OR);
      ((IMulticastServerAgent)serverGroup).
        requestConfirmation("Client message: OR");
      mcResult = ((IGroup)serverGroup).getResult();
      // At least one results has arrived, or
      // timeouts have exceeded.
    }
    else if (termMode.equals(
              "Incremental Termination")) {
      // Method incocation with INCREMENTAL
      // termination
      log("I'm not waiting at all!");
      // Perform multicast call
      ((IGroup)serverGroup).setType(
        MulticastResult.INCREMENTAL);
      ((IMulticastServerAgent)serverGroup).
        requestConfirmation(
          "Client message: INCREMENTAL");
      mcResult =
```

```
              ((IGroup)serverGroup).getResult();
            // Method has returned without waiting for
            // any result.
            while (mcResult.getNumberOfReturned() == 0)
              log("I'm doing something serious!\n");
          }
          // Evaluating the results
          log(mcResult.getNumberOfReturned() +
            " result(s) available:");
          for (int i = 0; i < numberOfServers; i++) {
            serverId =
              serverInfos[i].getIdentifier().toString();
            try {
              if (mcResult.isAvailable(serverId)) {
                serverResult =
                  (String)mcResult.getResult(serverId);
                log("Result from server '" + serverId +
                  "' = " + serverResult);
              }
            }
            catch (Throwable t) {
              log("Exception caught from server '" +
                serverId + "': ", t);
            }
          }
          termMode = requestTerminationMode();
        }
      }
    }
```

## 9.10.3 Running the Scenario

Requirements:

- A running agency domain service. Note that this service has to be started before the agencies, and the service's address has to be specified when starting the agencies in order to register them. Please refer to the User's Guide for more information about how to start agencies and agency domain services.
- Several running agencies
  *Since the agents in this scenario create own GUIs that may block the agency GUI, it is recommended that you do not activate the agency GUI. Instead, start the agencies just with their textual interface (command option -tui). Please refer to the paragraphs titled „Running the Examples"*

*at the beginning of Chapter 2 in order to get a detailed explanation about the possibly occurring GUI problems.*

- If you are using a JDK 1.2 environment, you must have generated a proxy class (named `IMulticastServerAgentP`) by invoking the Grasshopper stub generator with the interface class `IMulticastServerAgent` as input parameter. The file `IMulticastServerAgentP.class` should be stored either in a directory belonging to the Java classpath or in the code base directory of the MulticastClientAgent. In a JDK 1.3 environment, this class is not needed. Even if it is available, it will not be used. Instead, the proxy is dynamically generated by the MulticastClientAgent at runtime.

Running the Example:

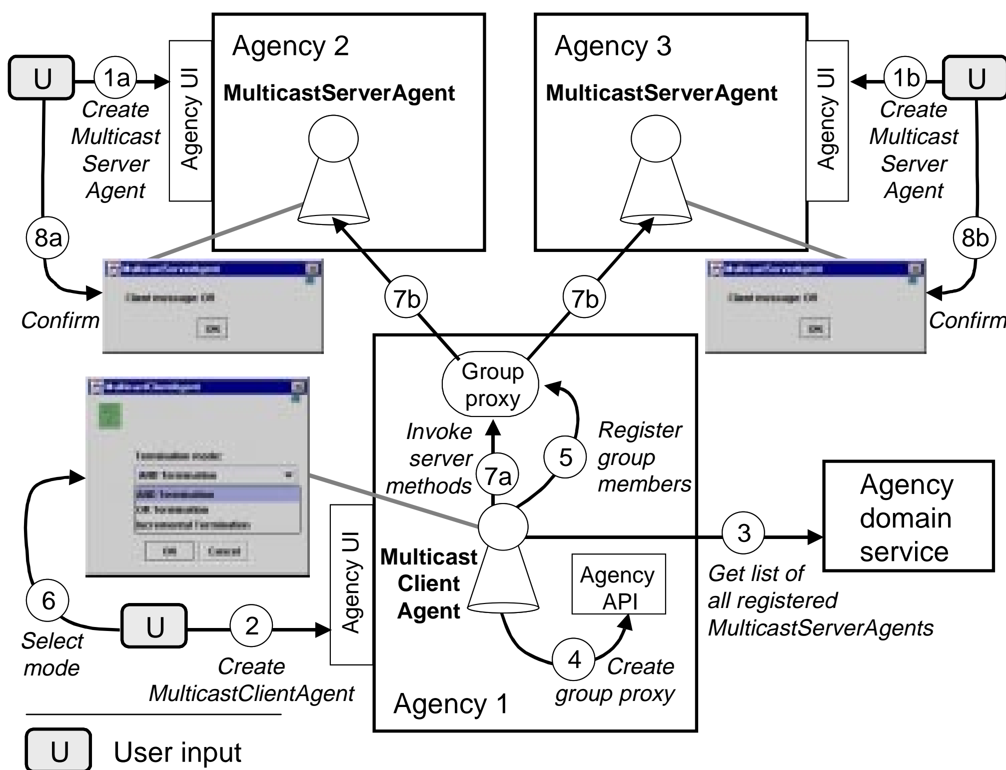



Figure 13: Multicast Communication Scenario

Create a set of MulticastServerAgents in the running agencies via the agencies' UI (1). (All server agents have to be created before the client agent.)

If you are using the textual user interface of the agency, please create the agents by means of the following command:

```
cr a examples.multicastCom.MulticastServerAgent
```

If the agents' classes are not included in the Java CLASSPATH environ-

ment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

Create one MulticastClientAgent in one of the running agencies (2). (If you start the client agent in an agency in which a server agent is already running, the agents' GUIs may block each other. Thus, it is recommended to start the client agent in a separate agency.

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.multicastCom.MulticastClientAgent
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

The client agent contacts the agency domain service in order to find all registered MigratingServerAgents (3), creates a group proxy (4), and registers all found server agents as members of the group (5). Select the termination mode in the client's GUI, press OK (6), and see what happens. Triggered by your interaction, the client agent invokes a method on the group proxy (7a) which forwards the invocation to all members of the group (7b, 7c). Depending on the termination mode, a different number of results will be retrieved by the client:

- AND termination causes the client agent to block until all servers have returned a result, i.e., until you have pressed the OK button on the GUIs of *all* server agents ((8a) *and* (8b)).
- OR termination causes the client agent to block until the first result has arrived, i.e., until you have pressed the OK button on the GUI of *one* server agent ((8a) *or* (8b)).
- INCREMENTAL termination allows the client to do something serious while the server agents are performing their asynchronously invoked method. Concerning the example, the client agent uses its waiting time for making some outputs on the agency's text console until the first result has arrived, i.e., until you have pressed the OK button on the GUI of *one* server agent ((8a) *or* (8b)).

After handling the asynchronous result(s), the client agent continues by creating its own GUI again, and the scenario proceeds with step (6).

### 9.10.4 Summary

- Instead of creating a set of server proxies and sequentially invoking the same method on each single proxy, a client agent may create a *group proxy*

and register a set of server agents at this proxy. When the client invokes a method on the group proxy, the proxy forwards the call to all registered agents.

- A Grasshopper group proxy is a Java object that implements two interfaces: the interface `de.ikv.grasshopper.communication.IGroup` and the server interface of those server agents that are to be added to the group.
- The group proxy generally performs a multicast call *asynchronously*. The method `getResult()` on the group proxy's interface `IGroup` returns an instance of the class `de.ikv.grasshopper.communication.MulticastResult`. This class offers several methods for handling the asynchronously arriving results.
- For retrieving the multicast results, a client can select one of the following modes: AND termination (blocks the client until all server results have arrived or a timeout period has expired), OR termination (blocks the client until the first server result has arrived or a timeout period has expired), or INCREMENTAL termination (does not block the client at all; the client can request the server result(s) on demand).

## 9.11 Accessing Agencies

A Grasshopper agency offers two programming interfaces to all locally residing agents as well as to remote entities:

- `IAgentSystem`: This interface contains methods that are directly associated with the offering agency itself. Among others, this interface enables the monitoring and control of local places as well as locally running agents.
- `IRegion`: This interface provides access to an agency domain service, i.e., a region registry or an LDAP server. Please refer to Section 9.12 for detailed information about this interface.

### 9.11.1 Agency Related Information

Every Grasshopper agency carries information about itself that may be accessed by other entities. This information is maintained by an instance of the class `de.ikv.grasshopper.type.AgentSystemInfo`. (similar to `de.ikv.grasshopper.type.AgentInfo` which covers agent-related

information, described in Chapter 5).

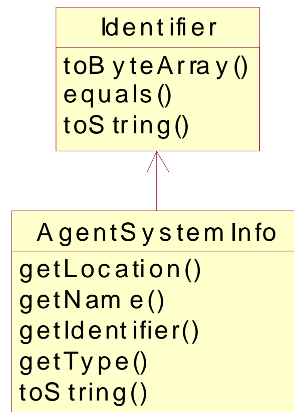



Figure 14: AgencyInfo Class Diagram

When a new agency is created, the agency generates a new `AgentSystemInfo` instance and uses creation arguments as well as environmental properties for its initialization. The `AgentSystemInfo` object is accessible for other entities via the `getInfo()` method of the agency's interface `IAgentSystem`.

**AgentSystemInfo**

The `AgentSystemInfo` class covers the following components:

- *Identifier*: The purpose of this component is to uniquely identify an agency in the distributed environment. The identifier is automatically generated by the agency during its creation. Detailed information about the structure of a Grasshopper identifier is provided in Section 5.1. (Java type: `de.ikv.grasshopper.type.Identifier`)
- *Location*: This component maintains the location of the agency in terms of a Grasshopper address. Detailed information about the structure of Grasshopper addresses is provided in Section 5.4. A concrete communication receiver can be determined by invoking the method `lookupCommunicationServer(...)` on the agency domain service at which the demanded agency is registered. Please refer to Section 9.12.1 for more information about this method. (Java type: `de.ikv.grassgopper.communication.GrasshopperAddress`)
- *Name*: This component maintains the name of the agency that has been set by the user who created the agency. (Java type: `java.lang.String`)
- *Type*: This component maintains the type of the agency. A Grasshopper agency is always of the type `GrasshopperAgentSystemType`. Other types may occur in the context of the MASIF standard. All of them are

defined as String constants in the class `de.ikv.grasshopper.util.GrasshopperConstants`. (Java type: `java.lang.String`)

Example 21 in Section 9.11.6 describes an agent that prints the contents of an agency's `AgentSystemInfo` object.

## 9.11.2 Interface IAgentSystem

The interface `IAgentSystem` represents the main access point to an agency for software entities. Its methods are meant to monitor and control locally running agents, local places, or the agency itself. The interface is accessible by locally running agents (see Section 9.11.3) as well as by remote entities (see Section 9.11.4).

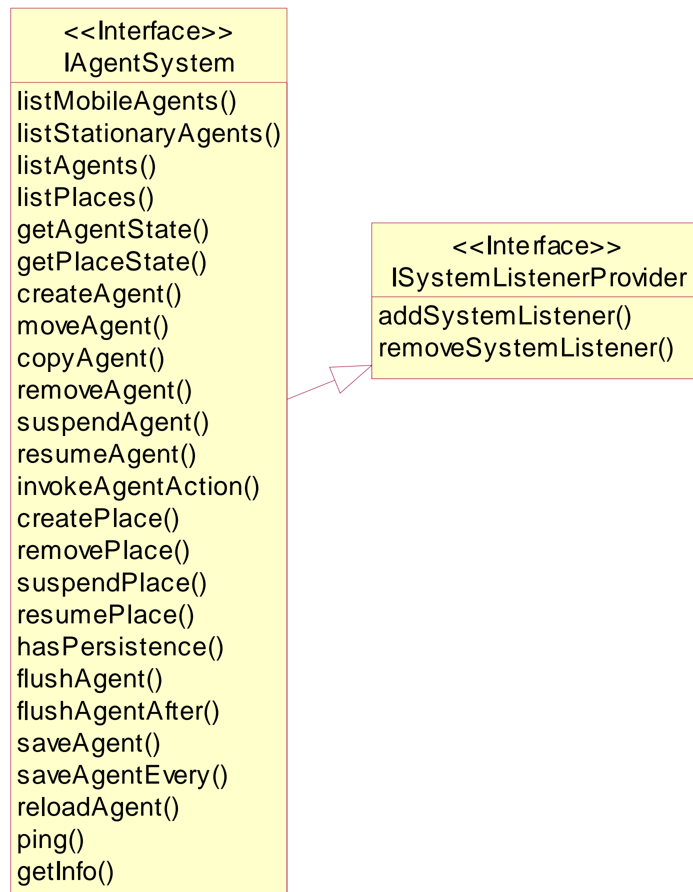



Figure 15: IAgentSystem Class Diagram

**Agent-related methods**

The interface `IAgentSystem` provides the following agent-related methods:

- `copyAgent(...)`: This method creates a copy of a specific, locally residing agent. The copy may be created inside the local agency or at another location. (Note that the agent may prohibit its copying by throwing a `VetoException` from inside the `beforeCopy()` method, as described in Chapter 8.)
- `createAgent(...)`: This method creates a new agent in the local agency.
- `flushAgent(...)`: This method flushes a locally residing agent. (This functionality is associated with the Grasshopper persistence service. Please refer to Chapter 10 for detailed information.)
- `flushAgentAfter(...)`: This method flushes a locally residing agent after a certain period of time. (This functionality is associated with the Grasshopper persistence service. Please refer to Chapter 10 for detailed information.)
- `getAgentState(...)`: This method returns the current state of a specific agent.
- `invokeAgentAction(...)`: This method invokes the `action()` method of a specific agent. Please refer to Chapter 7 for information about an agent's `action()` method.
- `listAgents(...)`: This method returns a list of locally residing agents. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- `listMobileAgents(...)`: This method returns a list of locally residing mobile agents. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- `listStationaryAgents(...)`: This method returns a list of locally residing stationary agents. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- `moveAgent(...)`: This method moves a locally residing agent to another location. (Note that the agent may prohibit its migration via the `beforeMove()` method, as described in Section 6.2.)
- `reloadAgent(...)`: This method reloads an agent. (This functionality is associated with the Grasshopper persistence service. Please refer to

Chapter 10 for detailed information.)

- `removeAgent(...)`: This method removes an agent from the local agency. (Note that the agent may prohibit its removal via its `beforeRemove()` method.)
- `resumeAgent(...)`: This method resumes a suspended agent. (Information about the different states of an agent is provided in Section 5.5.)
- `saveAgent(...)`: This method saves an agent. (This functionality is associated with the Grasshopper persistence service. Please refer to Chapter 10 for detailed information.)
- `saveAgentEvery(...)`: This method saves an agent periodically. (This functionality is associated with the Grasshopper persistence service. Please refer to Chapter 10 for detailed information.)
- `suspendAgent(...)`: This method suspends an active agent. (Information about the different states of an agent is provided in Section 5.5.)

**Place-related methods**

The interface `IAgentSystem` provides the following place-related methods:

- `createPlace(...)`: This method creates a new place in the agency.
- `getPlaceState(...)`: This method returns the current state of a specific place.
- `listPlaces(...)`: This method returns a list of local places. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- `removePlace(...)`: This method removes a place from the agency.
- `resumePlace(...)`: This method resumes a suspended place. (Similar to agents, places can be suspended and resumed. To suspend a place means to suspend all agents that are currently running inside the place. When a suspended place is resumed, all agents inside this place are also resumed.)
- `suspendPlace(...)`: This method suspends an active place. (Similar to agents, places can be suspended and resumed. To suspend a place means to suspend all agents that are currently running inside the place. When a suspended place is resumed, all agents inside this place are also resumed.)

**Agency-related methods**

The interface `IAgentSystem` provides the following agency-related methods:

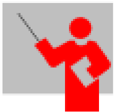
- `addSystemListener(...)`: This method enables a software component (e.g., an agent) to add a listener to an agency. The listener is notified about specific events occurring inside the attached agency, such as the cre-

ation, state change, and removal of agents and places. Please refer to Section 9.11.5 for detailed information about listening to agencies.

- `getInfo()`: This method returns the `AgentSystemInfo` object of the agency.
- `hasPersistence()`: This method provides information about whether the agency supports persistence or not.
- `removeSystemListener(...)`: This method removes an attached listener from an agency. Please refer to Section 9.11.5 for detailed information about listening to agencies.

### 9.11.3 Local Access

An agent can get access to the functionality of the local agency via the method `getAgentSystem()` which is provided by the agent's superclass `de.ikv.grasshopper.agent.Agent`. This method returns a reference to the interface `IAgentSystem`.



The following example code has been extracted from Example 9, Section 9.4.2. In this code fragment, the `ClientAgent` gets a reference to the local agency (interface `IAgentSystem`) in order to create an instance of the class `ServerAgent`.

```
ISserverAgent serverProxy;
public void init(Object[] creationArgs) {
  // Get proxy of local agency.
  agencyProxy = getAgentSystem();
  // Create the server agent.
  try {
    serverInfo =
      agencyProxy.createAgent(
            "ServerAgent",
            getInfo().getCodebase(),
            "InformationDesk",
            null);
  }
  catch (AgentCreationFailedException e) {
    System.out.println("## ClientAgent: Creation\\
    of server agent failed.");
}
```

### 9.11.4 Remote Access

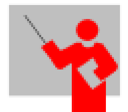
In order to contact a remote agency, an agent has to create an agency proxy. Similar to the creation of agent proxies (see Section 9.3), the agency that is to be contacted must be addressed correctly.

In contrast to the creation of an agent proxy which always requires the provision of the agent's identifier, the creation of an agency proxy can be performed by simply specifying the agency's name as well as the name of the host on which the agency is running. *Of course this assumes that all agencies running on the same host have different names.*

**Proxy creation**

For creating an agency proxy, the (client) agent uses the newInstance(...) method of the class de.ikv.grasshopper.communication.ProxyGenerator. As explained in Section 9.3, the second parameter of this method requires the identification of the component that is to be associated with the proxy. Concerning agency proxies, this parameter may be initialized in two different ways:

- The parameter may be initialized with the agency's identifier which can be retrieved from the agency's AgentSystemInfo object (method getIdentifier()). In order to get the AgentSystemInfo object, the client has to contact a running agency domain service via its listAgencies(...) method, assuming that the demanded agency is registered at this service.
- If the client knows the agency's name as well as the name of the host on which the agency is running, these two components can be used instead of the identifier. For this purpose, the host name and agency name have to be written into a String object, separated by a slash character: <hostName>/<agencyName>.
  If the client knows the complete address of the agency in terms of a GrasshopperAddress object (which is required for generating a proxy if no agency domain service is available), the client can invoke the method generateAgentSystemId() on the GrasshopperAddress object in order to get the <hostName>/<agencyName> string.



```
GrasshopperAddress agencyAddress = ...;
// Get proxy of remote agency.
agencyProxy = (IAgentSystem)
  ProxyGenerator.newInstance(
        IAgentSystem.class,
        agencyAddress.generateAgentSystemId(),
        agencyAddress);
// Invoke method on agency proxy
```

```
remoteAgents = agencyProxy.listAgents(
  new SearchFilter());
```

### 9.11.5 Listening to Agencies

Grasshopper agencies enable locally or remotely running software components to listen to internal events. This can be achieved by registering a listener object at an agency. In the following, the agency at which the listener object is registered is called *destination agency.* The *listening component*, i.e., the component that registers the listener at the destination agency and that wants to be notified about occurring events, may run at a another location, called *source location* or *source side* in the scope of this section.

The destination agency automatically notifies all registered listener objects about the occurrence of the following events:

- Agent creation
- Agent state change (suspension / resumption)
- Agent removal
- Place creation
- Place state change (suspension / resumption)
- Place removal

**Listener objects**

A listener object is an instance of a Java class that implements the interface `de.ikv.grasshopper.agency.ISystemListener`. This interface provides a set of methods where each method is associated with one of the events mentioned above. The agency automatically invokes one of these methods on all registered listeners when the corresponding event occurs.

**Methods for detecting events**

- `agentAdded(AgentInfo info)`
- `agentChanged(AgentInfo info)`
- `agentRemoved(AgentInfo info)`
- `placeAdded(PlaceInfo info)`
- `placeChanged(PlaceInfo info)`
- `placeRemoved(PlaceInfo info)`

The parameter `AgentInfo` or `PlaceInfo`, respectively, provides information about the agent or place that is associated with the occurred event.

**beforeRemove()**

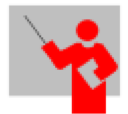
- `beforeRemove()`: Beside the event-detecting methods listed above, a system listener class has to implement the method `beforeRemove()`.

Similar to the `beforeRemove()` method of Grasshopper agents, this listener method is automatically called before the listener object is removed. Inside the method, the listener may prepare its removal, e.g., by releasing occupied resources.

- `getIdentifier()`: Finally, the method `getIdentifier()` has to be implemented by your listener class. This method has to return the identifier of the listener which is an instance of the class `de.ikv.grasshopper.type.Identifier`. Please generate the identifier during the listener's creation, and use „listener" as parameter value of the Identifier's constructor. The variable that maintains the listener identifier should be an instance variable of your listener class. Note that this identifier has the same structure as agent identifiers as described in Section 5.1.

**getIdentifier()**

Example:



```
class MyListener implements ISystemListener {
  Identifier listenerId;
  public MyListener() {
    listenerId = new Identifier(„listener");
  }
  public Identifier getIdentifier() {
    return listenerId;
  }
  public void beforeRemove() {
    ...
  }
  agentAdded(...) {...}
  agentChanged(...) {...}
  agentRemoved(...) {...}
  placeAdded(...) {...}
  placeChanged(...) {...}
  placeRemoved(...) {...}
}
```

**Registering listeners**

A listener object is registered at an agency via the method `addSystemListener(...)` which is provided by the agency's interface `de.ikv.grasshopper.agency.IAgentSystem`. This method is implemented with two different signatures:

`void addSystemListener(ISystemListener listener)`

A previously created listener object is transferred to the agency.

By using the Java reflection mechanism, the listener object is transferred *by value* to the demanded agency. In order to enable the destination agency to instantiate the listener, the listener class as well as all classes used by the listener class have to be inserted into the classpath environment setting of the destination agency. If this prerequisite is not fulfilled,

the listener should be added by using the method signature described below.

```
Identifier addSystemListener(
                java.lang.String className,
                java.lang.String codeBase,
                java.lang.Object[] arguments)
```

By using this signature, the listener is not created previously to the method invocation at the source side. Instead, the listener's class name, code base, and constructor parameters (if required) are specified, and the destination agency uses this information to create the listener object. Since a code base is given, the listener class need not be inserted in the destination agency's classpath. Instead, all required classes are retrieved via the Java class loading mechanism.

Note that the destination agency creates the listener object by means of a constructor that requires an object array (`Object[]`) as parameter. Thus, the listener class must implement such a constructor in order to enable the destination agency to create it. (In contrast to this, the method `addSystemListener(ISystemListener)` allows a listener object to be created by means of any individual constructor, since in this case the listener creation is performed at the source side.)

Usually, the creation of an agency listener object is initiated by a component that wants to be notified about specific events occurring at the destination agency. This *listening component* may reside at the same or a remote location, compared to the location of the destination agency. Since the listener object is running inside the destination agency, it has to establish a communication connection to the listening component in order to forward event notifications. This can be achieved by creating a proxy of the listening component and invoking methods on this proxy due to occurring events.

**Listening mechanism**

Figure 16 shows the general process of establishing a listener connection be-

tween an agent (i.e., the listening component) and a remote destination agency.

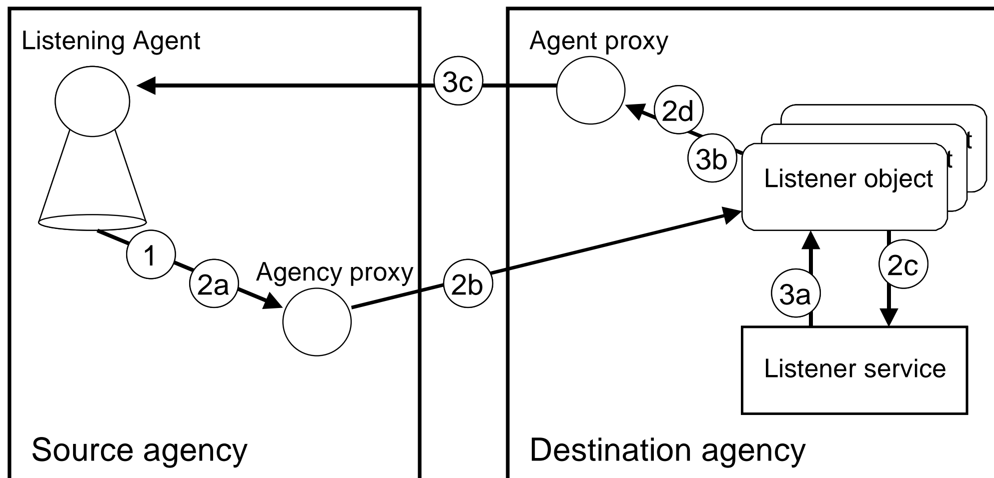



Figure 16: Listening Mechanism for Agencies

1. The listening agent creates a proxy of the destination agency as explained in Section 9.11.4.
2. The agent registers a listener object at the destination agency by invoking the `addSystemListener(...)` method on the agency proxy (2a, 2b). The listener object is automatically connected to the listener service (2c) of the destination agency. In order to forward event notifications to the listening agent, the listener object has to create a proxy of the agent (2d). Required information, such as the agent's identifier, location, and server interface name, must have been provided to the listener object as constructor arguments.
3. From now on, the listener object is notified by the listener service about occurring events, i.e., the creation, state change, and removal of an agent or place (3a). The listener object forwards corresponding event notifications to the listening agent by invoking a method on the agent's proxy (3b, 3c).

### 9.11.6 Example: AgencyClientAgent

The following example scenario consists of three classes/interfaces:

- `AgencyClientAgent`: This class represents the listening agent, i.e., the agent that wants to be notified about events occurring inside the local and inside a remote agency.
- `IListeningAgent`: This interface is implemented by the AgencyCli-

entAgent and used by the listener object in order to create a proxy of the AgencyClientAgent for the purpose of forwarding event notifications.

- `GHListener`: This class realizes the actual listener object by implementing the interface `de.ikv.grasshopper.agency.ISystemListener`.

### Class AgencyClientAgent

**Instance variables**

The class AgencyClientAgent maintains the following instance variables:

- `agencyAddress`: This variable is initialized with a creation argument inside the agent's `init(...)` method. The variable maintains the address of a remote agency at which a listener object is to be registered.
- `agencyProxy`: This variable is initialized with a proxy of a remote agency. The AgencyClientAgent uses this proxy in order to register a listener object at the remote agency.
- `remoteListenerId`: This variable maintains the identifier of the remotely registered listener object. This identifier is needed to enable the agent to remove the listener from the remote agency when it is not needed anymore.
- `localListener`: This variable maintains a reference to a locally registered listener.

**init(...)**

Inside its `init(...)` method, the AgencyClientAgent retrieves a creation argument that has to be specified by the user. This argument maintains the address of the remote agency at which a listener object is to be registered.

**beforeRemove()**

The `beforeRemove()` method is automatically called by the agency before the agent is removed (please refer to Section 4.2 for more information). The AgencyClientAgent uses this method to remove all previously attached listeners.

**live()**

Inside its `live()` method, the AgencyClientAgent checks whether an agency address has been provided as creation argument. If not, the agent removes itself at once.

If the user has specified a valid agency address, the agent invokes the method `newInstance(...)` of the class `ProxyGenerator` in order to create a proxy of the remote agency's `IAgentSystem` interface. In order to establish a communication connection via the proxy, the agent has to identify the demanded agency by specifying the agency's name as well as the name of the host on which the agency is running. As explained in Section 9.11.4, this set of information can be retrieved by calling the method `generateAgentSystemId()` on the `GrasshopperAddress` object that maintains the

complete agency address. The last parameter of the `newInstance(...)` method specifies the complete agency address that has been provided by the user as creation argument of the AgencyClientAgent. (In the case of an available agency domain service, the `newInstance(...)` method can also be invoked without specifying the complete agency address. In this case, agency name and host name are sufficient for identification purposes.)

By invoking methods on the created proxy, the AgencyClientAgent requests the following information from the contacted agency: its identifier, name, type location, and a list of all currently hosted agents. After this, the agent registers a listener object at the remote agency by calling the method `addSystemListener(...)` on the proxy.

The last action of the agent is to register a listener object at the local agency. Please have a look at both calls of the method `addSystemListener(...)`:

- The first call creates a listener object at the remote agency. The method parameters specify the name of the listener class, the code base from which the listener class can be retrieved (in this case, the agent's own code base is assumed to maintain the listener class), as well as arguments for the constructor of the listener class. As explained in Section 9.11.5, the remote agency tries to create the listener object via a constructor that uses an `Object` array as parameter. Thus, all required listener arguments are stored in a variable of the type `Object[]`.
  The method returns the identifier of the new listener object. The identifier is required for removing the listener later on (see the `beforeRemove()` method of the AgencyClientAgent).
  Note that also the second signature of the method `addSystemListener(...)` could have been used for registering the listener object at the remote agency. However, in this case, the listener class as well as all classes referenced by the listener class (such as the class IListeningAgent) would have to be stored in the Java CLASSPATH environment setting of the remote agency.

- The second call registers a listener object at the agency in which the AgencyClientAgent is running. In contrast to the first method call, the listener object is created by the AgencyClientAgent itself instead of the agency where the listener is to be registered. The agent creates the listener object by using an individual constructor, so that the required arguments do not have to be converted to `Object[]`. After its creation, the listener object itself is used as parameter of the `addSystemListener(...)` method. Note that in this case the method does not return a listener identifier. In order to remove the listener later on, its object reference is used for identi-

fication purposes (see the `beforeRemove()` method of the AgencyClientAgent).

Since no code base is specified when the listener is registered, the listener class as well as the class IListeningAgent have to be maintained in the Java CLASSPATH environment setting of the local agency. (The class IListeningAgent is needed because the listener object tries to create a proxy of the AgencyClientAgent after being registered at the local agency.)

**eventDetected(...)**

The method `eventDetected(...)` is defined in the agent's server interface IListeningAgent. By implementing this interface, the AgencyClientAgent is accessible via the communication service. The created listener objects invoke this method (via an agent proxy) in order to inform the AgencyClientAgent about events that occur inside the monitored agencies.

Example 21: AgencyClientAgent

```
package examples.simple;

import examples.util.*;
import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.util.*;
import de.ikv.grasshopper.communication.*;

// This class realizes an agent that contacts a remote
// agency.
public class AgencyClientAgent extends StationaryAgent
                implements IListeningAgent
{
  GrasshopperAddress agencyAddress;
  IAgentSystem agencyProxy;
  Identifier remoteListenerId;
  GHListener localListener;

  // Creation argument:
  //   args[0] =  address of the agency that has to be
  //              contacted by the AgencyAccessAgent.
  public void init(Object[] args) {
    // The init method expects the address of an agency
    // as argument.
    // If no argument is provided, the agent removes
    // itself at the beginning of its live method.
    if (args == null || args.length < 1)
      agencyAddress = null;
    else
```

```
    agencyAddress =
      new GrasshopperAddress((String) args[0]);
}

public String getName() {
  return "AgencyClientAgent";
}

// This method is automatically called before the
// agent is removed.
// The agent uses this method to remove the previously
// attached listener(s).
public void beforeRemove() {
  if (agencyAddress == null)
    log("No agency address specified. Removing...");
  if (remoteListenerId != null) {
    log("Removing remote listener...");
    try {
      // Remove remote listener.
        agencyProxy.removeSystemListener(
      remoteListenerId);
    }
    catch (ListenerRemovalFailedException e) {
      log("Listener removal failed: ", e);
    }
  }
  if (localListener != null) {
    log("Removing local listener...");
    // Remove local listener
    getAgentSystem().removeSystemListener(
      localListener);
  }
  log("Removing myself...");
}

public void live() {
  AgentInfo remoteAgents[];
  GHListener agencyListener;

  if (agencyAddress == null)
    // No agency address has been specified as
    // creation argument.
    try {
      remove();
    }
    catch (Exception e) {
      log("Cannot remove myself. ", e);
    }
```

```
log("Contacting agency '" +
  agencyAddress.toString() + "'.");
// Create proxy of remote agency
agencyProxy = (IAgentSystem)
  ProxyGenerator.newInstance(
    IAgentSystem.class,
    agencyAddress.generateAgentSystemId(),
    agencyAddress);
// Print some information about the contacted
// agency
log("Agency contacted.");
log("  Identifier: " +
  agencyProxy.getInfo().getIdentifier());
log("  Name       : " +
  agencyProxy.getInfo().getName());
log("  Type       : " +
  agencyProxy.getInfo().getType());
log("  Location   : " +
  agencyProxy.getInfo().getLocation());
log("  Hosted agents:");
if (agencyProxy != null) {
  // List all agents hosted by the remote agency.
  remoteAgents =
    agencyProxy.listAgents(new SearchFilter());
  if (remoteAgents != null)
    for (int i = 0; i < remoteAgents.length; i++)
      log("      " +
        remoteAgents[i].getAgentPresentation().
        getAgentName() + " - " +
        remoteAgents[i].getIdentifier());

  // Register a listener at the remote agency
  try {
    Object[] listenerArgs = new Object[4];
    // The following objects are constructor
    // arguments for the listener object
    listenerArgs[0] = (Identifier)
      getInfo().getIdentifier();
    listenerArgs[1] = (GrasshopperAddress)
      getInfo().getHome();
    listenerArgs[2] = (String) "Remote";
    // Add listener
    remoteListenerId =
      agencyProxy.addSystemListener(
        "examples.util.GHListener",
        getInfo().getCodebase(), listenerArgs);
  }
  catch (ListenerCreationFailedException e) {
```

```
        log("Cannot listen to " +
          agencyProxy.getInfo().getName() +
          ". ", e);
      }
      if (remoteListenerId != null)
        log("Listener added to remote agency '" +
          agencyProxy.getInfo().getName() +
          "'. Listening...");
    }
    else
      log("Agency '" + agencyAddress +
        "' not found.");

    // Register a listener at the local agency.
    // This is only possible if the classes
    // GHListener as well as IAgencyClientAgent are
    // contained in the classpath.
    localListener = new
      GHListener(getInfo().getIdentifier(),
        getInfo().getHome(), "Local");
    try {
      getAgentSystem().addSystemListener(
        localListener);
    }
    catch (Throwable e) {
      log("Cannot register local listener. ", e);
      localListener = null;
    }
  }

  // The following method is called by the listener
  // object(s) due to events occurring inside the
  // monitored agencies.
  public void eventDetected(String event) {
    log(event);
  }
}
```

#### Interface IListeningAgent

By means of this interface, the AgencyClientAgent is accessible via the Grasshopper communication service. The listener objects use the method `eventDetected(...)` defined by this interface in order to inform the agent about events that occur inside the monitored agencies (or, concerning Example 24 in Section 9.12.6, inside the monitored region registry).

Example 22: IListeningAgent

```
package examples.util;

import de.ikv.grasshopper.agent.IAgent;

// This interface is implemented by the following
// agents:
//    - examples.simple.AgencyClientAgent
//    - examples.simple.RegionClientAgent
// The method 'eventDetected' is called by a GHListener
// object due to a detected event. In this way, a
// listening agent is automatically
// informed about events occurring inside agencies or
// region registries.
public interface IListeningAgent
{
  public void eventDetected(String event);
}
```

### Class GHListener

By implementing the interface ISystemListener, this class realizes a listener that is able to monitor the events occurring inside Grasshopper agencies or region registries.

**Constructors**

The class provides two constructors:

```
public GHListener(
          String Identifier agentId,
          GrasshopperAddress agentLocation,
          String lName)
```

This (individual) constructor can be used if the listening object (i.e., the AgencyClientAgent or, concerning Example 24 in Section 9.12.6, the RegionClientAgent) creates the GHListener object by itself. In this case, the already created listener object is transferred to the demanded destination agency via the method addSystemListener(ISystemListener).
Concerning the AgencyClientAgent (see Example 21 above), the agent uses this constructor for creating the listener for the local agency.

```
public GHListener(Object[] creationArgs)
```

This constructor is automatically used by the destination agency if the listening object instructs the destination agency to create the listener by invoking the method addSystemListener(String, String, Object[]).
Concerning the AgencyClientAgent (see Example 21 above), the agent uses this constructor for registering a listener at the remote agency. The

RegionClientAgent (see Example 24 in Section 9.12.6) uses this constructor for registering a listener at a region registry.

Inside the constructor, the listener object creates a proxy of the AgencyClientAgent or RegionClientAgent, respectively. For this purpose, the constructor arguments provide sufficient information: the agent's identifier and its location. The agent's class name is hard coded in the call of the `newInstance(...)` method. Via the last constructor argument, the agent specifies a name for the listener object. The purpose of this name is just to enable the user to distinguish between the textual outputs of both listeners, since both are printed in the text console of the agency in which the agent is running.

**Listener methods**

The remaining methods of the class GHListener implement the interface ISystemListener. These methods are automatically called by the monitored agency when a corresponding event, such as the creation of a new agent, has occurred. Inside the methods, the listener object invokes the method `eventDetected(...)` of the AgencyClientAgent (or RegionClientAgent concerning Example 24) via the previously created agent proxy, in this way forwarding an event notification to the agent.

Example 23: GHListener

```
package examples.util;

import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.communication.*;

// This class realizes a system listener which can be
// used by Grasshopper agents in order to listen for
// events occurring inside agencies or region
// registries.
// The GHListener class is used by the following agents:
//    - examples.simple.AgencyClientAgent
//    - examples.simple.RegionClientAgent
public class GHListener implements ISystemListener
{
  IListeningAgent agentProxy;
  String listenerName;
  Identifier listenerId;

  // This constructor can be used when the listener
  // object is created at the source side and
  // transferred to the destination agency via the
  // method 'addSystemListener(ISystemListener)'.
  public GHListener(
            Identifier agentId,
```

```
            GrasshopperAddress agentLocation,
            String lName) {
    listenerName = lName;

    // Create a proxy of the agent that has added this
    // listener object to the agency.
    // The proxy is used by the listener for forwarding
    // event notifications to the agent.
    agentProxy = (IListeningAgent)
      ProxyGenerator.newInstance(
        IListeningAgent.class,
        (Identifier) agentId,
        (GrasshopperAddress) agentLocation);
    System.out.println("## " + listenerName +
      " GHListener: Created.");
    if (agentProxy != null)
      System.out.println("##   Agent proxy created.");
    else
      System.out.println("##    Could not create \\
      agent proxy.");

    // Generate listener identifier
    listenerId = new Identifier("listener");
  }

  // This constructor is automatically used by the
  // destination agency is the listener is added via
  // the method
  // 'addSystemListener(String, String, Object[])'
  public GHListener(Object[] creationArgs) {
    listenerName = (String) creationArgs[2];
    // Create a proxy of the agent that has added this
    // listener object to the agency.
    // The proxy is used by the listener for forwarding
    // event notifications to the agent.
    agentProxy = (IListeningAgent)
      ProxyGenerator.newInstance(
        IListeningAgent.class,
        (Identifier) creationArgs[0],
        (GrasshopperAddress) creationArgs[1]);
    System.out.println("## " + listenerName +
      " GHListener: Created.");
    if (agentProxy != null)
      System.out.println("##   Agent proxy created.");
    else
      System.out.println("##    Could not create \\
      agent proxy.");
  }
```

```
// The following methods are automatically called by
// the listener service when a corresponding event
// occurs inside the agency or region registry to
// which the listener has been attached.

public void agencyAdded(AgentSystemInfo info) {
  // This event can only occur when the listener is
  // attached to an agency domain service.
  System.out.println("## " + listenerName +
    " GHListener: Forwarding agency creation\\
    event...");
  agentProxy.eventDetected("## " + listenerName +
    " GHListener: Creation of agency '" +
    info.getName() + "' detected.");
}

public void agencyRemoved(AgentSystemInfo info) {
  // This event can only occur when the listener is
  // attached to an agency domain service.
  System.out.println("## " + listenerName +
    " GHListener: Forwarding agency removal\\
    event...");
  agentProxy.eventDetected("## " + listenerName +
    " GHListener: Removal of agency '" +
    info.getName() + "' detected.");
}

public void agentAdded(AgentInfo info) {
  System.out.println("## " + listenerName +
  " GHListener: Forwarding agent creation event...");
  agentProxy.eventDetected("## " + listenerName +
    " GHListener: Creation of agent '" +
    info.getAgentPresentation().getAgentName() +
    "' detected.");
}

public void agentChanged(AgentInfo info) {
  System.out.println("## " + listenerName +
    " GHListener: Forwarding agent change event...");
  agentProxy.eventDetected("## " + listenerName +
    " GHListener: Change of agent '" +
    info.getAgentPresentation().getAgentName() +
    "' detected. New state = " + info.getState());
}

public void agentRemoved(AgentInfo info) {
  System.out.println("## " + listenerName +
```

```
        " GHListener: Forwarding agent removal\\
        event...");
      agentProxy.eventDetected("## " + listenerName +
        " GHListener: Removal of agent '" +
        info.getAgentPresentation().getAgentName() +
        "' detected.");
    }

    public void placeAdded(PlaceInfo info) {
      System.out.println("## " + listenerName +
        " GHListener: Forwarding place creation\\
        event...");
      agentProxy.eventDetected("## " + listenerName +
        " GHListener: Creation of place '" +
        info.getName() + "' detected.");
    }

    public void placeChanged(PlaceInfo info) {
      System.out.println("## " + listenerName +
        " GHListener: Forwarding place change event...");
      agentProxy.eventDetected("## " + listenerName +
        " GHListener: Change of place '" +
        info.getName() + "' detected. New state = " +
        info.getState());
    }

    public void placeRemoved(PlaceInfo info) {
      System.out.println("## " + listenerName +
        " GHListener: Forwarding place removal\\
        event...");
      agentProxy.eventDetected("## " + listenerName +
        " GHListener: Removal of place '" +
        info.getName() + "' detected.");
    }

    public void beforeRemove() {
      System.out.println("## " + listenerName +
        " GHListener: Removing...");
    }

    public Identifier getIdentifier() {
      return listenerId;
    }
  }
```

Requirements:

- Two running agencies

- In order to enable the agent to register a listener object at the local agency, the classes `GHListener` and `IAgencyClientAgent` have to be inserted in the Java CLASSPATH environment setting of the agency in which the AgencyClientAgent is started. Note that this condition need not be fulfilled if the agent uses the other signature of the method `addSystemListener(...)` for registering a listener at the local agency. Detailed information about the two signatures is provided above, inside the description of the `live()` method of the AgencyClientAgent.
  If one of the classes `GHListener` and `IAgencyClientAgent` are missing in the CLASSPATH, you will see that the agent is still able to register the listener object at the remote agency, while the local listener registration fails.

Running the Example:

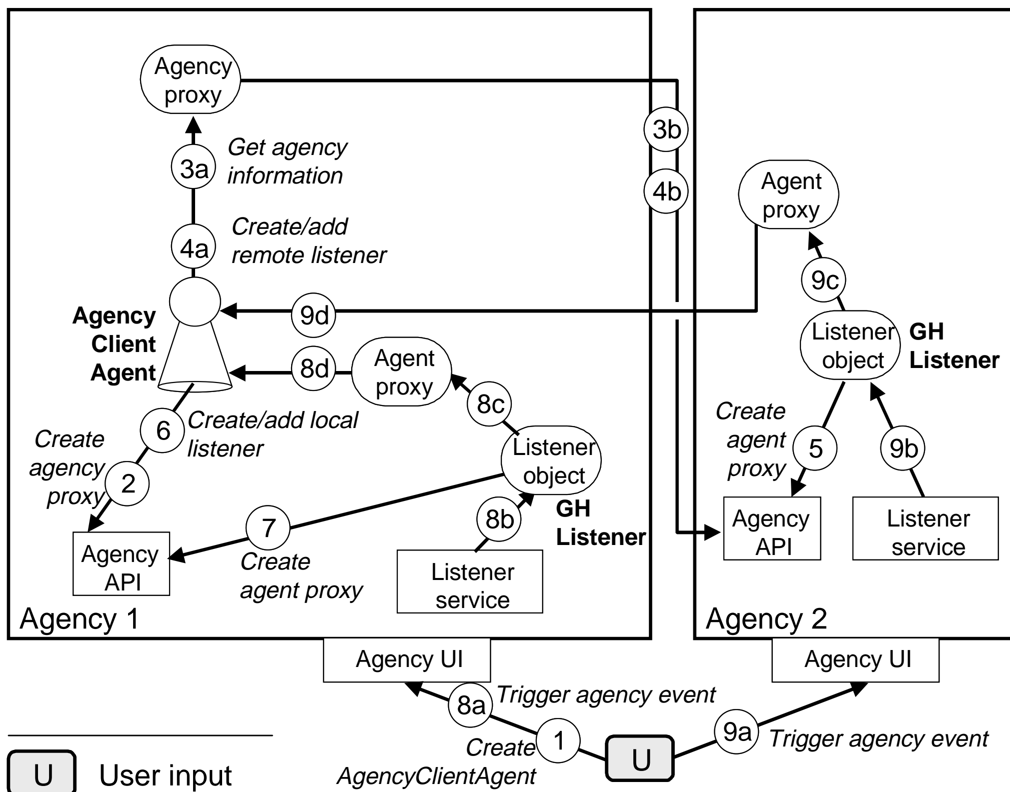



Figure 17: AgencyClientAgent Scenario

Create some simple agents (e.g., the HelloAgent and PrintInfoAgent) in each running agency.

Create the AgencyClientAgent in one of the running agencies, specifying the address of the remote agency as creation argument (1).

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.simple.AgencyClientAgent socket://
Host1:7000/Agency1
```

Note that you have to adapt the agency address in the line above to the address of your concrete agency. You can determine this address via the command 'status' of the agency's text console. If you have an agency domain service running and connected to both agencies, the address can be specified just in terms of the host name and agency name, separated by a slash character: `Host1/Agency1`.

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

Look at the output of the AgencyClientAgent in the local agency console. After creating a proxy of the previously specified remote agency (2), the agent will print a list of all agents that are running inside this agency (3). After this, the agent adds a listener to the remote agency (4) and another one to the local agency (6). Both listeners create a proxy of the AgencyClientAgent (5, 7) in order to be able to forward event notifications.

Now create, suspend, resume, and remove agents and places inside both agencies via the agencies' user interfaces (8a, 9a), and have a look at the corresponding console windows. The listener services of the agencies will detect the events and send a notification to the attached listener object (8b, 9b). The listeners in turn will use their agent proxies (8c, 9c) in order to forward the notifications to the AgencyClientAgent (8d, 9d).

### 9.11.7 Summary

- Every Grasshopper agency carries information about itself that may be accessed by other entities. This information is maintained by an instance of the class `de.ikv.grasshopper.type.AgentSystemInfo`. The maintained information includes an agency's identifier, address, name, and type.
- An agency's functionality is accessible via the interface `de.ikv.grasshopper.agency.IAgentSystem`. Agents can get a reference of the `IAgentSystem` interface of their local agency via their superclass `de.ikv.grasshopper.agency.Agent`. Access to remote agencies can be achieved by creating an agency proxy.
- Agents can register listener objects at agencies in order to be notified

about agency-internal events.

## 9.12 Accessing an Agency Domain Service

An agency domain service can be contacted by an agent in two different ways:

1. Via the interface `de.ikv.grasshopper.agency.IRegion` of the local agency, i.e., of the agency in which the agent is currently running:

   The agent can chose between contacting the agency domain service at which the local agency is registered or contacting any other available agency domain service. In the latter case, the agent has to specify the address of the demanded agency domain service when invoking a method on the `IRegion` interface. In both cases, the agent accesses the service via the local `IRegion` interface. Note that, in contrast to accessing a remote agency, there is no need for an agent to create an own proxy of an agency domain service. An agent can retrieve a reference to the `IRegion` interface by invoking the method `getRegion()` of its superclass `Agent` (which is similar to the access of the local `IAgentSystem` interface, as described in Section 9.11.3).

   Note that the agent uses the same methods of the `IRegion` interface, independent of whether a Grasshopper region registry or an LDAP server is contacted.

   Detailed information about the available methods of the IRegion interface are provided in Section 9.12.1.

2. Via a proxy of the interface `de.ikv.grasshopper.agency.RegionRegistration`:

   The `IRegionRegistration` interface is explicitly associated with Grasshopper region registries. That means, in contrast to the `IRegion` interface provided by Grasshopper agencies, no LDAP server can be contacted. An agent can get access to the `IRegionRegistration` interface in the usual way, i.e., by creating a proxy object via the method `newInstance(...)` of the class `de.ikv.grasshopper.communication.ProxyGenerator`, (cf. Section 9.3).

   The `IRegionRegistration` interface is of particular importance if an agent wants to add a listener to a region registry, since this cannot be achieved via the `IRegion` interface. For detailed information about this aspect, please refer to Section 9.12.5. If an agent just wants to invoke the list methods of an agency domain service, it is, from an implementation point of view, more comfortable to use the `IRegion` interface of the local

agency.

Detailed information about the available methods of the `IRegionRegistration` interface are provided in Section 9.12.2.

### 9.12.1 Interface IRegion

**Agent-related methods**

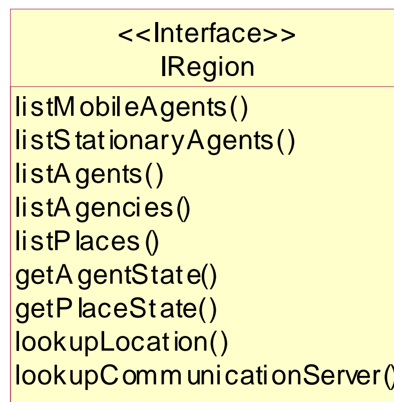



Figure 18: IRegion Class Diagram

The `IRegion` interface provides the following agent-related methods:

- `getAgentState(...)`: This method returns the current state of a specific agent.
- `listAgents(...)`: This method returns a list of agents that are registered at the contacted agency domain service. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- `listMobileAgents(...)`: This method returns a list of mobile agents that are registered at the contacted agency domain service. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- `listStationaryAgents(...)`: This method returns a list of stationary agents that are registered at the contacted agency domain service. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- `lookupLocation(...)`: This method returns the current location of a specific agent.

**Place-related methods**

The IRegion interface provides the following place-related methods:

- getPlaceState(...): This method returns the state of a specific place.
- listPlaces(...): This method returns a list of places that are registered at the contacted agency domain service. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.

**Agency-related methods**

The IRegion interface provides the following agency-related methods:

- listAgencies(...): This method returns a list of agencies that are registered at the contacted agency domain service. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- lookupCommunicationServer(...): This method requires a String parameter, specifying the agency's name as well as the name of the host on which the agency is running. This information has to be provided with the following syntax: <hostName>/<agencyName>.
  If the invoking entity knows the complete address of the agency in terms of a GrasshopperAddress object (which is required for generating a proxy if no agency domain service is available), the client can invoke the method generateAgentSystemId() on the GrasshopperAddress object in order to get the <hostName>/<agencyName> string.

### 9.12.2 Interface IRegionRegistration

The interface IRegionRegistration inherits the interfaces ISystemListenerProvider, enabling the registration and de-registration of listener objects, and the interface IDirectoryService, providing access to the registration information of the registry.

Figure 19: IRegionRegistration Class Diagram

**Agent-related methods**

The `IRegionRegistration` interface provides the following agent-related methods:

- `getAgentState(...)`: This method returns the current state of a specific agent.
- `listAgents(...)`: This method returns a list of agents that are registered at the contacted region registry. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- `listMobileAgents(...)`: This method returns a list of mobile agents that are registered at the contacted region registry. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- `listStationaryAgents(...)`: This method returns a list of stationary agents that are registered at the contacted region registry. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- `lookupLocation(...)`: This method returns the current location of a specific agent.

**Place-related methods**

The `IRegionRegistration` interface provides the following place-related methods:

- `getPlaceState(...)`: This method returns the state of a specific place.
- `listPlaces(...)`: This method returns a list of places that are registered at the contacted region registry. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.

**Agency-related methods**

The `IRegionRegistration` interface provides the following agency-related methods:

- `listAgencies(...)`: This method returns a list of agencies that are registered at the contacted region registry. The search can be restricted by setting filters. Please refer to Section 9.13 for detailed information about searching Grasshopper components.
- `lookupCommunicationServer(...)`: This method requires a `String` parameter, specifying the agency's name as well as the name of the host on which the agency is running. This information has to be provided with the following syntax: `<hostName>/<agencyName>`.
  If the invoking entity knows the complete address of the agency in terms of a `GrasshopperAddress` object (which is required for generating a proxy if no agency domain service is available), the client can invoke the method `generateAgentSystemId()` on the `GrasshopperAddress` object in order to get the `<hostName>/<agencyName>` string.

**Registry-related methods**

The `IRegionRegistration` interface provides the following registry-related methods:

- `addSystemListener(...)`: This method enables a software component (e.g., an agent) to add a listener to a region registry. The listener is notified about specific events occurring inside the attached registry, such as the creation, state change, and removal of agencies, agents, and places. Please refer to Section 9.12.5 for detailed information about listening to region registries.
- `removeSystemListener(...)`: This method removes an attached listener from a region registry. Please refer to Section 9.12.5 for detailed information about listening to region registries.

### 9.12.3 Local Access

Similar to Section 9.11.3 and Section 9.11.4 where a separation was made between the local and remote access of an agency, there are two ways of contacting an agency domain service. However, it has to be noted that the local access described in the current section allows the access of both Grasshopper region registries as well as LDAP servers, while the remote access described in Section 9.12.4 is restricted to the access of region registries.

Concerning the local access, an agent does not need to create a proxy of an agency domain service, since the functionality of this service is provided via the interface `IRegion` of the local agency. The actual access of the domain service is performed by the local agency and hidden behind the `IRegion` interface.

An agent can retrieve a reference to the `IRegion` interface via the method `getRegion()` which is provided by the agent's superclass `de.ikv.grasshopper.agent.Agent`.

Note that the registration of an agency at an agency domain service is optional. If an agent wants to contact an agency domain service in the case that the hosting agency is not registered at this service, the agent has to provide the address of the domain service when invoking a method on the `IRegion` interface. Another case in which an agent has to specify the domain service address is that the hosting agency has access to a specific agency domain service, but the agent wants to contact a different domain service.

When an agent specifies the address of an agency domain service, the agent has to know whether the service is represented by a region registry or by an LDAP server, since their address schemes are slightly different:

**Addressing a region registry**

If the demanded agency domain service is a Grasshopper region registry, its address is a usual Grasshopper address as described in Section 5.4. Represented as String, the address has the following format:

<protocol>://<hostName>:<portNumber>/<regionRegistryName>

**Addressing an LDAP server**

If the demanded agency domain service is an LDAP server, its address has the following format:

`ldap://<hostName>:portNumber>/<distinguishedName>`

As shown above, the protocol type must be set to `ldap`, and a distinguished name has to be provided, such as '`o=IKV,c=DE`'.

The following example code, extracted from Example 13 in Section 9.6.2, shows how a client agent can access an agency domain service via the methods of the IRegion interface of the local agency. Concerning the code below, the default agency domain service is to be contacted, i.e., the domain service at which the local agency is registered. Therefore, no address is specified, and the value null is used as first parameter of the listAgents(...) method.

```
// Get proxy of local agency
regionProxy = getRegion();
// Look for the server agent in the
// agency domain service
SearchFilter filter = new SearchFilter(
    SearchFilter.NAME+"=AsyncServerAgent");
serverInfos = regionProxy.listAgents(null, filter);
```

## 9.12.4 Remote Access

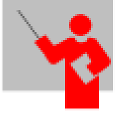
In order to contact a remote region registry, an agent has to create a registry proxy, based on the interface IRegionRegistration (see Section 9.12.2).

Similar to the creation of agent proxies (see Section 9.3), the registry that is to be contacted must be addressed correctly. In contrast to the creation of an agent proxy which always requires the provision of the agent's identifier, the creation of a registry proxy can be performed by simply specifying the registry's name as well as the name of the host on which the registry is running.

**Proxy creation**

For creating a region registry proxy, the (client) agent uses the newInstance(...) method of the class de.ikv.grasshopper.communication.ProxyGenerator. As explained in Section 9.3, the second parameter of this method requires the identification of the component that is to be associated with the proxy.

The identification of a region registry can be achieved by specifying the registry's name as well as the name of the host on which the registry is running. For this purpose, the host name and registry name have to be written into a String object, separated by a slash character: <hostName>/<registryName>.
If the client knows the complete address of the registry in terms of a GrasshopperAddress object, the client can invoke the method generateRegionId() on the GrasshopperAddress object in order to get the <hostName>/<registryName> string.



```
GrasshopperAddress registryAddress = ...;
// Get proxy of region registry.
registryProxy = (IRegionRegistration)
  ProxyGenerator.newInstance(
        IRegionRegistration.class,
        registryAddress.generateRegionId(),
        registryAddress);
// Invoke method on registry proxy
remoteAgents = registryProxy.listAgents(
  new SearchFilter());
```

## 9.12.5 Listening to Region Registries

Grasshopper region registries enable locally or remotely running software components to listen to internal events. This can be achieved by registering a listener object at a registry. The region registry automatically notifies all registered listener objects about the occurrence of the following events:

- Agency creation
- Agency removal
- Agent creation
- Agent state change (suspension / resumption)
- Agent removal
- Place creation
- Place state change (suspension / resumption)
- Place removal

**Listener objects**

A listener object is an instance of a Java class that implements the interface `de.ikv.grasshopper.agency.ISystemListener`. This interface provides a set of methods where each method is associated with one of the events mentioned above. The region registry automatically invokes one of these methods on all registered listeners when the corresponding event occurs.

**Methods for event detection**

- `agencyAdded(AgentSystemInfo info)`
- `agencyRemoved(AgentSystemInfo info)`
- `agentAdded(AgentInfo info)`
- `agentChanged(AgentInfo info)`
- `agentRemoved(AgentInfo info)`
- `placeAdded(PlaceInfo info)`

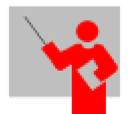
- `placeChanged(PlaceInfo info)`
- `placeRemoved(PlaceInfo info)`

The parameter `AgentSystemInfo`, `AgentInfo`, or `PlaceInfo`, respectively, provides information about the agency, agent, or place that is associated with the occurred event.

**beforeRemove()**

- `beforeRemove()`: Beside the event-detecting methods listed above, a system listener class has to implement the method `beforeRemove()`. Similar to the `beforeRemove()` method of Grasshopper agents, this listener method is automatically called before the listener object is removed. Inside the method, the listener may prepare its removal, e.g., by releasing occupied resources.
- `getIdentifier()`: Finally, the method `getIdentifier()` has to be implemented by your listener class. This method has to return the identifier of the listener which is an instance of the class `de.ikv.grasshopper.type.Identifier`. Please generate the identifier during the listener's creation, and use „listener" as parameter value of the Identifier's constructor. The variable that maintains the listener identifier should be an instance variable of your listener class. Note that this identifier has the same structure as agent identifiers as described in Section 5.1.

  Example:



```
class MyListener implements ISystemListener {
  Identifier listenerId;
  public MyListener() {
    listenerId = new Identifier(„listener");
  }
  public Identifier getIdentifier() {
    return listenerId;
  }
  public void beforeRemove() {
    ...
  }
  agencyAdded(...) {...}
  agencyRemoved(...) {...}
  agentAdded(...) {...}
  agentChanged(...) {...}
  agentRemoved(...) {...}
  placeAdded(...) {...}
  placeChanged(...) {...}
  placeRemoved(...) {...}
}
```

**Registering listeners**

A listener object is registered at a region registry via the method `addSystemListener(...)` which is provided by the registry's interface

`de.ikv.grasshopper.agency.IRegionRegistration`. This method is implemented with two different signatures:

```
void addSystemListener(ISystemListener listener)
```

A previously created listener object is transferred to the region registry.

By using the Java reflection mechanism, the listener object is transferred *by value* to the demanded registry. In order to enable the registry to instantiate the listener, the listener class as well as all classes used by the listener class have to be inserted into the registry's classpath environment setting. If this prerequisite is not fulfilled, the listener should be added by using the method signature described below.

```
Identifier addSystemListener(
                java.lang.String className,
                java.lang.String codeBase,
                java.lang.Object[] arguments)
```

By using this signature, the listener is not created previously to the method invocation at the source side. Instead, the listener's class name, code base, and constructor parameters (if required) are specified, and the region registry uses this information to create the listener object. Since a code base is given, the listener class need not be inserted in the registry's classpath. Instead, all required classes are retrieved via the Java class loading mechanism.
Note that the region registry creates the listener object by means of a constructor that requires an object array (`Object[]`) as parameter. Thus, the listener class must implement such a constructor in order to enable the registry to create it. (In contrast to this, the method `addSystemListener(ISystemListener)` allows a listener object to be created by means of any individual constructor, since in this case the listener creation is performed at the source side.)

Usually, the creation of a listener object is initiated by a component that wants to be notified about specific events occurring at the region registry. Since the listener object is running inside the registry, it has to establish a communication connection to the listening component in order to forward event notifications. This can be achieved by creating a proxy of the listening component and invoking methods on this proxy due to occurring events.

**Listening mechanism**

Figure 16 shows the general process of establishing a listener connection be-

tween an agent (i.e., the listening component) and a region registry.

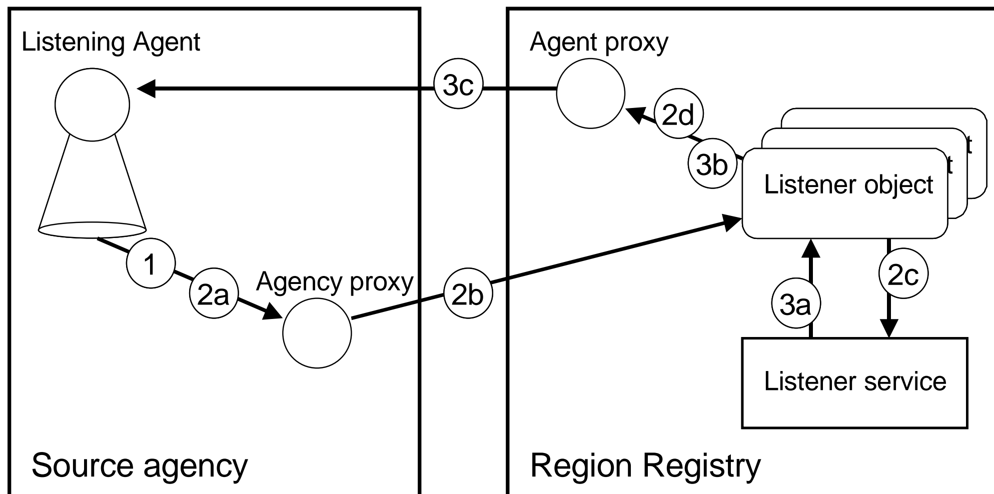



Figure 20: Listening Mechanism for Region Registries

1. The listening agent creates a proxy of the registry as explained in Section 9.12.4.
2. The agent registers a listener object at the registry by invoking the `addSystemListener(...)` method on the proxy (2a, 2b). The listener object is automatically connected to the listener service (2c) of the registry. In order to forward event notifications to the listening agent, the listener object has to create a proxy of the agent (2d). Required information, such as the agent's identifier, location, and server interface name, must have been provided to the listener object as constructor arguments.
3. From now on, the listener object is notified by the listener service about occurring events, i.e., the creation, state change, and removal of an agency, agent, or place (3a). The listener object forwards corresponding event notifications to the listening agent by invoking a method on the agent's proxy (3b, 3c).

### 9.12.6 Example: RegionClientAgent

The following example scenario consists of three classes/interfaces:

- `RegionClientAgent`: This class represents the listening agent, i.e., the agent that wants to be notified about events occurring inside a specific region registry.
- `IListeningAgent`: This interface is implemented by the RegionCli-

entAgent and used by the listener object in order to create a proxy of the RegionClientAgent for the purpose of forwarding event notifications.

- `GHListener`: This class realizes the actual listener object by implementing the interface `de.ikv.grasshopper.agency.ISystemListener`.

### Class RegionClientAgent

**Instance variables**

The class RegionClientAgent maintains the following instance variables:

- `adsAddress`: This variable is initialized with a creation argument inside the agent's `init(...)` method. The variable maintains the address of the agency domain service at which a listener object is to be registered.
- `adsProxy`: This variable is initialized with a reference of the `IRegion` interface of the local agency. The RegionClientAgent uses this interface in order to retrieve information from the domain service whose address has been provided by the user as creation argument of the agent.
- `registryProxy`: This variable is initialized with a proxy of a region registry, based on the interface `IRegionRegistration`. (Note that, for this purpose, the address provided by the user must refer to a Grasshopper region registry and not to an LDAP server.) The RegionClientAgent uses this proxy in order to retrieve information from the region registry whose address has been provided by the user. Besides, the RegionClientAgent uses this proxy in order to register a listener object at the registry.
- `listenerId`: This variable maintains the identifier of the remotely registered listener object. This identifier is needed to enable the agent to remove the listener from the region registry when it is not needed anymore.

**init(...)**

Inside its `init(...)` method, the RegionClientAgent retrieves a creation argument that has to be specified by the user. This argument maintains the address of an agency domain service. Note that, since the agent tries to register a listener at the domain service, it is recommended to specify the address of a region registry and not the address of an LDAP server.

**beforeRemove()**

The `beforeRemove()` method is automatically called by the agency before the agent is removed (please refer to Section 4.2 for more information). The RegionClientAgent uses this method to remove the previously attached listener.

**live()**

Inside its `live()` method, the RegionClientAgent gets a reference to the `IRegion` interface of the local agency. The agent uses this interface in order to retrieve a list of all agents that are registered at the agency domain service whose address has been previously specified by the user.

After this, the agent creates a region registry proxy, based on the interface `IRegionRegistration`. Similar to the `IRegion` interface mentioned above, the agent uses the proxy for retrieving a list of all registered agents. The retrieved list should be the same in both cases, since the same agency domain service is contacted.

Finally, the agent uses the `IRegionRegistration` proxy for registering a listener at the contacted region registry. For this purpose, the agent calls the method `addSystemListener(...)` on the proxy, providing the class name of the listener object, the code base from which the listener class can be retrieved, as well as listener-specific parameters. Detailed information about registering listeners at a region registry is given in Section 9.12.5.

**eventDetected(...)**

The method `eventDetected(...)` is defined in the agent's server interface IListeningAgent. By implementing this interface, the RegionClientAgent is accessible via the communication service. The created listener objects invoke this method (via an agent proxy) in order to inform the RegionClientAgent about events that occur inside the monitored region registry.

Example 24: RegionClientAgent

```
package examples.simple;

import examples.util.*;
import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.util.*;
import de.ikv.grasshopper.communication.*;

// This class realizes an agent that contacts a remote
// region registry.
public class RegionClientAgent extends StationaryAgent
                implements IListeningAgent
{
  GrasshopperAddress adsAddress;
  IRegion adsProxy;
  IRegionRegistration registryProxy;
  Identifier listenerId;

  // Creation argument:
  //   args[0] = address of a region registry
  public void init(Object[] args) {
    // The init method expects the address of a region
    // registry as argument.
    if (args == null || args.length < 1)
      adsAddress = null;
```

```
      else
        adsAddress = new GrasshopperAddress((
          String) args[0]);
      listenerId = null;
    }

    public String getName() {
      return "RegionClientAgent";
    }

    public void beforeRemove() {
      if (listenerId != null) {
        log("Removing listener...");
        try {
          registryProxy.removeSystemListener(
            listenerId);
        }
        catch (ListenerRemovalFailedException e) {
          log("Listener removal failed: ", e);
        }
      }
      log("Removing myself...");
    }

    public void live() {
      AgentInfo registeredAgents[];
      SearchFilter filter;

      if (adsAddress == null)
        // No domain service address has been specified
        // as creation argument.
        try {
          remove();
        }
        catch (Exception e) {
          log("Cannot remove myself. ", e);
        }
      // Get access to an agency domain service via the
      // IRegion interface of the local agency.
      adsProxy = getRegion();
      // Test the proxy by listing all agents that are
      // registered at the domain service.
      filter = new SearchFilter();
      log("Agent list retrieved via IRegion interface:");
      registeredAgents = adsProxy.listAgents(
        adsAddress, filter);
      for (int i = 0; i < registeredAgents.length; i++)
        log("    " + registeredAgents[i].
```

```
      getAgentPresentation().getAgentName() + ": " +
      registeredAgents[i].getIdentifier());

  // Get access to an agency domain service via a
  // proxy based on the IRegionRegistration
  // interface.
  registryProxy = (IRegionRegistration)
    ProxyGenerator.newInstance(
      IRegionRegistration.class,
      adsAddress.generateRegionId(),
      adsAddress);
  // Test the proxy by listing all agents that are
  // registered at the domain service.
  filter = new SearchFilter();
  log("Agent list retrieved via \\
    IRegionRegistration proxy:");
  registeredAgents =
    registryProxy.listAgents(filter);
  for (int i = 0; i < registeredAgents.length; i++)
    log("   " + registeredAgents[i].
      getAgentPresentation().getAgentName() +
      ": " + registeredAgents[i].getIdentifier());

  // Register a listener at the
  // IRegionRegistration proxy
  Object[] listenerArgs = new Object[3];
  try {
    // The following objects are constructor
    // arguments for the listener object
    listenerArgs[0] =
      (Identifier) getInfo().getIdentifier();
    listenerArgs[1] = (
      GrasshopperAddress) getInfo().getHome();
    listenerArgs[2] = (String) "Region";
    // Add listener
    listenerId = registryProxy.addSystemListener(
      "examples.util.GHListener",
      getInfo().getCodebase(), listenerArgs);
  }
  catch (ListenerCreationFailedException e) {
    log("Cannot listen to registry. Reason: ", e);
  }
  if (listenerId != null)
    log("Listener added to registry. Listening...");
}

// The following method is called by the listener
// object(s) due to events occurring inside the
```

```
    // monitored agencies.
    public void eventDetected(String event) {
      log(event);
    }
  }
```

**Interface IListeningAgent**

By implementing this interface, the RegionClientAgent is accessible via the Grasshopper communication service. The listener object invokes the method `eventDetected(...)` defined by this interface in order to inform the agent about events that occur inside the monitored region registry. Note that also the AgencyClientAgent described in Section 9.11.6 implements this interface. Please refer to this section for detailed information as well as a source code listing.

**Class GHListener**

By implementing the interface `ISystemListener`, the class `GHListener` realizes a listener that is able to monitor the events occurring inside Grasshopper agencies or region registries. Note that also the AgencyClientAgent described in Section 9.11.6 uses this class. Please refer to this section for detailed information as well as a source code listing.

Requirements:

- A running agency domain service. Note that the domain service has to be started before the agencies, and its address has to be specified when starting an agency. Please refer to the User's Guide for information about how to start agencies and agency domain services.
- At least one running agency

Running the Example:

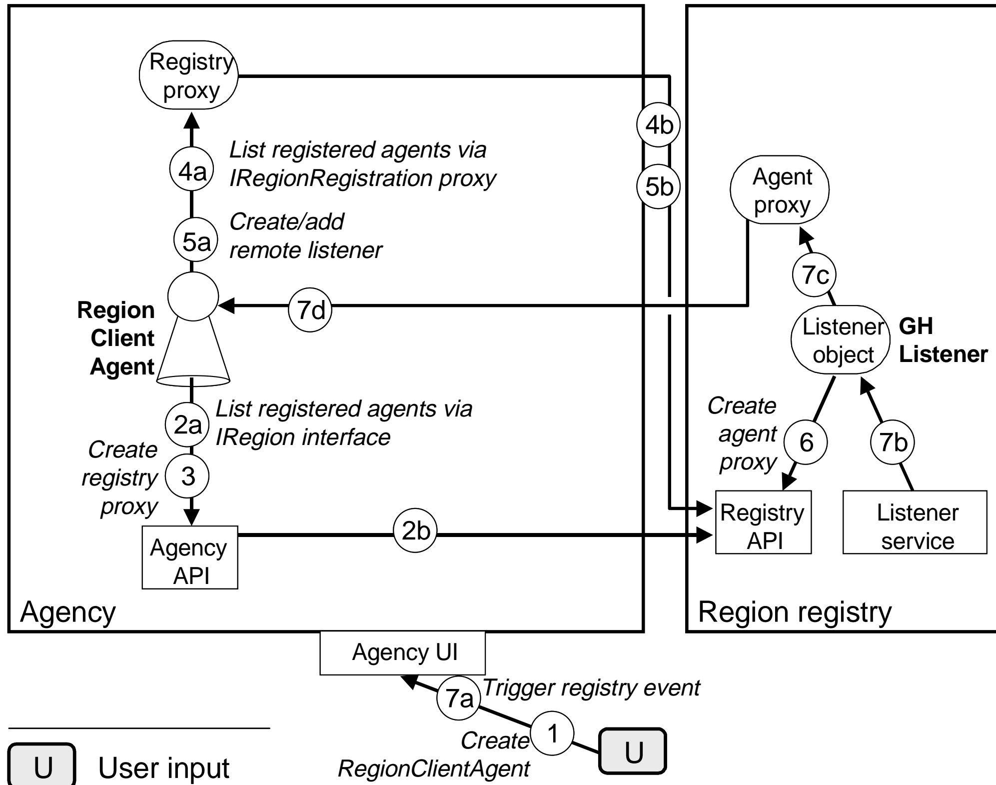



Figure 21: RegionClientAgent Scenario

Create some simple agents (e.g., the HelloAgent or PrintInfoAgent) in each running agency.

Create the RegionClientAgent in one of the running agencies (1).

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.simple.RegionClientAgent socket://
Host1:7020/Registry1
```

Note that you have to adapt the region registry address in the line above to the address of your concrete registry. You can determine this address via the command 'status' of the registry's text console.

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

The RegionClientAgent will print a list of all agents that are currently registered at the contacted agency domain service. Note that this list will be printed two times: The agent retrieves the first list via the `IRegion` inter-

face of the local agency (2) and the second one via a `IRegionRegistration` proxy (4). Both lists should comprise the same set of agents.

After retrieving and printing the agent lists, the RegionClientAgent adds a listener to the registry (5). The listener in turn creates a proxy of the RegionClientAgent for the purpose of notification forwarding (6).

The last output of the agent should be the following:

```
Listener added to registry. Listening...
```

In order to prove that the agent is now listening to the registry, please create a new agency and provide the registry address as creation argument, so that the new agency registers itself at the monitored registry. The RegionClientAgent should notice the creation and confirm it by means of a corresponding textual output. Now create, suspend, resume, and remove agents and places in all running agencies (7) and look at the textual output of the RegionClientAgent.

### 9.12.7 Summary

- An agent can use the interface `de.ikv.grasshopper.agency.IRegion` in order to access an agency domain service. The interface `IRegion` is provided by each agency, and it is accessible for locally residing agents via the agents' superclass `de.ikv.grasshopper.agency.Agent`. The interface `IRegion` offers a unified access to agency domain services, independent of whether they are represented by a Grasshopper region registry or by an LDAP server.
- An agent can use the interface `de.ikv.grasshopper.agency.IRegionRegistration` in order to access a Grasshopper region registry. In contrast to the interface `IRegion`, the interface `IRegionRegistration` is not provided by Grasshopper agencies. Instead, an agent has to use this interface in order to create a region registry proxy.
- The interface `IRegionRegistration` enables an agent to register a listener object at a region registry in order to be notified about registry-internal events.

## 9.13 Searching Grasshopper Components

The interfaces `IAgentSystem` (see Section 9.11.2) and `IRegion` (see Sec-

tion 9.12.1) of a Grasshopper agency provide several methods for searching agencies, places, and agents. These methods are of particular interest for agents acting as communication clients in order to find suitable server agents. Usually, a client needs a server with specific capabilities. When using the available list methods, a client can define a search filter by specifying the demanded server characteristics. In this case, the list methods only return information about those servers that match to the defined filter criteria.

Search filters are represented as instances of the class `de.ikv.grasshopper.util.SearchFilter`. The filter criteria are specified in terms of a `String` object whose content must match to the following syntax:

**Filter syntax**

```
filter      = not-filter
not-filter  = ('!')? or-filter*
or-filter   = and-filter ( '|' and-filter )*
and-filter  = item ( '&' item )*
item        =   key comperator value
              | '(' not-filter ')'
comperator  =   '='         # equals
              | '^'         # starts with
              | '$'         # ends with
              | '~'         # contains
key         = java.lang.String
value       = java.lang.String
```

**Syntax explanation**

The syntax considers the following rules (where the character sequence `symbol` represents any symbol in the above syntax, such as `filter` or `item`):

- `symbol`

  Non-terminal symbol. A non-terminal symbol at the left side of the '=' character substitutes the rule that is defined at the right side of the same line.

  Example:
  The rule `filter = not-filter`
  is equal to `filter = (!)? or-filter*`.

- `'symbol'`

  Terminal symbol. A terminal symbol cannot be substituted anymore. It is represented by the concrete character sequence surrounded by `''`.

- `symbol1 symbol2`

  `symbol2` *follows* `symbol1`

- `symbol1 | symbol2`

  `symbol1` *or* `symbol2`

- `symbol?`

  `symbol` is *optional*. If it occurs, than *only once*.

- `symbol*`

  `symbol` is *optional*. If it occurs, than with *any number of repetitions*

- `(symbol1 symbol2)`

  *Group* of symbols. Any operator typed behind the closing bracket is applied to the whole group.

**Minimal filter**

The minimal filter consists of a single `item`, represented by a `key` and a `value` which are connected by means of a `comparator`. The corresponding rule is

```
item = key comperator value
```

**Filter keys**

The supported keys are defined in the class `de.ikv.grasshopper.util.SearchFilter`. Note that some keys cannot be applied to all kinds of Grasshopper components (i.e., agencies, places, and agents). For instance, the `CODEBASE` key can only be applied to agents, and the LASTLOCATION key is only valid for mobile agents. If, for example, a list method is invoked with a filter that contains the LASTLOCATION key, all Grasshopper components that cannot be applied to this key are automatically excluded from the search. Thus, a method call that uses the LASTLOCATION key within its filter can only return a list of mobile agents.

The following table lists all available keys and the components to which they can be applied.

Table 3: Filter Keys

| Filter key | Description | Can be applied to |
|---|---|---|
| `CODEBASE` | Code base of the searched component's Java classes. The corresponding value must match to the code base syntax defined in Section 5.3.<br><br>A value of this key corresponds to the return value of the method `getCodebase()` of the class `AgentInfo`. | • Agents |

Table 3: Filter Keys

| **Filter key** | **Description** | **Can be applied to** |
|---|---|---|
| `DESCRIPTION` | Textual description of the searched component.<br>A value of this key corresponds to the return value of the method `getAgentDescription()` of the class `AgentPresentation`, respectively to the return value of the method `getDescription()` of the class `PlaceInfo`. | • Agents<br>• Places |
| `HOME` | Home location of the searched component.<br>A value of this key corresponds to the return value of the method `getHome()` of the class `AgentInfo`.<br>The syntax of the corresponding value must match to a textual representation of the class `de.ikv.grasshopper.communication.GrasshopperAddress`,<br>as defined in Section 5.4. | • Agents<br>(Equal to LOCATION for stationary agents) |
| `INTERFACENAME` | Full qualified class name of the searched component.<br>A value of this key corresponds to the return value of the method `getAgentInterfaceName()` of the class `AgentInfo`. | • Agents |

Table 3: Filter Keys

| Filter key | Description | Can be applied to |
|---|---|---|
| `DESCRIPTION` | Textual description of the searched component.<br>A value of this key corresponds to the return value of the method `getAgentDescription()` of the class `AgentPresentation`, respectively to the return value of the method `getDescription()` of the class `PlaceInfo`. | • Agents<br>• Places |
| `HOME` | Home location of the searched component.<br>A value of this key corresponds to the return value of the method `getHome()` of the class `AgentInfo`.<br>The syntax of the corresponding value must match to a textual representation of the class `de.ikv.grasshopper.communication.GrasshopperAddress`,<br>as defined in Section 5.4. | • Agents<br>(Equal to LOCATION for stationary agents) |
| `INTERFACENAME` | Full qualified class name of the searched component.<br>A value of this key corresponds to the return value of the method `getAgentInterfaceName()` of the class `AgentInfo`. | • Agents |

Table 3: Filter Keys

| Filter key | Description | Can be applied to |
|---|---|---|
| `LASTLOCATION` | Previous location of the searched component.<br>A value of this key corresponds to the return value of the method `getLastLocation()` of the class `AgentInfo`.<br>The syntax of the corresponding value must match to a textual representation of `de.ikv.grasshopper.communication.GrasshopperAddress`,<br>as defined in Section 5.4. | • Mobile agents |
| `LOCATION` | Current location of the searched component.<br>A value of this key corresponds to the return value of the method `getLocation()` of the `AgentInfo` of the class `AgentSystemInfo`.<br>The syntax of the corresponding value must match to a textual representation of the class `de.ikv.grasshopper.communication.GrasshopperAddress`,<br>as defined in Section 5.4. | • Agents<br>• Agencies<br>• Places |

Table 3: Filter Keys

| Filter key | Description | Can be applied to |
|---|---|---|
| `NAME` | Name of the searched component.<br><br>A value of this key corresponds to the return value of the method `getName()` of the class `PlaceInfo` or `AgentSystemInfo`, respectively to the return value of the method `getAgentName()` of the class `AgentPresentation`. | • Agents<br>• Agencies<br>• Places |
| `SERVICEID` | Identifier of the searched component.<br><br>A value of this key corresponds to the return value of the method `getIdentifier()` of the class `AgentInfo` or `AgentSystemInfo`.<br><br>The syntax of the corresponding value must match to a textual representation of<br>`de.ikv.grasshopper.type.Identifier`,<br>as defined in Section 5.1. | • Agents<br>• Agencies |
| `STATE` | Current state of the searched component.<br><br>A value of this key corresponds to the return value of the method `getState()` of the class `AgentInfo` or `PlaceInfo`. | • Agents<br>• Places |
| `TYPE` | Type of the searched component.<br><br>A value of this key corresponds to the return value of the method `getType()` of the class `AgentInfo` or `AgentSystemInfo`. | • Agents<br>• Agencies |

**Creating a Filter**

To create an empty filter which matches all entries, just create an empty `SearchFilter` object.

```
SearchFilter filter = new SearchFilter();
```

To define a specific filter, create a `String` object and initialize it with a filter rule that matches the above syntax. Create a `SearchFilter` object and provide the previously defined filter rule as parameter of the `SearchFilter` constructor. Another possibility is to create an empty `SearchFilter` object and use its `setFilter(...)` method in order to specify the filter rule.

## 9.14 Migrating Servers and Clients

The fundamental characteristic of mobile agents is their ability to autonomously migrate from one network location to another. This ability may cause problems if an agent wants to migrate while being involved in a communication session. The Grasshopper communication service is able to handle the following scenarios of migrating agents:

**First Migration Scenario: Migrating Server Agent**

A client agent creates a proxy of a server agent and starts a communication session by periodically invoking methods on the proxy. The communication service forwards the method invocations to the remotely running server agent. After a few invocations performed by the client agent, the server agent migrates to a new location. After the server agent's migration, the client agent again tries to invoke a server method.

Concerning this scenario, two aspects have to be considered:

1. *The server agent migrates away before the invoked method has returned a result to the client agent.*

   In this case, the method will be completed, even in the absence of the server agent. The reason is that the method is performed in a separate thread, created by the communication service. The server agent's own thread runs independently in parallel. If the server agent moves away, its own thread terminates, but its (passive) object instance remains valid as long as it is referenced by other objects. After completing the method, the communication service returns the result to the client agent, terminates the thread that has performed the method, and releases its references to the server agent's object instance. After this, the Java garbage collector is able to remove the server object.

2. *The client agent again tries to invoke a server method after the server*

*agent has migrated to a new location.*

In this case, both involved agencies (i.e., the agency hosting the client agent as well as the agency hosting the server agent) must be registered at the same agency domain service. If this condition is fulfilled, the proxy automatically contacts the agency domain service in order to retrieve the new location of the server agent. After this, the proxy forwards the method invocation to the new location. The client agent does not need to be aware of the fact that the server agent has moved.

**Second Migration Scenario: Migrating Client Agent**

A client agent creates a proxy of a server agent and starts a communication session by periodically invoking methods on the proxy. The communication service forwards the method invocations to the remotely running server agent. After a few invocations, the client agent migrates to a new location. Once arrived at the new location, the client agent again tries to invoke a method on the server agent.

The client agent's proxy remains valid after the migration. Thus, the client can continue invoking the server methods via the same proxy.

If the client agent invokes the server method asynchronously, the agent is not blocked until the server method returns. Thus, the client agent may move to another location right after the invocation, i.e., without waiting for a result. If the invoked server method does not return any result or the client agent does not need it, nothing special has to be considered. If the client wants to retrieve the server result at its new location, the asynchronous method invocation must be performed with the listener approach. Concerning the client agent's implementation, the following conditions must be fulfilled:

- The client agent itself must implement the interface `de.ikv.grasshopper.communication.ResultListener`. Thus, the client agent must provide the method `resultHasArrived(...)`.
- After invoking the server method and retrieving the `FutureResult` object, the client agent must add itself as listener to this object by invoking the method `addResultListener(...)`.

When the server method returns, the method `resultHasArrived(...)` is automatically called, in this way notifying the client agent and allowing it to handle the result. If the conditions above are fulfilled, the notification is automatically forwarded to the client agent if it has changed its location after invoking the server method. For detailed information about the listener approach and the asynchronous communication capabilities of Grasshopper in general, please refer to Section 9.6.

The following example incorporates all migration scenarios described above.

## 9.15 Migration Scenario

The example scenario for showing the migration capabilities of communicating agents consists of three classes/interfaces, covered by the package `examples.migratingCom`:

- `MigratingServerAgent` (see Example 25 in Section 9.15.1): An agent that provides one method to the communication service.
- `IMigratingServerAgent` (see Example 26 in Section 9.15.1): The server interface that contains the method which has to be accessible for the client agent. This interface is the basis for the generation of server proxies.
- `MigratingClientAgent` (see in Example 27 Section 9.15.2): The client agent that invokes the accessible method of the server.

### 9.15.1 Example: MigratingServerAgent

The purpose of the MigratingServerAgent is to perform a method that is invoked by the MigratingClientAgent via the Grasshopper communication service. This method suspends the thread in which it is running, just in order to produce a delay before returning a result. The purpose of this delay is to enable the MigratingClientAgent to move to another location before the server method has returned a result.

**init(...)**

Inside its `init(...)` method, the MigratingServerAgent expects a creation parameter, defining the delay for the server method in milliseconds. If no delay is defined by the user, the default value of 4000 milliseconds is used.

**live()**

The `live()` method creates a graphical user interface, requesting a new location from the user. After the user has pressed the OK button of the GUI, the agent tries to migrate to the specified location where the `live()` method is started again.

**serverMethod(...)**

The method named `serverMethod(...)` is meant to be called by the MigratingClientAgent. This method produces a delay in order to enable the MigratingClientAgent to migrate to another location before returning a result.

Example 25: MigratingServerAgent

```
package examples.migratingCom;
```

```
import de.ikv.grasshopper.agent.MobileAgent;
import de.ikv.grasshopper.communication.*;
import javax.swing.*;
import java.awt.*;

// This class realizes the server agent of the migrating
// communication scenario.
public class MigratingServerAgent extends MobileAgent
                implements IMigratingServerAgent
{
  // Data state of the agent, since not transient
  int delay;

  // Required creation argument:
  //   args[0] = delay time between invocation and
  //     termination of the method 'serverMethod'.
  public void init(Object[] creationArgs) {

    if (creationArgs == null ||
        creationArgs.length < 1){
      log("Creation argument needed: <delayTime>");
      log("Exiting.");
      throw new RuntimeException();
    }
    if (creationArgs != null)
      delay =
        Integer.parseInt((String)creationArgs[0]);
    else
      delay = 4000;
  }

  public String getName() {
    return "MigratingServerAgent";
  }

  public void live() {
    String location;

    log("Waiting for new location...");
    // Wait for user input
    location = JOptionPane.showInputDialog(
      null, "Where shall I go?");
    if (location != null) {
      log("Trying to move...");
      try {
        // Go away!
        move(new GrasshopperAddress(location));
      }
```

```
        catch (Exception e) {
          log("Migration failed: ", e);
        }
      }
    }

    // The following method is accessible via the
    // communication service.
    public String serverMethod(int value) {
      log("Performing client request with value = " +
        value);
      try {
        Thread.currentThread().sleep(delay);
      }
      catch (InterruptedException e) {
        log("Sleep interrupted.");
      }
      log("Returning result to client.\n");
      return new String("Server result = " +
        Integer.toString(value));
    }
  }
```

Example 26: IMigratingServerAgent

```
  package examples.migratingCom;

  public interface IMigratingServerAgent
  {
    public String serverMethod(int value);
  }
```

### 9.15.2 Example: MigratingClientAgent

The MigratingClientAgent periodically invokes the method named `server-Method(...)` of the MigratingServerAgent via the communication service. By invoking the client agent's action() method, a user can order the agent to move to another location.

The MigratingClientAgent maintains the following instance variables which are, since not declared transient, part of the agent's data state:

- `syncServerProxy`: This variable maintains a proxy of the Migrating-ServerAgent which is able to handle synchronous method calls. Its initialization is performed inside the `init(...)` method of the

MigratingClientAgent.

- `asyncServerProxy`: This variable maintains a proxy of the Migrating-ServerAgent which is able to handle asynchronous method calls. Its initialization is performed inside the `init(...)` method of the MigratingClientAgent.
- `futureResult`: This variable maintains a `FutureResult` object which is responsible for retrieving an asynchronously arriving result from the MigratingServerAgent.
- `serverParameter`: The integer value of this variable is used as parameter of the method `serverMethod(...)` of the MigratingServerAgent. The value is incremented after each method call on the MigratingServer-Agent in order to enable the user to find corresponding outputs performed by the client and the server in the text console windows of both involved agencies.
- `requestedLocation`: This variable is initialized inside the `action()` method of the MigratingClientAgent. By means of this method, which can be called by the user via the agency's UI, the agent requests a new location. Inside the `live()` method, the agent migrates to this location as soon as it has been specified by the user.
- `serverId`: This variable maintains the identifier of the contacted MigratingServerAgent.
- `comMode`: The value of this variable has to be specified by the user as first creation argument for the MigratingClientAgent. Allowed values are „sync“ and „async“. Depending on the specified value, the client agent performs the server method synchronously or asynchronously.

**init(...)**

Inside its `init(...)` method, the MigratingClientAgent stores the provided creation arguments in its non-transient instance variables. After this, the agent contacts an agency domain service via the `IRegion` interface of the local agency in order to look for a running MigratingServerAgent. The client agent selects the first server agent from the retrieved list and creates two server proxies: one for synchronous and one for asynchronous communication. Finally, the instance variable `serverParameter` is set to 0.

**action()**

The `action()` method of the client agent enables a user to send the agent to a new location. Inside the method (which can be invoked by the user via the agency's UI, as explained in Chapter 7), an input dialog window is created, requesting a new destination location from the user. The location is maintained by the instance variable `requestedLocation`. The corresponding `move(...)` operation is called inside the agent's `live()` method.

**syncLive()**

The method `syncLive()` is called from inside the agent's `live()` method, if the synchronous communication mode has been selected by the user. Inside this method, the agent periodically invokes the method `serverMethod(...)` of the MigratingServerAgent via the synchronous server proxy. The method calls are performed inside a `while` loop which ends either due to a caught communication exception or due to a retrieved migration request performed by the user.

An `ObjectNotBoundException` may occur due to the call of the server method if the server agent is not available for communication. Two possible reasons may lead to this exception:

- The MigratingServerAgent is not alive anymore.
- The MigratingServerAgent is currently migrating to another location.

In the second case, the MigratingServerAgent will probably become available again after a short time. Thus, the client agent retries the method invocation five times, waiting for one second between two communication attempts.

If any other exception is thrown, the client agent assumes that the server agent will not become available again and thus terminates its `while` loop at once.

If the reason for terminating the `while` loop was a migration request performed by a user, the client agent tries to move to the new destination location.

**asyncLive()**

The method `asyncLive()` is called from inside the agent's `live()` method, if the asynchronous communication mode has been selected by the user. Inside this method, the agent invokes the method `serverMethod(...)` of the MigratingServerAgent via the asynchronous server proxy.

After invoking the server method, the client agent retrieves a reference of the `FutureResult` object from the asynchronous server proxy and uses this object for registering itself as result listener. After this, the client agent's method `resultHasArrived(...)` is automatically called due to an incoming server result. Please refer to Section 9.5 for detailed information about asynchronous communication in Grasshopper.

**live()**

Inside its `live()` method, the agent checks the availability of the MigratingServerAgent. If one of the server proxies refers to null, the client agent assumes that no server agent has been found and removes itself. Otherwise, the client agent invokes either the method `syncLive()` or `asyncLive()`, depending on the communication mode that has been selected by the user.

**resultHasArrived(...)**

If the user has selected the asynchronous communication mode, the method `resultHasArrived(...)` is automatically invoked after retrieving an asynchronously incoming server result. Please refer to Section 9.5 for detailed information about asynchronous communication in Grasshopper.

Example 27: MigratingClientAgent

```
package examples.migratingCom;

import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.communication.*;
import de.ikv.grasshopper.util.*;
import javax.swing.*;
import java.awt.*;

// This class realizes the client agent of the migrating
// communication scenario.
public class MigratingClientAgent extends MobileAgent
                implements ResultListener
{
  // Data state of the agent, since not transient
  IMigratingServerAgent syncServerProxy;
  IMigratingServerAgent asyncServerProxy;
  FutureResult futureResult;
  int serverParameter;
  String requestedLocation;
  String serverId;
  String comMode;

  // Required creation arguments:
  //   args[0] = communication mode. Expected values =
  //     "sync" or "async"
  public void init(Object[] creationArgs) {
    IRegion regionProxy;
    AgentInfo[] serverInfos;

    if (creationArgs == null ||
        creationArgs.length < 1) {
      log("Creation argument needed: <comMode>");
      log("Exiting.");
      throw new RuntimeException();
    }
    // Get creation argument
    // comMode: Expected values = "sync" or "async"
    comMode = (String) creationArgs[0];

    // Get domain service proxy of local agency
    regionProxy = getRegion();
    // Look for the server agent in the
    // agency domain service
```

```
  SearchFilter filter =
    new SearchFilter(
      SearchFilter.NAME+"=MigratingServerAgent");
  serverInfos = regionProxy.listAgents(
    null, filter);
  // Create proxies of the server agent
  // (One for sync. and one for async. communication)
  syncServerProxy = null;
  asyncServerProxy = null;
  serverId = null;
  if (serverInfos != null) {
    serverId =
      serverInfos[0].getIdentifier().toString();
    syncServerProxy = (IMigratingServerAgent)
      ProxyGenerator.newInstance(
        IMigratingServerAgent.class,
        serverId, ProxyGenerator.SYNC);
    asyncServerProxy = (IMigratingServerAgent)
      ProxyGenerator.newInstance(
        IMigratingServerAgent.class,
        serverId, ProxyGenerator.ASYNC);
  }
  // Initialize parameter for server method.
  serverParameter = 0;
}

public String getName() {
  return "MigratingClientAgent";
}

// By invoking this method via the agency's user
// interface, the user can move the agent to another
// location.
public void action() {
  if (comMode.equals("sync")) {
    log("Waiting for new location...");
    requestedLocation =
      JOptionPane.showInputDialog(
        null, "Where shall I go?");
  }
}

public void beforeMove() {
  log("Moving.");
}

public void afterMove() {
  log("Arrived.");
```

```
}

// This method is performed if the agent has been
// started with the communication mode set to "sync".
public void syncLive() {
  String serverResult;
  int numberOfRetries;

  numberOfRetries = 0;
  requestedLocation = null;
  while ( numberOfRetries < 5 &&
          requestedLocation == null) {
    try {
      log("Calling server method with value = " +
        serverParameter);
      // Invoke server method synchronously
      // by using the sync. server proxy
      serverResult =
        syncServerProxy.serverMethod(
          serverParameter);

      serverParameter++;
      log("Result has arrived: " + serverResult +
        "\n");
    }
    catch (ObjectNotBoundException e) {
      // Server agent not found. Possible reason:
      // server agent is currently migrating
      // or server agent has been removed.
      // Retry 5 times.
      log("Server agent currently not available.\\
      Retrying " +
      Integer.toString(5 - numberOfRetries) +
      " times...");
      numberOfRetries++;
      serverParameter--;
      // Wait for a second until retrying contact.
      try {
        Thread.currentThread().sleep(1000);
      }
      catch (InterruptedException e2) {
        log("Sleep interrupted.");
      }
    }
    catch (Throwable t) {
      // Something unexpected happened. Terminate
      // while loop.
      log("Communication exception caught: ", t);
```

```
        numberOfRetries = 5;
      }
    }
    if (requestedLocation != null) {
      log("Trying to move...");
      try {
        // Go away!
        move(
          new GrasshopperAddress(requestedLocation));
      }
      catch (Exception e) {
        requestedLocation = null;
        log("Migration failed: ", e);
      }
    }
  }

  // This method is performed if the agent has been
  // started with the communication mode set to
  // "async".
  public void asyncLive() {
    int numberOfRetries;
    String requestedLocation;

    log("Waiting for new location...");
    requestedLocation =
      JOptionPane.showInputDialog(
        null, "Where shall I go?");

    // Invoke server method asynchronously
    // by using the async. server proxy
    try {
      log("Starting asynchronous call");
      // Invoke server method asynchronously
      // by using the async. server proxy
      asyncServerProxy.serverMethod(serverParameter);
      // Get futureResult object from the proxy
      futureResult = ((IFutureResult)
        asyncServerProxy).getFutureResult();
      // The client agent adds itself as result
      // listener to the futureResult object
      futureResult.addResultListener(this);
      // Note: The result will be retrieved by the
      // method resultHasArrived
      // of the client's result listener.
      log("Listening for notification");
      serverParameter++;
    }
```

```
      catch (Throwable t) {
        log("Cannot contact server: ", t);
      }

      log("Trying to move...");
      try {
        // Go away!
        move(new GrasshopperAddress(requestedLocation));
      }
      catch (Exception e) {
        log("Migration failed: ", e);
      }
    }

    public void live() {
      if (syncServerProxy == null ||
          asyncServerProxy == null) {
        log("No MigratingServerAgent found. Removing\\
        myself...");
        try {
          remove();
        }
        catch (Exception e) {
          log("Removal failed.");
        }
      }

      if (comMode.equals("sync"))
        syncLive();
      else
        asyncLive();
    }

    // The following method is only needed if the client
    // agent has been started in asynchronous mode.
    // The method is automatically called when an
    // asynchronous server result has arrived.
    public void resultHasArrived(ResultEvent e){
      FutureResult fResult;
      String serverResult = null;
      log("Listener notified.");
      fResult = (FutureResult) e.getSource();
      try {
        serverResult = (String) fResult.getResult();
      }
      catch (Throwable t) {
        log("Exception caught: ", t);
      }
```

```
      if (serverResult != null)
        log("Notified server result = " + serverResult);
    }
  }
```

### 9.15.3 Running the Scenario

Requirements:

- A running agency domain service. Note that this service has to be started before the agencies, and the service's address has to be specified when starting the agencies in order to register them. Please refer to the User's Guide for more information about how to start agencies and agency domain services.
- Three running agencies
  *Since the agents in this scenario create own GUIs that may block the agency GUI, it is recommended that you do not activate the agency GUI. Instead, start the agencies just with their textual interface (command option -tui). Please refer to the paragraphs titled „Running the Examples" at the beginning of Chapter 2 in order to get a detailed explanation about the possibly occurring GUI problems.*
- If you are using a JDK 1.2 environment, you must have generated a proxy class (named `IMigratingServerAgentP`) by invoking the Grasshopper stub generator with the interface class `IMigratingServerAgent` as input parameter. The file `IMigratingServerAgentP.class` should be stored either in a directory belonging to the Java classpath or in the code base directory of the MigratingClientAgent. In a JDK 1.3 environment, this class is not needed. Even if it is available, it will not be used. Instead, the proxy is dynamically generated by the MigratingClientAgent at runtime.

Running the Example:

Create a MigratingServerAgent in one of the running agencies (1). This agency will be referred to as Agency_1 in the scope of this section. As creation argument, you can optionally provide a delay time in milliseconds. This delay time will be used by the agent to delay the method that is remotely called by the MigratingClientAgent. If you do not specify a delay time, the default value of 4000 milliseconds will be used.

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.migratingCom.MigratingServerAgent 5000
```

Note that you may specify any other (positive) delay time that fits into the range of an `int` variable. The only purpose of this delay time is to enable the user to send the client agent to another location before the server method returns.

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

Create a MigratingClientAgent in one of the running agencies (2). Do not use the agency in which the MigratingServerAgent has been created before. The hosting agency of the MigratingClientAgent agency will be referred to as Agency_2 in the scope of this section. The MigratingClientAgent expects one creation argument, specifying the desired communication mode. The mode may be set to „sync“ for synchronous communication or „async“ for asynchronous communication.

If you are using the textual user interface of the agency, please create the agent by using one of the following commands, depending on the desired communication mode:

- `cr a examples.migratingCom.MigratingClientAgent sync`
- `cr a examples.migratingCom.MigratingClientAgent async`

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

After creating both agents in the order mentioned above, the behavior of the scenario depends on the communication mode that has been specified as creation argument of the MigratingClientAgent:

**Synchronous scenario**

Synchronous scenario, performed if the MigratingClientAgent has been creat-

ed with the value „sync“ as demanded communication mode:

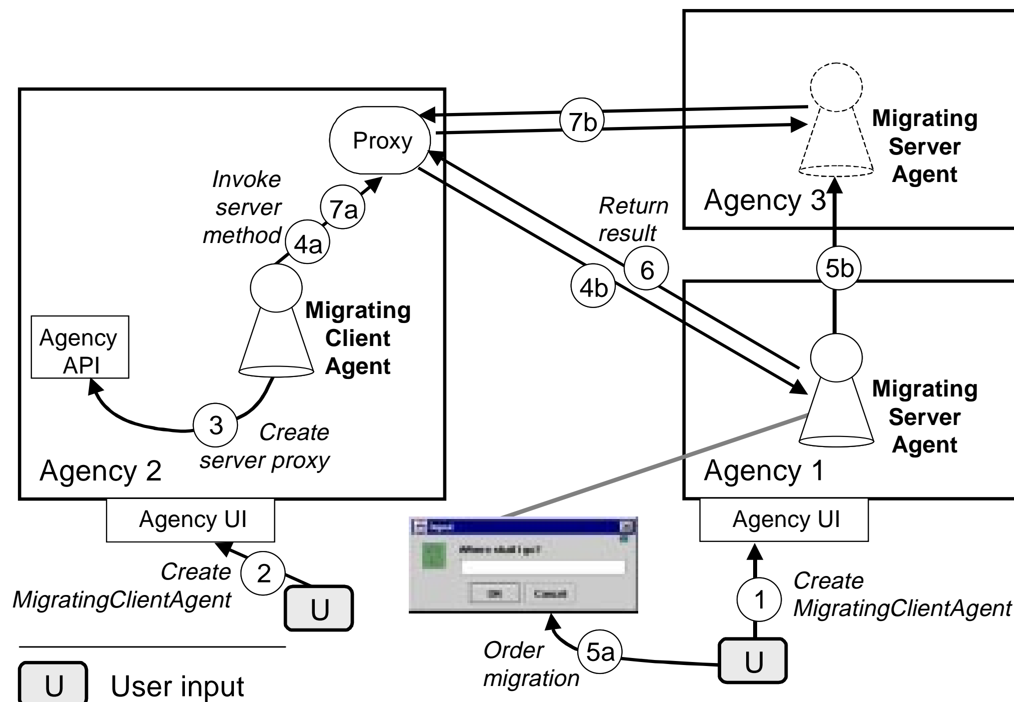



Figure 22: Synchronous Migration Scenario

The MigratingServerAgent creates a graphical input dialog window, requesting a new destination location from the user.

The MigratingClientAgent creates a proxy of the MigratingServerAgent (3) and periodically invokes the method `serverMethod(...)` via the proxy (4). Before returning, the server method produces the previously specified delay or the default delay of 4000 milliseconds, as described above.

Please type the address of the third agency into the server agent's GUI, i.e., the address of the agency in which no agent is currently running, and press the OK button (5a). This agency is referred to as Agency_3 in the scope of this section. You will see that the server agent is migrating at once (5b), and you will also notice that the currently running method `server-Method(...)`, invoked by the MigratingClientAgent before the migration request, is completed in spite of the absence of the server agent (6). Finally you will notice that all subsequent method invocations, performed by the client agent (7a), are automatically forwarded to the new location of the MigratingServerAgent (7b).

Please invoke the `action()` method of the MigratingClientAgent. If you are using the textual user interface of the agency, you can do this via the

`invoke` command. (Please refer to the User's Guide for detailed information about this command.) After performing this command, the client agent creates a graphical user interface, requesting a new destination location. Please type in the address of Agency_2 and press the OK button. You will see that the client agent waits for the completion of the previously invoked server method and, after this, migrates to Agency_2. Once arrived at its new destination, the client agent continues invoking the server method.

The client agent as well as the server agent perform outputs on the text consoles of their hosting agency. Among others, these outputs comprise the value of the integer variable `serverParameter` which is incremented after each method call. By looking at this number, you can see that the result of every invoked method is retrieved by the client agent, independent of the migration behavior of both agents.

Please proceed with the scenario by moving both agents back and forth and looking at the output occurring in the agencies' text consoles.

**Asynchronous scenario**

Asynchronous scenario, performed if the MigratingClientAgent has been created with the value „async" as demanded communication mode:

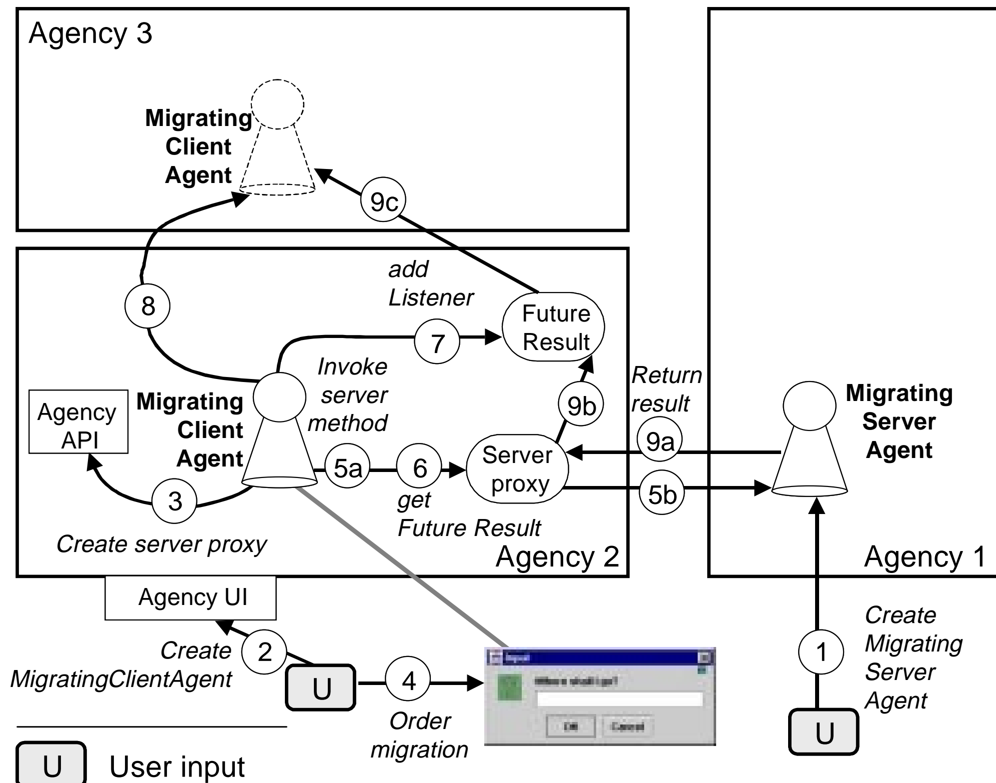



Figure 23: Asynchronous Migration Scenario

The MigratingServerAgent creates a graphical input dialog window, requesting a new destination location from the user.

After creating a server proxy (3), the MigratingClientAgent also creates a graphical input dialog window, requesting a new destination location from the user.

Please type the address of Agency_3 into the input dialog window of the client agent and press the OK button (4).

The client agent invokes the server method asynchronously (5), gets a FutureResult object from the server proxy (6), and registers itself as result listener (7). After this, the client agent and migrates to Agency_3 without waiting for the result of the invoked method (8). By looking at the client agent's output in the text console of Agency_3, you will notice that the result of the server method is automatically forwarded to the client agent's new location (9).

Please proceed with the scenario by moving both agents back and forth and looking at the output occurring in the agencies' text consoles.

### 9.15.4 Summary

- The Grasshopper communication service is able to keep track of migrating server and client agents.
- If a client agent invokes a method on a server agent and the server agent migrates to another location before completing the method, the method is completed by the agency even without the presence of the server agent. All subsequent invocations, performed by the client agent, are automatically forwarded to the server agent's new location. Note that this forwarding mechanism only works if the same agency domain service is available at all involved agencies.
- If a client agent uses asynchronous, notification-based communication for invoking methods of a server agent, and if the client agent itself implements the required result listener interface, then the client agent may migrate *before* retrieving the result of a previously invoked server method. The communication service will automatically forward the result to the client agent's new location.

# 9.16 Interacting with External Applications

The sections above have described how to use the different communication mechanisms of Grasshopper for interactions between agents, agencies, and agency domain services. However, beside these „Grasshopper-internal“ interactions, the communication service also allows external applications to interact with the Grasshopper platform. The external application may act as communication client and/or server.

### External Client Applications

In order to use the Grasshopper communication service as a client, an external application has to behave exactly as a Grasshopper agent: the application just creates a proxy of the desired server object/agent by using the `newInstance(...)` method of the class `de.ikv.grasshopper.communication.ProxyGenerator`.

Note: An external application always has to specify the complete server address, when creating a server proxy. The simplification to use a combination of host name and agency name instead (which is possible for client agents if the local agency is registered at an agency domain service) cannot be applied. In order to determine the address of a server agent or agency, the external application can contact an appropriate region registry by creating a registry proxy and specifying the complete registry address (see the example below).

### External Server Applications

Similar to agents acting as servers, an external application must implement a server interface, i.e., an interface that defines those methods that have to be made accessible via the communication service. Please refer to Section 9.1 for detailed information. Concerning the proxy generation, please refer to Section 9.2.

**External-CommService**

In order to enable an external application to use the Grasshopper communication service as a server, Grasshopper provides the class `de.ikv.grasshopper.communication.ExternalCommService`. This class provides the following methods:

- `startReceiver(...)`: This method starts a new communication receiver at the `ExternalCommunicationService` object. A communication receiver is characterized by a Grasshopper address, and it is needed by communication clients for accessing server objects. The complete address of the communication receiver has to be provided as method parameter. Note that one instance of the class `ExternalCommunicationService` can maintain several communication receivers, for exam-

ple in order to support different protocols.

- `registerObject(...)`: This method registers a server object at the `ExternalCommunicationService` instance. Similar to Grasshopper server agents, a server object is characterized by implementing a server interface, i.e., a Java interface that defines those methods of the server object which are to be accessible for clients via the communication service.
- `deregisterObject(...)`: This method deregisters a server object from the `ExternalCommunicationService` object.
- `shutdown()`: This method terminates the `ExternalCommunicationService` object.

The external application has to perform the following steps in order to become accessible as server:

1. Create an instance of the class `de.ikv.grasshopper.communication.ExternalCommService`.
2. Create an instance of the class `de.ikv.grasshopper.communication.GrasshopperAddress`. As constructor argument, provide a complete Grasshopper address. This address will be used later on by clients to connect themselves to the server object.
3. Start a communication receiver on the `ExternalCommService` object that has been created in step 1. As method parameter, provide the Grasshopper address object that has been created in step 2. (Note that, similar to Grasshopper agencies and region registries, multiple communication receivers can be started, e.g., in order to support multiple protocols.)
4. Register the server object (i.e., the object that implements the server interface of the external application) at the `ExternalCommService` object. The server object may be the application itself or a separate Java object. As method parameters, an identifier for the server object as well as a reference of the server object have to be specified. The identifier will be used later on for identifying the server object when a client wants to create a server proxy. (Note that in the case of Grasshopper agents, the automatically generated agent identifier is used for this purpose. In the case of an external application, the identifier is user-defined, and thus its uniqueness is not guaranteed.)

# 9.17 External Communication Scenario

The following examples shows how an external application can use the Grasshopper communication service. The external application acts in the client and in the server role.

The example scenario consists of the following classes/interfaces, covered by the package `examples.externalCom`:

- `ExternalApplication` (see Example 28 in Section 9.17.1): A stand-alone Java application that interacts with the Grasshopper environment as communication client and server.
- `ServerObject` (see Example 29 in Section 9.17.1): The actual server object class. An instance of this class is created by the ExternalApplication, and it is contacted by the ExternalAccessAgent that acts as a client to this object.
- `IServerObject` (see Example 30 in Section 9.17.1): The server interface of the external application, implemented by the class ServerObject.
- `ExternalAccessAgent` (see Example 31 in Section 9.17.2): A Grasshopper agent that is created by the ExternalApplication and that acts as a client on the ServerObject.
- `IExternalAccessAgent` (see Example 32 in Section 9.17.2): The server interface of the ExternalAccessAgent.

## 9.17.1 Example: ExternalApplication

**Class ExternalApplication**

The class ExternalApplication acts as client as well as server of the Grasshopper communication service. It realizes a stand-alone Java application that requires the following creation arguments:

**Creation arguments**

- The complete address of a running region registry
- An identifier for the server object. This name will be used later on by the ExternalAccessAgent to create a proxy of the server object.
- an address for the communication receiver that will be created by the ExternalApplication. This address will be used later on by the ExternalAccessAgent to create a proxy of the server object.

**main(...)**

At the beginning of its main(...) method, the ExternalApplication performs the following steps:

1. creation of the server object
2. creation of a `ExternalCommService` object
3. start of a communication receiver on the `ExternalCommService` object
4. registration of the server object at the `ExternalCommService` object.

After these steps, the server object is accessible for clients via the Grasshopper communication service, provided that the client knows the address of the communication receiver as well as the class and (user-defined) identifier of the server object.

Now the ExternalApplication acts as client on several Grasshopper components by performing the following steps:

5. creation of a proxy of the region registry.
6. retrieval of a list of all agencies that are registered at the region registry
7. creation of a proxy of the first agency in the retrieved list
8. creation of a new place inside the contacted agency
9. creation of an ExternalAccessAgent in the place 'InformationDesk' of the contacted agency
10. creation of a proxy of the created agent by using the agent's server interface IExternalAccessAgent
11. invocation of an agent method, ordering the agent to move to the place that has been created in step 8.

Example 28: ExternalApplication

```
package examples.externalCom;

import de.ikv.grasshopper.communication.*;
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.util.SearchFilter;
import de.ikv.grasshopper.type.*;
import examples.simpleCom.*;

// This class realizes a stand-alone application that
// acts as client as well as a server for the agent
// 'ExternalAccessAgent'.
public class ExternalApplication
{
  public static void main(String args[])
             throws Exception {
```

```
// Creation arguments:
//  args[0] = Region registry address
//  args[1] = Server object identifier
//  args[2] = Communication service address
//  args[3] = Agent code base
if (args == null || args.length < 4) {
  System.out.println("## ExternalApplication:\\
  Creation arguments needed: <registryAddress>\\
  <serverObjectId> <commServiceAddress>\\
  <agentCodebase>");
  System.out.println("## Exiting.");
  System.exit(1);
}
String registryAddress = args[0];
String serverObjectId = args[1];
GrasshopperAddress commServiceAddress =
  new GrasshopperAddress(args[2]);
String agentCodebase = args[3];

// Create server object
System.out.println("## ExternalApplication:\\
Creating server object");
ServerObject serverObject = new ServerObject();

// Create communication service
ExternalCommService commService = new
  ExternalCommService();

// Start communication receiver
commService.startReceiver(commServiceAddress);

// Register server object at communication service
System.out.println("## ExternalApplication:\\
Registering server object");
commService.registerObject(serverObjectId,
  (ServerObject) serverObject);

// Contact region registry
System.out.println("## ExternalApplication:\\
Contacting region registry '" + registryAddress +
  "'.");
GrasshopperAddress regionAddr =
  new GrasshopperAddress(registryAddress);
IRegionRegistration regionProxy =
  (IRegionRegistration)
    ProxyGenerator.newInstance(
      IRegionRegistration.class,
      regionAddr.generateRegionId(),
```

```
      regionAddr);

// Request a list of all registered agencies from
// the region registry
AgentSystemInfo[] agentSystemInfo =
  regionProxy.listAgencies(new SearchFilter());
System.out.println("## ExternalApplication: " +
  agentSystemInfo.length + " agencies found.");
for (int i = 0; i < agentSystemInfo.length; i++){
  System.out.println("##     " + (i+1) + ". " +
    agentSystemInfo[i].getLocation());
}

// Contact first agency of the list
System.out.println("## ExternalApplication:\\
Contacting agency '" +
agentSystemInfo[0].getLocation().
generateAgentSystemId() + "'.");
GrasshopperAddress address =
  agentSystemInfo[0].getLocation();
String serverAddresses[] =
  regionProxy.lookupCommunicationServer(
    address.generateAgentSystemId());
System.out.println("## ExternalApplication:\\
The agency has the following communication\\
servers:");
for(int i = 0; i < serverAddresses.length; i++){
  System.out.println("##     " +
    serverAddresses[i]);
}
System.out.println("## ExternalApplication:\\
Selecting server " + serverAddresses[0]);
GrasshopperAddress agencyAddress =
  new GrasshopperAddress(serverAddresses[0]);
IAgentSystem agencyProxy = (IAgentSystem)
  ProxyGenerator.newInstance(
    IAgentSystem.class,
    agencyAddress.generateAgentSystemId(),
    agencyAddress);

// Create a new place inside the contacted agency.
System.out.println("## ExternalApplication:\\
Creating place 'NewPlace' in contacted agency.");
agencyProxy.createPlace("NewPlace", "");

// Create an agent in the place 'InformationDesk'
// of the contacted agency.
System.out.println("## ExternalApplication:\\
```

```
      Creating agent inside the place\\
      'InformationDesk'.");
      Object agentCreationArgs[] = new Object[2];
      agentCreationArgs[0] = (String)serverObjectId;
      agentCreationArgs[1] =
        (String)commServiceAddress.toString();
      AgentInfo agentInfo =
        agencyProxy.createAgent(
          "examples.externalCom.ExternalAccessAgent",
          agentCodebase, "", agentCreationArgs);

      // Contact the new agent.
      System.out.println("## ExternalApplication:\\
      Contacting the new agent.");
      IExternalAccessAgent agentProxy =
        (IExternalAccessAgent)
        ProxyGenerator.newInstance(
          IExternalAccessAgent.class,
          agentInfo.getIdentifier(),
          agencyAddress);

      // Move the agent to the new place via the agent's
      // own 'go(...)' method.
      System.out.println("## ExternalApplication: \\
      Moving the agent to the place 'NewPlace'.");
      GrasshopperAddress newLocation = agencyAddress;
      newLocation.setPlace("NewPlace");
      agentProxy.go(newLocation.toString());

      // That's all
      System.out.println("## ExternalApplication:\\
      Ready.");
      System.exit(0);
    }
  }
```

### Class ServerObject

This class represents the server part of the ExternalApplication. It implements the method `printMessage(...)` that is defined in the corresponding server interface IServerObject.

Example 29: ServerObject

```
  package examples.externalCom;

  // This class realizes a server object that is offered
```

```
// by the stand-alone application
// 'ExternalApplication'.
// It is accessible for Grasshopper agents via the
// communication service.
public class ServerObject implements IServerObject
{
  public void printMessage(String msg) {
    System.out.println("## ServerObject: \\
    Receiving message: '" + msg + "'.");
  }
}
```

**Interface IServerObject**

This interface represents the server interface that is implemented by the class ServerObject and that is used by the ExternalAccessAgent for creating a proxy of the server object.

Example 30: IServerObject

```
package examples.externalCom;

public interface IServerObject
{
  public void printMessage(String msg);
}
```

## 9.17.2 Example: ExternalAccessAgent

**Class ExternalAccessAgent**

**Instance variables**

The ExternalAccessAgent maintains the following instance variables:

- `serverObjectAddress`: the complete address of the server object that is provided by the ExternalApplication. This address is delivered to the agent as creation argument.
- `serverObjectId`: the (user-defined) identifier of the server object. This identifier is delivered to the agent as creation argument.
- `serverObjectProxy`: a proxy to the server object, created inside the agent's `init(...)` method

**init(...)**

Inside its `init(...)` method, the agent creates a proxy of the server object and invokes the `printMessage(...)` method on this object. For the creation of the proxy, the client agent uses the server object's interface, the server

object identifier, and the address of the communication receiver. All these parameters have been delivered by the ExternalApplication during the agent's creation.

**go(...)**

The agent's `go(...)` method is accessible via the communication service, since it is defined in the agent's server interface IExternalAccessAgent. The ExternalApplication uses this method to move the agent to a new place.

After its creation as well as before and after its migration, the agent invokes the method `printMessage(...)` on the server object of the ExternalApplication.

Example 31: ExternalAccessAgent

```
package examples.externalCom;

import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.communication.*;
import de.ikv.grasshopper.type.Identifier;
import de.ikv.grasshopper.agency.
  AgentCreationFailedException;

// This class realizes an agent that acts as client as
// well as a server for the stand-alone application
// 'ExternalApplication'.
public class ExternalAccessAgent extends MobileAgent
                implements IExternalAccessAgent
{
  GrasshopperAddress serverObjectAddress;
  Identifier serverObjectId;
  IServerObject serverObjectProxy;

  // Creation arguments:
  //   args[0] = Name of the external server object that
  //     has to be contacted by the agent
  //   args[1] = Address of the external communicytion
  //     receiver that has to be contacted by the agent
  public void init(Object[] args) {

    if (args == null || args.length < 2) {
      log("Creation arguments needed:\\
      <ServerObjectName> <ComReceiverAddress>");
      log("Exiting.");
      throw new RuntimeException();
    }
    String serverObjectName = (String)args[0];
    String serverObjectAddr = (String)args[1];
```

```
    // Contact the server object which has been
    // created by the server application
    log("Creating proxy of server object");
    serverObjectAddress = new
      GrasshopperAddress(serverObjectAddr);
    serverObjectId = new
      Identifier(serverObjectName.getBytes());
    serverObjectProxy =
      (IServerObject)ProxyGenerator.newInstance(
        examples.externalCom.IServerObject.class,
        serverObjectId,
        serverObjectAddress);

    // Invoke the server object's method
    log("Notifying server about my creation.");
    serverObjectProxy.printMessage(
      "ExternalAccessAgent created.");
  }

  public String getName() {
    return "ExternalAccessAgent";
  }

  public void afterMove() {
    log("Notifying server about my arrival.");
    serverObjectProxy.printMessage(
      "ExternalAccessAgent arrived.");
  }

  public void go(String location) {
    log("Roger, moving to " + location);
    try {
      log("Notifying server about my migration.");
      serverObjectProxy.printMessage(
        "ExternalAccessAgent moving.");
      move(new GrasshopperAddress(location));
    }
    catch (Exception e) {
      log("Migration failed. Exception = ", e);
      serverObjectProxy.printMessage(
        "ExternalAccessAgent couldn't move.");
    }
  }

  public void live() {
  }
}
```

**Interface IExternalAccessAgent**

This interface represents the server interface that is implemented by the ExternalAccessAgent and that is used by the ExternalApplication for creating a proxy of the agent.

Example 32: IExternalAccessAgent

```
package examples.externalCom;

public interface IExternalAccessAgent
{
  public void go(String location);
}
```

### 9.17.3 Running the Scenario

Requirements:

- A running region registry. Note that the registry has to be started before the agency, and the registry's address has to be specified when starting the agency in order to register them. Please refer to the User's Guide for more information about how to start agencies and region registries.
- One running agency
- If you are using a JDK 1.2 environment, you must have generated proxy classes (named `IServerObjectP` and `IExternalAccessAgentP`) by invoking the Grasshopper stub generator with the interface classes `IServerObject` respectively `IExternalAccessAgent` as input parameter. The files `IServerObjectP.class` and `IExternalAccessAgentP.class` should be stored either in a directory belonging to the Java classpath or in the code base directory. In a JDK 1.3 environment, the proxy classes are not needed. Even if they are available, they will not be used. Instead, the proxies are dynamically generated by the respective clients at runtime.

Running the Example:

Start the ExternalApplication as a stand-alone Java application. The following creation arguments are required:

- The complete address of the previously started region registry. (To determine the address, please use the '`status`' command in the registry's TUI.

- An identifier for the server object that will be created by the ExternalApplication. This identifier will be used by the ExternalAccessAgent for creating a proxy of the server object.
- A complete Grasshopper address for the communication receiver of the ExternalApplication. This address will be used by the ExternalAccessAgent for creating a proxy of the server object.

```
java examples.externalCom.ExternalApplication <registryAddress> <serverId> <comServiceAddress>
```

Now have a look at the output in the terminal window of the ExternalApplication and at the text console and GUI of the running agency.

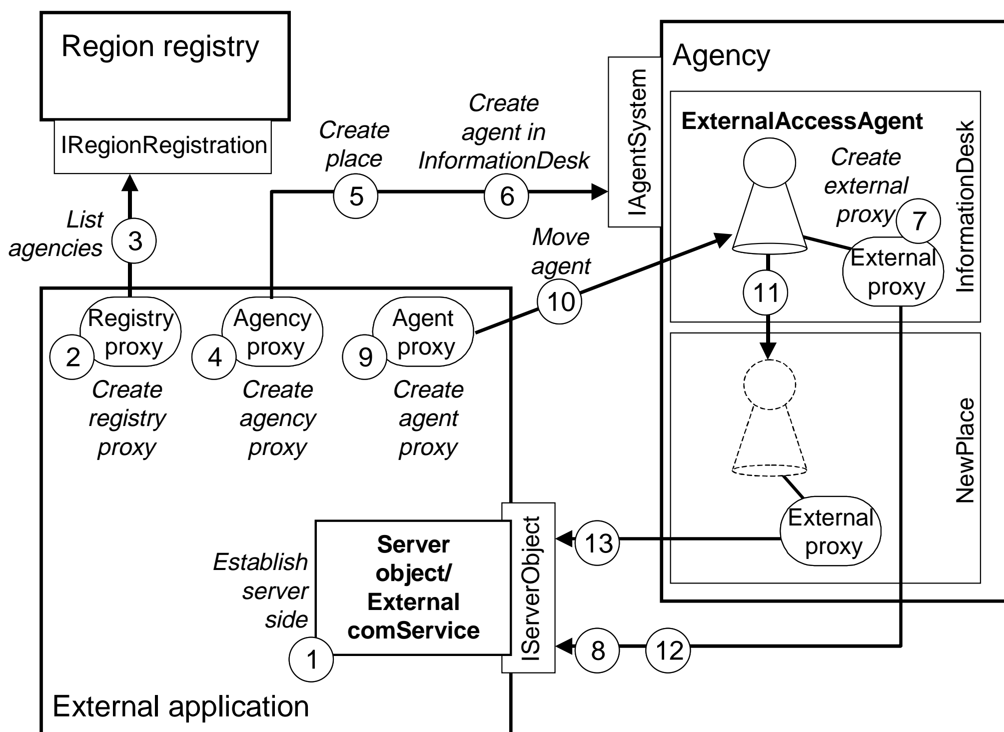



Figure 24: External Application Scenario

After its creation, the ExternalApplication establishes its server side (1) by creating the server object, creating a communication service instance, registering the server object at the communication service, and starting a communication receiver.

After this, the ExternalApplication acts as a client by creating a proxy of the region registry (2), contacting the registry by listing all registered agencies (3), and creating a proxy of the first agency in the retrieved list (4). The ExternalApplication contacts the agency in order to create a new place (5) and a new agent (6). From this point in time on, the ExternalApplication also acts as com-

munication server, accessed by the ExternalAccessAgent that has been created in step (6). The ExternalAccessAgent creates a proxy of the ServerObject (7) and invokes the object's `printMessage(...)` method (8).

Now, the ExternalApplication creates a proxy of the agent (9) and moves the agent to the previously created place (10). Before and after its migration, the agent again notifies the ExternalApplication about its actions by invoking the server object's method `printMessage(...)` (12, 13).

### 9.17.4 Summary

- The Grasshopper communication service enables external applications/ objects to interact with Grasshopper components (agents, agencies, and region registries). The external application may act as server and/or client.
- To act as communication client, an external application has to behave in the same way as usual Grasshopper client agents. That means, the external application simply creates a proxy of the desired server side component (e.g., a server agent) and is then able to invoke server methods.
- To act as communication server, the external component has to implement a server interface (similar to Grasshopper server agents). The object implementing the server interface has to be registered at an instance of the class `de.ikv.grasshopper.communication.ExternalCommService`. After starting a communication receiver on this instance, the server object is available for communication clients.
- In contrast to Grasshopper agents where a unique identifier is automatically generated by the creating agency, an external server object requires a user-defined identifier (whose uniqueness is usually not guaranteed.) Clients have to use this identifier as well as the address of the communication receiver (started on the `ExternalCommService` instance) in order to create a proxy on the external server object.

# 10The Persistence Service

The persistence service is part of the core functionality of Grasshopper agencies. Its purpose is to persistently store the data states of all currently hosted agents as well as runtime information about all places that exist on the agency.

Note that the persistence service is de-activated by default. In order to use its functionality, an agency has to be started with the parameter '-persistence'. Besides, only those agents can be persistently stored which are derived from one of the Grasshopper super classes PersistentMobileAgent or PersistentStationaryAgent, both contained in the package de.ikv.grasshopper.agent.

The reason for disabling the persistence service by default is that it has a negative impact on an agency's performance. The advantage of preserving the data states of all running agents is combined with a slower execution.

The API of the persistence service is divided into two parts, one provided by the agency (interface de.ikv.grasshopper.agency.IAgentSystem) and the other provided by the persistence-supporting agent (classes PersistentMobileAgent, or PersistentStationaryAgent).

Before describing the persistence API, some terms have to be explained:

- save: Saving an agent means to store the agent's data state (i.e., the agent's AgentInfo structure as well as all user-defined, non-transient instance variables of the agent class) in the local file system of the hosting agency. After the save procedure, the agent continues its task execution. The save procedure can be performed either automatically by the hosting agency after a predefined time interval or explicitly via the agency's or the agent's API.

  **save**

  To save a place means to save all agents inside the place. An agency with activated persistence service automatically saves all places periodically after a predefined time interval.

  The purpose of saving agents and places is to preserve their important runtime information in case of a system crash or agency shutdown. After restarting the agency, all saves agents and places are automatically restarted. The agents are supplied with their preserved data states, so that they are able to continue their tasks.

- flush: To flush an agent means to save the agent and, after this, remove its instance and thread from the hosting agency. (In contrast to this, the save procedure does not stop the agent's execution.

  **flush**

An agent can be flushed either automatically after a certain timeout period in which the agent has not been accessed by any clients, or explicitly via the agency's or the agent's API.

The purpose of flushing an agent is to save the runtime resources of an agency by removing those agents from the memory which are not accessed by any clients for a certain period in time. Note that, if an agency is ordered to flush an agent automatically, this flush procedure is performed *independent of the agent's active behavior.* That means, even if the agent is currently performing important tasks inside its `live()` method, the agency flushes the agent when the predefined timeout period has passed. To avoid this, the agent itself can set an infinite timeout period and explicitly initiate its flushing after performing its tasks.

**reload**

- `reload`: The reload procedure is performed on flushed agents in order to re-activate them. During the reload procedure, the agent instance is re-created in the agency in which it has been flushed before. The agent is supplied with its preserved data state, and the agent's thread is re-started. In this way, the agent continues its task execution from that point in execution at which it has been flushed.

  An agency automatically reloads a flushed agent if any client component tries to access it by invoking a method on the flushed agent's proxy. Besides, an agency provides a method that enables the explicit reload of a flushed agent.

**Persistence support via agency API (interface `IAgentSystem`)**

- `flushAgent(...)`: This method flushes an agent at once. The agent to be flushed is specified in terms of its identifier.
- `flushAgentAfter(...)`: This method orders the agency to flush an agent after a certain timeout period in which the agent has not been accessed by any client. The agent to be flushed is specified in terms of its identifier.
- `hasPersistence()`: This methods checks whether the persistence service of the agency is active.
- `reloadAgent(...)`: This method explicitly reloads a flushed agent at once. The agent to be reloaded is specified in terms of its identifier.
- `saveAgent(...)`: This method explicitly saves a flushed agent at once. The agent to be saved is specified in terms of its identifier.
- `saveAgentEvery(...)`: This method orders the agency to save an agent periodically after a certain time period. The agent to be saved is specified in terms of its identifier.

**Persistence support via agent API (classes `PersistentMobileAgent` and `PersistentStationaryAgent`)**

- `flush(...)`: This method flushes the agent at once. Note that an agent should handle this method with care, since the agent itself is not able to reload itself. Instead, the agent has to be reloaded by the hosting agency.
- `getFlushTimeout()`: This method returns the currently valid timeout period after which the agent is automatically flushed. The timeout period defines the period in which the agent has not been accessed by any client.
- `getSaveInterval()`: This method returns the currently valid interval after which the agency automatically saves the agent's data state.
- `beforeFlush()`: This method is automatically invoked by the local agency before an agent is flushed. In this way, similar to the methods `beforeCopy()` and `beforeMove()` of mobile agents, the agent is able to prepare its flushing, if required.
- `afterLoad()`: This method is automatically invoked by the local agency after a flushed agent has been reloaded. In this way, similar to the methods `afterMove()` and `afterCopy()` of mobile agents, the agent is able to prepare the continuation of its task, if required.
- `beforeSave()`: This method is automatically invoked by the local agency before an agent is saved. In this way, similar to the methods `beforeCopy()` and `beforeMove()` of mobile agents, the agent is able to prepare its saving, if required.
- `save()`: This method saves the agent's data state at once.
- `setFlushTimeout(...)`: This method sets the flush timeout, i.e., the period after which the agency flushes the agent if no client has tried to access the agent.
- `setSaveInterval(...)`: This method sets the save interval of the agent, i.e., the period after which the agency automatically saves the agent's data state.

# 10.1 Example: SleepyAgent

The following examples shows how an agent can access the functionality of the Grasshopper persistence service.

**init(...)**

Inside its `init(...)` method, the SleepyAgent configures its personal persistence settings: its save interval and its flush timeout period. Both parameters have to be provided to the agent as creation arguments.

**before-Flush(), af-terLoad(), beforeSave()** The methods `beforeFlush()`, `afterLoad()`, and `beforeSave()` just perform an output in the agency's text console in order to show that they are invoked automatically.

**action()** As explained above, a persistent agent is automatically flushed by the hosting agency if the agent has not been accessed by any clients for the duration of the flush timeout period. The flushed agent is automatically reloaded by the agency if a client tries to access the agent. This client may be another agent that maintains a proxy of the flushed agent, or it may be a user who tries to contact the agent via the agency's UI. Concerning the SleepyAgent, the agent's `action()` method is used for triggering the agent's reload: If the agent has been flushed and the user invokes the action() method via the agency's UI, the agent is automatically reloaded.

**live()** Inside its `live()` method, the agent periodically performs an output every second. The purpose of this behavior is to show the automatic flushing after the predefined timeout period.

Example 33: SleepyAgent

```
package examples.simple;

import de.ikv.grasshopper.agent.*;

public class SleepyAgent extends PersistentMobileAgent
{
  long sheepCount;
  long saveInterval;
  long flushTimeout;

  // Creation arguments:
  //   args[0] = save interval
  //   args[1] = flush timeout
  public void init(Object[] args) {

    if (args == null || args.length < 2) {
      log("Creation arguments needed: <saveInterval>\\
      <flushTimeout>");
      log("Exiting.");
      throw new RuntimeException();
    }

    saveInterval = new
      Integer((String)args[0]).longValue();
    flushTimeout = new
      Integer((String)args[1]).longValue();
    sheepCount = 0;
```

```
    // configure persistence behaviour
    setSaveInterval(saveInterval);
    setFlushTimeout(flushTimeout);
  }

  public String getName() {
    return "SleepyAgent";
  }

  public void action() {
  }

  // This method is automatically invoked before the
  // agent is saved by the agency.
  public void beforeSave() {
    log("Saving my memory...");
  }

  // This method is automatically invoked before the
  // agent is flushed by the agency.
  public void beforeFlush() {
    log("Falling asleep...");
  }

  // This method is automatically invoked after the
// agent has been loaded by the agency.
  public void afterLoad() {
    log("Waking up...");
  }

  public void live() {

    while (true) {
      log("Counting sheeps (" + sheepCount + ")...");
      sheepCount++;
      try {
        Thread.currentThread().sleep(1000);
      }
      catch (InterruptedException e) {
        log("Exception caught.", e);
      }
    }
  }
}
```

Requirements:

- one running agency, started with enables persistence. Please refer to the User's Guide for information about how to start an agency.

Running the example:

Create the SleepyAgent inside the running agency via the agency's UI (1). The required parameters are the agents save interval and its flush timeout period, both given in milliseconds.

If you are using the textual user interface of the agency, please create the agent by means of the following command (the given parameter values are just meant as examples and may be changed):

```
cr a examples.simple.SleepyAgent 4000 10000
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

After its creation, the agent periodically performs an output in the agency's text console, until its timeout period has exceeded. At this point in time, the agency flushes the agent.

In order to re-activate the agent, invoke its `action()` method via the 'invoke' command of the agency. In this way, you try to contact the agent as a client, and this orders the agency to reload the agent. After its reload, the agent continues performing its boring outputs. Note that the counter variable starts with the value that was valid when the agent was flushed.

As a variation of the scenario, please start the agent as described above, and shutdown the agency after a while. (The agent should have been saved at least once before the shutdown.)

Restart the agency. You will see that the agent will be automatically recreated and continue performing its outputs. Note that the counter variable starts with the value that was valid when the agent was saved before the agency's shutdown.

## 10.2 Summary

- The persistence service is part of the core functionality of Grasshopper agencies.
- By default, the persistence service is disabled. To activate it, an agency has to be started with the parameter '`-persistence`'. The reason is that the persistence service has negative impact on an agency's performance.

- Only those agents can be persistently stored which are derived from one of the Grasshopper super classes `PersistentMobileAgent` or `PersistentStationaryAgent`.
- Saving an agent means to preserve its data state in the file system of the hosting agency. Saving a place means to save all agents running inside the place.
- Flushing an agent means to save the agent and afterwards remove the agent from the agency.
- Reloading an agent means to re-activate a flushed agent. The agent continues its task from that point in execution at which it has been flushed.

# 11 Special Places

By default, every Grasshopper agency offers the same functionality to all hosted agents, provided via the interface `IAgentSystem` (see Section 9.11). This functionality is associated with the entire agency. That means, an agent can access the interface `IAgentSystem` independent of the place in which the agent is currently running. Even remote access across a network is possible.

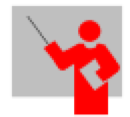
In addition to this default functionality, a user can add specific capabilities to single places within an agency. This place-related functionality is only accessible from within the place.

**Defining a place service**

The following steps are needed for allocating additional functionality to a place:

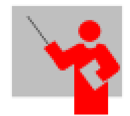
1. Definition of a Java interface whose methods shall represent the place functionality. These interface methods will only be accessible for agents running within this place.

   Example:
   `interface IPlaceService {...}`



2. Realization of a Java class that implements the methods defined by the interface.

   Example:
   `class PlaceService implements IPlaceService {...}`



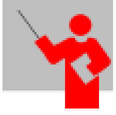
**Place property file**

3. Creation of a place property file which defines the association between a specific place name and its desired functionality.

   The place property file must be stored within a directory that belongs to the system's Java classpath, and the filename must be `<PlaceName>.properties` where `<PlaceName>` has to be substituted by the name of the place which shall provide the additional functinality.

   Inside the file, the following two properties must be defined:

   - InterfaceClass (referring to the place interface as defined in step 1 above)
   - InterfaceImpl (referring to the interface implementation as defined in step 2 above)



Example:

File name = „SpecialPlace.properties“

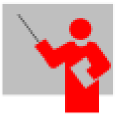
File content:

```
InterfaceClass=IPlaceService
InterfaceImpl=PlaceService
```

After performing the three steps described above, every place with the name 'SpecialPlace' will automatically offer the functionality realized by the class PlaceService.

**Searching a place service**

In order to find a place that offers a specific functionality, an agent can search for this place either inside the local agency or inside an agency domain service. The search key may be the place name or the name of the interface offered by the place.



Example:

Looking for a place inside the local agency

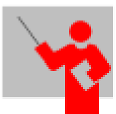
```
PlaceInfo placeList[];
placeList = getAgentSystem().listPlaces(
  new SearchFilter(
  "INTERFACENAME=examples.place.IPlaceService"));
```

Looking for a place inside the whole region

```
PlaceInfo placeList[];
placeList = getRegion().listPlaces(
  new SearchFilter(
  "INTERFACENAME=examples.place.IPlaceService"));
```

**Accessing a place service**

In order to access the functionality of a specific place, an agent has to migrate to this place and invoke the method getPlace() which is provided by the agent's superclass Agent. On the returned reference, the agent invokes the method getInterface() and cast the result to the interface class of the place service.



Example:

```
IPlaceService placeService =
  (IPlaceService)(getPlace().getInterface());
```

The concept of adding functionality to single places can be of particular interest in combination with defining place-specific access rights. In this way it is possible to restrict the access to a specific place (and its functionality) to those agents which have certain certificates. Please refer to Chapter 12 for further information.

The idea behind place services is that they should only be accessible within the offering places. In order to follow this concept when implementing your own place service, simply avoid making the service class or interface serializable. In this way, an agent must release its reference to the place service before leaving the place, since an agent is generally not able to migrate while maintaining non-serializable instance variables.

## 11.1 Example Scenario for Special Places

The following scenario consists of the following three classes/interfaces, covered by the package `examples.place`:

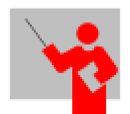
- `PlaceService` (see Example 34 in Section 11.1.1): The class that realizes the place functionality and that is accessible only from inside the place.
- `IPlaceService` (see Example 35 in Section 11.1.1): The interface that defines the place functionality and that is implemented by the class `PlaceService`.
- `PlaceAccessAgent` (see Example 37 in Section 11.1.2): An agent accessing the place functionality that is realized by the class `PlaceService`.

Beside these classes/interfaces, this example scenario comprises a place property file (see Example 36 in Section 11.1.1). This file defines which place shall offer the additional functionality.

### 11.1.1 Example: PlaceService

The purpose of the class shown in Example 34 is to be added to a specific place inside an agency. The method `serviceAccess()` represents the functionality of the place which shall be only accessible for agents running inside the place.

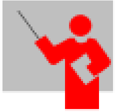
Example 34: PlaceService



```
package examples.place;

import de.ikv.grasshopper.type.*;

public class PlaceService implements IPlaceService
{
```

```
    int accessCount;

    public PlaceService() {
      System.out.println("I'm a special place.");
      accessCount = 1;
    }

    public void serviceAccess() {
      System.out.println("You're the " + accessCount++
        + ". agent accessing my service.");
    }
  }
```

The interface `IPlaceService` shown below is meant to be used by agents in order to access the class PlaceService.



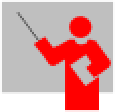
Example 35: IPlaceService

```
  package examples.place;

  public interface IPlaceService
  {
    public void serviceAccess();
  }
```

Example 36 shows a place property file which specifies that the interface `IPlaceService` and the corresponding implementation `PlaceService` realize the additional functionality for specific places. This file has to be stored inside the classpath of the agency which shall be able to create such special places. The name of the property file must be

```
<placeName>.properties
```

where `<placeName>` is the name of the place that shall offer the additional functionality.



Example 36: Place Property File

```
InterfaceClass=examples.place.IPlaceService
InterfaceImpl=examples.place.PlaceService
```

### 11.1.2 Example: PlaceAccessAgent

The PlaceaccessAgent maintains the following instance variables:

- `state`: This variable indicates the case statement within the `live()` method with which the agent shall start its execution after the next migration. Please refer to Chapter 6 for learning about the concept of execution states in the context of Grasshopper.
- `placeName`: This variable maintains the name of the place which provides the additional functionality needed by the agent.
- `placeService`: This variable holds a reference of the place's service class. Note that this variable is not declared transient and thus belongs to the agent's data state. However, in contrast to the remaining data state (comprised by the variables `state` and `placeName`), the interface `IPlaceService` as well as the associated implementaton `PlaceService` are intentionally(!) not serializable (i.e. they do not extend/implement the interface `java.io.Serializable`). Thus, the agent has to set `placeName` to `null` before its next migration. In this way it is not possible for the agent to access the place service from outside the place.

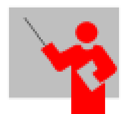
The agent's action() method is used to set the instance variable placeService to null.

The live() method is separated into two execution blocks.

Within the first block (`state` = 0), which the agent processes after its initial creation, the agent looks for a place that offers the interface `examples.place.IPlaceService`. If such a place exists within the local agency, the agent determines the place name, sets `state` to 1 and migrates to this place.

The agent performs its second execution block (`state` = 1) after migrating to the desired place. Inside this block, the agent retrieves a reference to the place service class `PlaceService` and invokes the offered method `serviceAccess()`. After this, the live() method ends.

Example 37: PlaceAccessAgent



```
package examples.place;

import de.ikv.grasshopper.agent.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.util.*;
import de.ikv.grasshopper.communication.*;

public class PlaceAccessAgent extends MobileAgent
{
  int state;
  String placeName;
```

```
IPlaceService placeService;

public void init(Object[] creationArgs) {
  state =0;
}

public String getName() {
  return "PlaceAccessAgent";
}

public void action() {
  log("Releasing place reference.");
  placeService = null;
}

public void live() {
  PlaceInfo placeList[];
  GrasshopperAddress newLocation;

  switch (state) {
    case 0:
      // Look for a place with the needed
      // functionality.
      log("Looking for place with interface\\
      'IPlaceService'...");
      placeList = null;
      placeList = getAgentSystem().listPlaces(
        new SearchFilter("INTERFACENAME=examples.\\
        place.IPlaceService"));
      if ((placeList != null) &&
          (placeList.length > 0)) {
        // Take first place of the found list
        placeName = placeList[0].getName();
        // Move to the special place
        newLocation = new GrasshopperAddress(
            getAgentSystem().getInfo().
            getLocation().toString() + "/" +
            placeName);
        try {
          log("Moving to " + placeName);
          state = 1;
          move(newLocation);
        }
        catch(Exception e) {
          log("Coudn't move to right place." +
            newLocation.toString());
          state = 0;
        }
```

```
        }
        else
          log("Couldn't find right place.");
        break;
      case 1:
        // Check if agent is running inside the
        // desired place
        if (getInfo().getLocation().getPlace().
            equals(placeName)) {
          // Desired place reached.
          // Get place interface
          placeService = (IPlaceService)
            (getPlace().getInterface());
          // Access place functionality
          log("Accessing place functionality");
          if (placeService != null)
            placeService.serviceAccess();
          else
            log("I got lost. Place does not \\
            provide needed service.");
        }
        else {
          // Agent is not in the desired place
          log ("Migration has failed. Place '" +
            placeName + "' not reached.");
          if (getInfo().getLocation().getPlace().
              equals("InformationDesk"))
            log("Maybe I don't have the needed \\
            access rights :-(");
        }
        state = 0;
        break;
    }
    log("Exiting.");
  }
}
```

### 11.1.3 Running the Scenario

This section explains how to run the example whose parts (i.e., `PlaceService`, `IPlaceService` and `PlaceAccessAgent`) have been introduced in the previous sections.

Requirements:

- One running agency.

- The classes PlaceService and IPlaceService have been stored in the class-path of the running agency.
- A place property file (see Example 36 in Section 11.1.1) has been created and stored in the classpath of the running agency. The name of this file is `SmartPlace.properties`.
  (Note that you may use any filename ending with '`.properties`'. The character sequence before this suffix represents the name of the place that shall offer the additional functionality.)

<u>Running the Example:</u>

Create a place named '`SmartPlace`' inside the running agency. If you are using the textual user interface of the agency, please create the place by means of the following command:

```
cr p SmartPlace
```

The place will confirm its creation in terms of the following textual output: 'I'm a special place.'.

Create the PlaceAccessAgent inside the running agency. If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.place.PlaceaccessAgent
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

Since you did not define the place in which the agent shall be crerated, the agency creates the agent within the default place InformationDesk.

The first action of the agent is to look for a place which provides the interface '`examples.place.IPlaceService`'. After this, the agent determines the corresponding place name and migrates to this place. Once arrived, the agent invokes the method `serviceAccess()`.

Now try to move the agent to another place. If you are using the agency's GUI, you can simply do this via drag and drop. If you are using the TUI, please perform the following command (after substituting the agent number '`#4`', the host name '`myhost`' as well as the port number '`7000`' by the corresponding values of your running agency):

```
m #4 socket://myhost:7000/InformationDesk
```

You will see that the agent cannot migrate. The error message of the agency will be

```
Failed to move agent!
```

```
An agent could not be externalized
```

The reason for this failure is that the agent tried to run away with the reference of the place service in its data state.

Now invoke the agent's `action()` method, either by double-klicking on the agent icon inside the agency GUI or by invoking the following command on the textual console (again by substituting the agent number '`#4`' with the number of your agent):

```
invoke #4
```

The agent releases its reference to the place service.

Try again to migrate the agent by repreating the `move` command mentioned above. You will see that the agent now migrates to the place `InformationDesk`. (However, since the guy seems to be crazy about using the place service, he migrates back to the place `SmartPlace` right after arriving at the `InformationDesk`.)

# 12The Security Service

Grasshopper supports two kinds of security mechanisms:

- External security protects all remote interactions that are performed via the Grasshopper communication service. For this purpose, X.509 certificates and the Secure Socket Layer (SSL) protocol are used. SSL is an industry standard protocol that makes use of both symmetric and asymmetric cryptography. By using SSL, confidentiality, data integrity, and mutual authentication of clients and servers can be achieved.
- Internal security protects interfaces of agencies and agents as well as certain agency resources (such as the local file system) from unauthorized access, performed by agents. This access control is achieved by authenticating and authorizing the owner of the accessing agent. Due to the authentication/authorization results, access control policies are activated. Internal security within Grasshopper is mainly based on the inherent security mechanisms of Java.

Both external and internal security is mainly transparent for agent programmers. The following two sections explain those security aspects which have to be considered when programming Grasshopper agents.

## 12.1 External Security

The administration of the external security comprises the start of secure communication receivers on agencies and region registries, i.e., receivers using one of the protocol types ’socketssl’ or ’rmissl’. This has to be done by an agency user and is thus explained in the User’s Guide.

The only aspect of interest for agent programmers is that an agent can actively select a secure protocol for interacting with remote components or for moving to another agency. Note that in both cases the involved agencies must support external security which requires the installation of a set of additional Java security packages. Please refer to the User’s Guide for further details.

In order to use a secure protocol for remote interactions, the agent has to specify a suitable communication receiver when creating the server proxy.

Example:

```
serverProxy = (IAsyncServerAgent)ProxyGenerator.
  newInstance(
    IAsyncServerAgent.class,
```

```
      serverId,
      „socketssl://myHost:7000/myAgency‟);
```

In order to use a secure protocol for its migration, the agent has to specify a suitable communication receiver as parameter of its move method.

Example:

```
  move(new GrasshopperAddress(
             „socketssl://myHost:7000/myAgency‟));
```

The examples above assume that a secure communication receiver is running on the contacted agency. Apart from the general availability of external security in all involved agencies, this is a prerequisite for agents.

## 12.2 Internal Security

The administration of the internal security comprises the configuration of security policies. This has to be done by an agency user and is thus explained in the User's Guide.

The only aspect of interest for agent programmers is that, depending on an agent's access rights, an exception may be thrown if the agent tries to access resources without having the associated permission. The following exception is thrown in this case:

`de.ikv.grasshopper.security.AccessControlException`

The first parameter of this exception presents a textual description of the access violation. This description has one of the following general contents:

1. Agent from <owner> has no, bad or unknown signature.
2. Agent from <owner> is denied access to place <placeName>
3. Access denied for <subject>

If the third description applies to a thrown exception, the second parameter of the exception contains an instance of the class `java.security.Permission` which specifies the permission that would have been required for the denied action. The following permissions are checked by the Grasshopper security manager (for a detailed description, please refer to the HTML documentation of the class
`de.ikv.grasshopper.security.GHSecurityManager`):

- accept connections from a specified host address
- access a specified thread or thread group
- access the AWT event queue

- connect to a specified host/port
- connect to a specified object on a specified host/port
- create a new class loader
- delete a specified file
- execute a specified command
- exit the virtual machine
- link a native library
- listen to a specified port
- access members of a specified class
- use multicast communication
- access a specified package
- define classes in a specified package
- initiate a print job
- access system properties
- access a specified system property
- read from a file descriptor
- read from a specified file
- define the security subsystem and trigger the specified action
- set a socket or stream handler factory
- access the system clipboard
- bring up a top-level window
- write to a file descriptor
- write to the specified file

## 12.3 Example: SecretAgent

This example shows how to configure

Example 38: Keytool Usage: Generate Key

```
D:\Utils\jdk1.3\bin>keytool -genkey -alias Bond
Enter keystore password:  agent-007
What is your first and last name?
```

```
  [Unknown]:  James
What is the name of your organizational unit?
  [Unknown]:  GH
What is the name of your organization?
  [Unknown]:  IKV
What is the name of your City or Locality?
  [Unknown]:  Berlin
What is the name of your State or Province?
  [Unknown]:  Berlin
What is the two-letter country code for this unit?
  [Unknown]:  DE
Is <CN=James, OU=GH, O=IKV, L=Berlin, ST=Berlin, C=DE>
correct?
  [no]:  y

Enter key password for <Bond>
        (RETURN if same as keystore password):

D:\Utils\jdk1.3\bin>
```

Example 39: Keytool Usage: List Keys

```
D:\Utils\jdk1.3\bin>keytool -list
Enter keystore password:  agent-007

Keystore type: jks
Keystore provider: SUN

Your keystore contains 2 entries:

bond, Fri Aug 18 10:35:48 GMT+02:00 2000, keyEntry,
Certificate fingerprint (MD5):
DE:4B:B0:6C:99:D9:9C:73:59:16:8E:A0:58:73:3B:6F
mykey, Thu Jul 27 10:01:51 GMT+02:00 2000, keyEntry,
Certificate fingerprint (MD5):
3D:9B:B3:E8:6C:8D:43:BA:60:7D:04:AC:AA:B7:DF:BD
```

Example 40: Keytool Usage: Export Key

```
D:\Utils\jdk1.3\bin>keytool -export -alias Bond -file
Bond.cer
Enter keystore password:  agent-007
Certificate stored in file <Bond.cer>
```

# 13Grasshopper and CORBA

As described in Chapter 9, Grasshopper provides an advanced communication service in order to enable local and remote interactions between the platform components (agencies, agents, and region registries). This service has been designed for coping with the specific demands of mobile, communicating entities. Please refer to Section 9.14 in order to see how the Grasshopper communication service handles migrating client and server agents by forwarding method invocations and results.

**What is CORBA?**

CORBA (Common Object Request Broker Architecture) is the standard distributed object architecture developed by the Object Management Group (OMG) consortium. Since 1989, the OMG has specified an architecture for an Object Request Broker (ORB), i.e., an open software bus, on which object components written by different vendors in different programming languages can interoperate across networks and operating systems. This standard allows CORBA objects to invoke each other in a location-transparent way, i.e., without knowing where the accessed objects reside. Interfaces to CORBA objects are defined by using the OMG-specified Interface Definition Language (IDL). Up to now, a standardized IDL language mapping exists for numerous programming languages.

**Why Grasshopper and CORBA?**

In some cases, a Grasshopper agent may need to interact with existing applications which provide their own programming interfaces. Due to the wide dissemination of distributed applications that are based on CORBA, this chapter explains how Grasshopper agents can act as CORBA clients and/or servers.

**Which CORBA?**

Currently, numerous CORBA implementations are available, realizing more or less parts of the CORBA specifications. Several products offer additional capabilities that go beyond the specifications standardized by the Object Management Group (OMG), providing for example specific communication protocols or services. Concerning the integration of Grasshopper into a CORBA environment, the only prerequisite is that the used CORBA implementation is compliant with the CORBA/IIOP 2.0 Specification (orbos/97-02-25) and the IDL-to-Java Language Mapping (orbos/98-01-06 Final)[1]. Besides, for running the examples included in this chapter, a CORBA Naming Service is required which is compliant with the Naming Service Specification described in CORBAservices: Common Object Services Specification.

**Java IDL**

Java IDL is a CORBA/IIOP 2.0 compliant Object Request Broker provided with the JDK 1.2 (and higher releases). Together with the idltojava compiler

1. The mentioned OMG documents are available via download from the OMG's FTP server: ftp://ftp.omg.org/pub/docs/orbos/.

(downloadable from the Java Developer Connection) which realizes the standardized IDL-to-Java Language Mapping, it can be used to define, implement, and access CORBA objects from the Java programming language.

The CORBA parts of the examples included in this chapter have been generated by using Java IDL. However, it should be possible to use any other Object Request Broker that fulfils the requirements described above.

Note that this chapter does not include an introduction into CORBA. Before working through this chapter, you should know about the concepts and programming basics of this architecture.

## 13.1 CORBA Enhanced Grasshopper Agents

Some time ago, the OMG has initiated several specifications concerning the realization of mobility in a CORBA environment, such as the Objects by Value specification (orbos/98-01-01). However, our objective for achieving an integration of Grasshopper and CORBA is to use only a minimal set of functionality of the underlying CORBA platform, in this way granting compatibility with most of the currently available CORBA/IIOP 2.0 compliant implementations.

In order to implement a Grasshopper mobile agent that provides a CORBA interface, the following issues have to be taken into account:

1. The CORBA-related part of the agent, i.e., the CORBA object maintained by the agent, consists partly of classes that have been generated automatically by the IDL-to-Java compiler of the used CORBA implementation. Depending on the IDL-to-Java compiler, these classes may not be serializable. In this case, a mobile agent that declares these classes as non-transient instance variables will not be able to migrate (see Section 6.3 for an explanation). Even if all classes are serializable, problems may occur after the agent's migration, since some CORBA-related variable values may be associated with the local environment (IP address, port number, etc.) at which they have been assigned. These considerations lead to the

   *First requirement for CORBA-enhanced Grasshopper agents*:

   The CORBA-related part of an agent has to be separated from the remaining code, e.g., by encapsulating this part in one or more separate Java classes. In order to avoid the attempt of an agency to serialize the CORBA-related part during an agent's migration, this part should not be declared as non-transient instance variable of the associated agent. Otherwise, the agent must release all its references to the CORBA part

before migrating to a new location. Once arrived at its destination, the agent can re-create its CORBA part.

2. In a CORBA 2.0 compliant environment, a usual way for a client to retrieve the reference of a CORBA object (i.e., the object's Interoperable Reference, IOR), is to contact a CORBA Naming Service. This service is part of the CORBA Common Object Services Specification (COSS), and its purpose is to maintain name bindings, i.e., mapping between composed object names and IORs. In order to announce the availability of a new CORBA object, a name binding for the object has to be provided to the Naming Service.
   If a Grasshopper mobile agent maintains a CORBA object, and assuming that (according to the serialization issue mentioned above) this CORBA object is not part of the agent's data state, the agent has to re-create the object after each migration. In this way, the object gets a new IOR, and its previous IOR (which is still registered at the Naming Service) becomes invalid. Thus, the CORBA object has to update its name binding by contacting the Naming Service and providing its new IOR. This leads to the

   *Second requirement for CORBA-enhanced Grasshopper agents*:

   If a Grasshopper mobile agent provides a CORBA object, the Naming Service entry of this object has to be updated after each migration of the agent.

The following scenarios show all CORBA-related aspects and procedures that have to be considered when implementing a CORBA-enhanced Grasshopper mobile agent

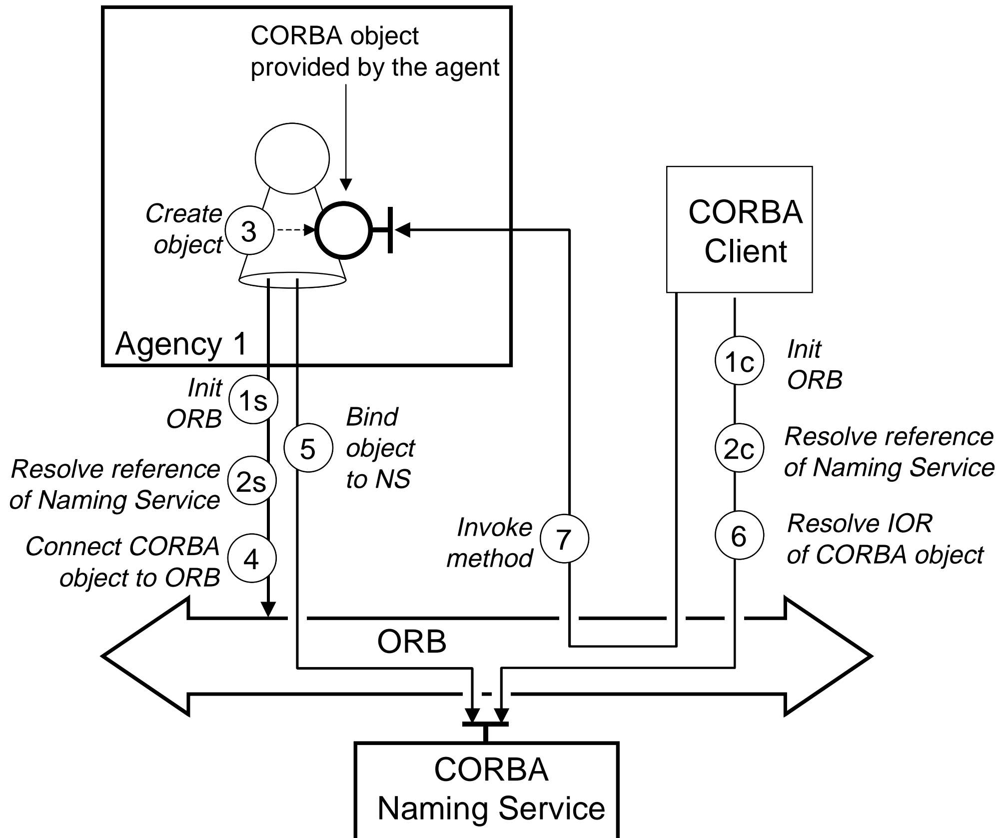



Figure 25: CORBA Object Creation and Connection Establishment

As shown in Figure 25, the initial steps for CORBA servers as well as clients is to initialize the Object Request Broker (1s, 1c)[1] and to request the reference to the running Naming Service from the ORB (2s, 2c).

The next step for the server (agent) is to create the CORBA object (3), to connect this object to the ORB (4), and to bind the object to the Naming Service (5).

Now the CORBA object is available for CORBA clients. That means, a client can contact the Naming Service in order to retrieve the IOR of the CORBA object (6). By using this reference, the client is able to invoke the methods belonging to the interface of the CORBA object (7).

1. The letters s and c indicate whether an interaction step is associated with the server (s) or the client (c) side.

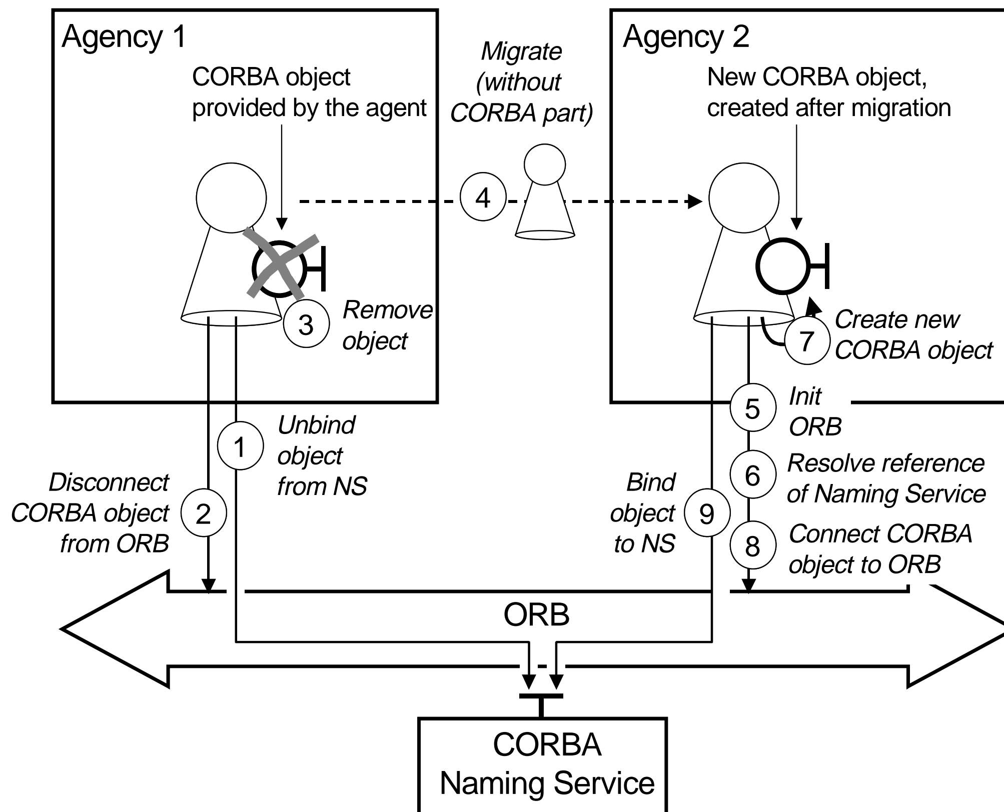



Figure 26: Migration of a CORBA Server Agent

Figure 26 shows the migration procedure of a CORBA-enhanced Grasshopper agent. At first, the agent has to unbind the object from the Naming Service (1) and to disconnect the object from the ORB (2). If the CORBA object has been declared as non-transient instance variable of the agent, the agent has to release all references of the object (3) before migrating to the new location (4). Once arrived at its destination agency, the agent has to perform the same initial steps that have already been described in Figure 25 above: the initialization of the ORB (5), the retrieval of the Naming Service IOR (6), the (re-)creation of the CORBA object (7), the connection of the object to the ORB (8), and the object's registration at the Naming Service (9).

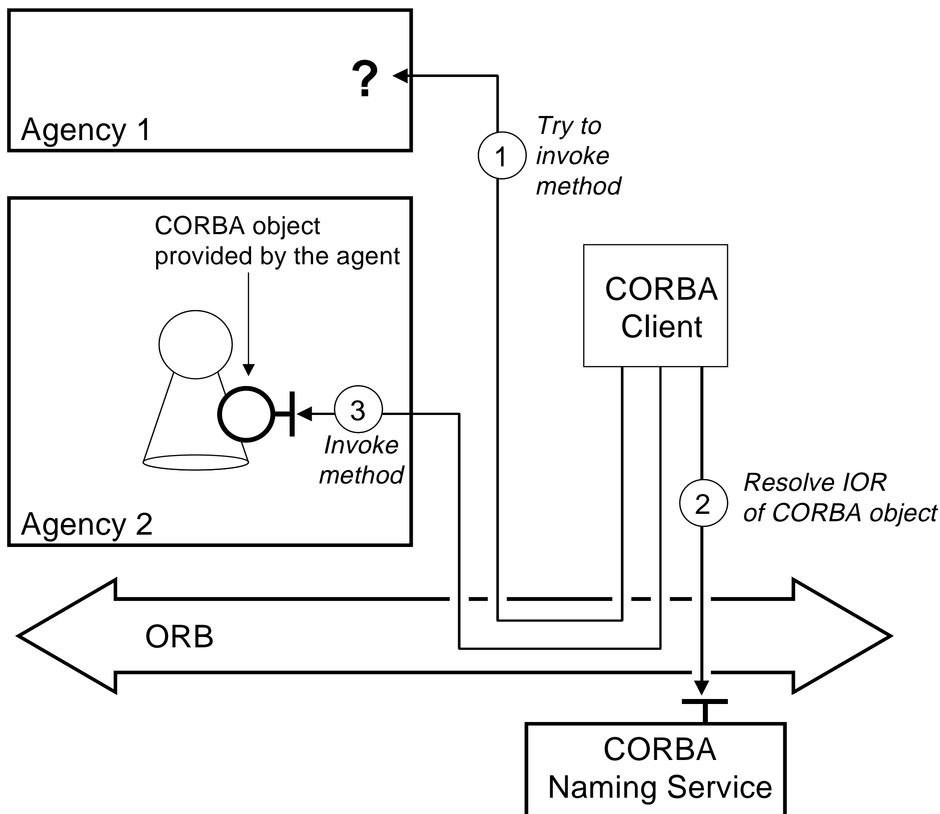



Figure 27: Connection Re-establishment by CORBA Client Agent

After the agent's migration, the client application (mentioned in Figure 25 above) may still want to interact with the CORBA object provided by the agent. Since the client is not aware of the fact that the agent has migrated, it still uses the old (and now invalid) IOR of the CORBA object, as shown in step (1) of Figure 27. The underlying Object Request Broker throws an exception to the client, indicating that the IOR has become invalid, and due to this exception the client requests the new IOR by contacting the Naming Service (2). By using the retrieved IOR, the client is now able to contact the CORBA object at its new location (3).

### 13.1.1 Example: CORBA Enhanced Agents

The following example scenario consists of the following classes:

- `CORBAServerAgent`: This class realizes an agent that maintains a CORBA object and, in this way, offers a CORBA interface to its environment. On request of a user, the agent migrates from one agency to another,

taking into account the CORBA-related issues described in Section 13.1.

- `CORBAServant`: This class represents the implementation of the IDL interface CI_CORBAServerAgent. Note that this class is covered by the same source file as the class CORBAServerAgent.
- `CORBAClientAgent`: This class realizes an agent that acts as CORBA client by periodically invoking a method on the CORBAServerAgent. Once the server agent has changed its location, the client agent requests the new IOR of the server agent's CORBA object from the Naming Service and re-establishes the connection to the server agent.

**IDL Interface CI_CORBAServerAgent**

The following listing shows the CORBA interface to be offered by the CORBAServerAgent. The interface is described by means of the CORBA Interface Definition Language (IDL), and it has to be translated into the Java programming language by means of the IDL to Java compiler of the installed CORBA platform. The prefix 'CI' of the interface stands for 'CORBA Interface'.

Example 41: CI_CORBAServerAgent

```
module examples {
  module corbaCom {
    module idl {
      interface CI_CORBAServerAgent {
        string getAgencyName();
      };
    };
  };
};
```

**Class CORBAServant**

**Instance variables**

The class CORBAServant maintains the following instance variable:

- `agencyName`: This variable holds the name of the agency in which the CORBAServerAgent is currently running. The name is provided to the CORBAServant object during its creation, and it is used as return value of the method getAgencyName().

**getAgencyName()**

The method `getAgencyName()` can be called by any CORBA client that maintains the IOR of the CORBAServant object. The method just performs a textual output and returns the name of the agency in which the associated CORBAServerAgent is currently running.

The source code of this class is shown in Example 42 below.

### Class CORBAServerAgent

**Instance variables**

The class CORBAServerAgent maintains the following instance variables:

- `corbaObjectRef`: This variable is initialized with a reference of the CORBA object (which is an instance of the class CORBAServant).
- `ncRef`: This variable holds a reference of the CORBA Naming Service.
- `name`: This variable maintains the CORBA name which is used to register the CORBAServant object at the Naming Service.
- `orb`: This variable holds a reference to the Object Request Broker.

Note that all instance variables are declared transient in order to exclude them from the agent's data state.

**createCorbaPart()**

Inside its method `createCorbaPart()`, the CORBAServerAgent performs all CORBA-related steps that have been described in Figure 25:

- ORB initialization
- Retrieval of the Naming Service reference
- Creation of the CORBA object (i.e., an instance of the class CORBAServant)
- Connection of the CORBA object with the ORB
- Registration of the CORBA object at the Naming Service

Note that the method `createCorbaPart()` is not only performed after the agent's initial creation, but also after each migration of the agent. Please refer to the beginning of Section 13.1 for a detailed explanation.

**beforeMove()**

The agent uses its `beforeMove()` method to unbind the CORBA object from the Naming Service and to disconnect it from the ORB.

**beforeRemove()**

The `beforeRemove()` method is implemented similar to the `beforeMove()` method: Before the agent is removed, it unbinds its CORBA object from the Naming Service and disconnects it from the ORB.

**live()**

Inside its `live()` method, the agent creates its CORBA object (see method `createCorbaPart()`). After this, the agent requests a new location from the user and migrates to this location.

Example 42: CORBAServerAgent

```
package examples.corbaCom;

import de.ikv.grasshopper.agent.MobileAgent;
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.communication.
```

```
  GrasshopperAddress;
import javax.swing.*;
import java.awt.*;
// import files generated by idltojava compiler
import examples.corbaCom.idl.*;
// import general CORBA stuff
import org.omg.CORBA.*;
// import naming service stuff
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

// The following class represents the actual CORBA
// object that will be created by the CORBAServerAgent
class CORBAServant
          extends _CI_CORBAServerAgentImplBase
{
  String agencyName;

  public CORBAServant(String agyName) {
    agencyName = agyName;
  }

  // This method is the implementation of the IDL method
  // inside the CORBA interface CI_CORBAServerAgent
  public String getAgencyName() {
    System.out.println("## CORBAServerAgent:\\
    Retrieving client request. Returning '" +
    agencyName + "'.");
    return agencyName;
  }
}

// This class realizes the server agent of the CORBA
// communication scenario.
public class CORBAServerAgent extends MobileAgent
{
  // All CORBA-related instance variables are
  // declared transient, since they must not
  // be serialized when the agent migrates.
  transient CORBAServant corbaObjectRef;
  transient NamingContext ncRef;
  transient NameComponent[] name;
  transient ORB orb;

  public String getName() {
    return "CORBAServerAgent";
  }
```

```
// This method initializes the ORB, retrieves the IOR
// of the CORBA Naming Service, creates the CORBA
// object, connects it to the ORB,
// and registers it at the Naming Service.
public void createCorbaPart() {
  try {
    // Initialize the ORB
    log("Initializing ORB...");
    orb = ORB.init(new String[0], null);

    // Get the reference of the Naming Context
    // interface, provided by the CORBA Naming
    // Service
    log("Connecting to Naming Service...");
    org.omg.CORBA.Object objRef =
      orb.resolve_initial_references(
        "NameService");
    ncRef = NamingContextHelper.narrow(objRef);

    // Create the CORBA object
    log("Creating CORBA object...");
    corbaObjectRef = new
      CORBAServant(
        getAgentSystem().getInfo().getName());

    // Connect the CORBA object to the ORB
    log("Connecting CORBA object to ORB...");
    orb.connect(corbaObjectRef);

    // Bind the agent's CORBA object to the Naming
    // Service.
    // The object name will be "CORBAServerAgent"
    log("Binding CORBA object to NS...");
    name = new NameComponent[1];
    name[0] = new NameComponent("CORBAServerAgent",
      "");
    ncRef.bind(name, corbaObjectRef);
    log("Ready for client requests.");
  }
  catch(Exception e) {
    log("Exception: ", e);
  }
}

// The agent's beforeMove() method is used to unbind
// from the Naming Service and to disconnect from
// the ORB.
public void beforeMove() {
```

```
    try {
      // Unbind from Naming Service
      log("Unbinding from NS...");
      ncRef.unbind(name);

      // Disconnect from ORB
      log("Disconnecting from ORB...");
      orb.disconnect(corbaObjectRef);
      orb.shutdown(false);
    }
    catch (Exception e) {
      log("Exception caught. ", e);
    }
  }

  // The agent's beforeRemove() method is used to
  // unbind from the Naming Service
  // and to disconnect from the ORB.
  public void beforeRemove() {
    try {
      // Unbind from Naming Service
      log("Unbinding from NS...");
      ncRef.unbind(name);

      // Disconnect from ORB
      log("Disconnecting from ORB...");
      orb.disconnect(corbaObjectRef);
      orb.shutdown(false);
    }
    catch (Exception e) {
      log("Exception caught. ", e);
    }
  }

  public void live() {
    String location;

    // Create the CORBA object
    createCorbaPart();

    // Request a new location from the user
    location = JOptionPane.showInputDialog(null,
      "Where shall I go?");
    while (location != null) {
      log("Moving...");
      try {
        // Go away!
        // All CORBA-related procudures required for
```

```
            // the migration are included in the agent's
            // beforeMove() method (see above).
            move(new GrasshopperAddress(location));
          }
          catch (Exception e) {
            log("Migration failed. ", e);
            location = JOptionPane.showInputDialog(null,
              "Where shall I go?");
          }
        }
      }
    }
```

### Class CORBAClientAgent

The class CORBAClientAgent maintains the following instance variable:

**Instance variables**

- `ncRef`: This variable holds a reference of the CORBA Naming Service.
- `serverAgentRef`: This variable maintains the IOR to the CORBAServant, i.e., the CORBA object provided by the CORBAServerAgent.
- orb: This variable holds a reference to the Object Request Broker.

Note that all instance variables are declared transient in order to exclude them from the agent's data state.

**connectToOrb()**

Inside its method `connectToOrb(...)`, the client agent performs the following CORBA-related steps that have been described in Figure 25:

- ORB initialization
- Retrieval of the Naming Service reference

**connectToServerAgent()**

Inside the method `connectToServerAgent()`, the client agent requests the IOR of the CORBAServant object from the Naming Service. If the IOR is not available at once, the client agent performs five retries, assuming that the server agent is currently migrating (and thus not registered at the Naming Service). After five fruitless retries, the client agent assumes that something bad has happened to the server agent (such as its removal), and thus exists its loop.

**beforeMove()**

The agent uses its `beforeMove()` method to shut down the ORB.

**beforeRemove()**

The `beforeRemove()` method is implemented similar to the `beforeMove()` method: Before the agent is removed, it shuts down the ORB.

Inside its `live()` method, the client agent initializes the ORB and connects itself to the server agent's CORBA object. After this, the client agent periodically (i.e., once per second) invokes the CORBA method `getAgencyName()` of the server agent's CORBA object. If the client agent catches an

exception, it tries to re-establish the connection to the CORBA object (see method `connectToServerAgent()`). After five fruitless attempts, the client agent terminates.

Example 43: CORBAClientAgent

```
package examples.corbaCom;

import de.ikv.grasshopper.agent.MobileAgent;
// import files generated by idltojava compiler
import examples.corbaCom.idl.*;
// import general CORBA stuff
import org.omg.CORBA.*;
// import naming service stuff
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

// This class realizes the client agent of the CORBA
// communication scenario.
public class CORBAClientAgent extends MobileAgent
{
  // All CORBA-related instance variables are
  // declared transient, since they must not
  // be serialized when the agent migrates.
  transient NamingContext ncRef;
  transient CI_CORBAServerAgent serverAgentRef;
  transient ORB orb;

  public String getName() {
    return "CORBAClientAgent";
  }

  // This method initializes the ORB and
  // retrieves the IOR of the CORBA Naming Service.
  public void initOrb() {
    try {
      orb = null;
      ncRef = null;
      // Initialize the ORB
      log("Initializing ORB...");
      orb = ORB.init(new String[0], null);

      // Get the reference of the Naming Context
      // interface, provided by the CORBA Naming
      // Service
      log("Connecting to Naming Service...");
      org.omg.CORBA.Object objRef =
```

```
        orb.resolve_initial_references(
          "NameService");
      ncRef = NamingContextHelper.narrow(objRef);
    }
    catch (Exception e) {
      log("Exception caught. ", e);
    }
  }

  // This method establishes a connection to the CORBA
  // object provided by the CORBAServerAgent.
  public CI_CORBAServerAgent connectToServerAgent() {
    short numberOfRetries;
    CI_CORBAServerAgent ref;

    log("Connecting to CORBAServerAgent...");

    // Generate the CORBA name of the server agent's
    // CORBA object
    NameComponent nc = new
      NameComponent("CORBAServerAgent", "");
    NameComponent path[] = {nc};

    numberOfRetries = 0;
    ref = null;
    while (numberOfRetries < 5) {
      try {
        // Resolve IOR of the server agent's CORBA
        // object
        ref =
          CI_CORBAServerAgentHelper.narrow(
            ncRef.resolve(path));
        log("Reference retrieved.");
        numberOfRetries = 5;
      }
      catch (Exception e) {
        // The exception may have occurred because the
        // server agent is currently moving.
        // Thus, wait a bit and then retry to establish
        // the connection.
        log("Could not connect to server agent.", e);
        log("   Retrying " + (5-numberOfRetries) +
          " time(s)...");
        try {
          Thread.currentThread().sleep(1000);
        }
        catch (InterruptedException i) {
          log("CORBAClientAgent:", e);
```

```
      }
      numberOfRetries++;
    }
  }
  return ref;
}

// The agent's beforeMove() method is used to shut
// down the ORB.
public void beforeMove() {
  try {
    // Shutdown ORB
    log("Shutdown ORB...");
    orb.shutdown(false);
  }
  catch (Exception e) {
    log("Exception caught. ", e);
  }
}

// The agent's beforeRemove() method is used to shut
// down the ORB.
public void beforeRemove() {
  try {
    // Shutdown ORB
    log("Shutdown ORB...");
    orb.shutdown(false);
  }
  catch (Exception e) {
    log("Exception caught. ", e);
  }
}

public void live() {
  String agencyName;

  initOrb();
  serverAgentRef = connectToServerAgent();

  while (serverAgentRef != null) {
    try{
      // Invoke server method via CORBA
      agencyName = serverAgentRef.getAgencyName();
      log("Server agency = '" + agencyName + "'.");
      Thread.currentThread().sleep(1000);
    }
    catch(Exception e) {
      // The exception may have occurred because the
```

```
          // server agent is currently moving.
          // Try to re-connect to server agent.
          log("Exception when invoking server method. ",
            e);
          log("   Server agent may have moved.");
          log("   Try to re-establish connection.");
          serverAgentRef = connectToServerAgent();
        }
      }
      // The server agent has not re-appeared for five
      // seconds. Thus, the client agent assumes that
      // something terrible has happened to the server...
      log("Permanent failure. Server agent may be dead.\\
      Exiting...");
    }
  }
```

Requirements:

- A running CORBA environment. Note that, depending on the used CORBA platform, different components/processes may be required. The minimal environment (as required in the current Java 2 CORBA installation of Sun Microsystems) consists just of a running CORBA Naming Service.
- At least two running agencies

Running the example:

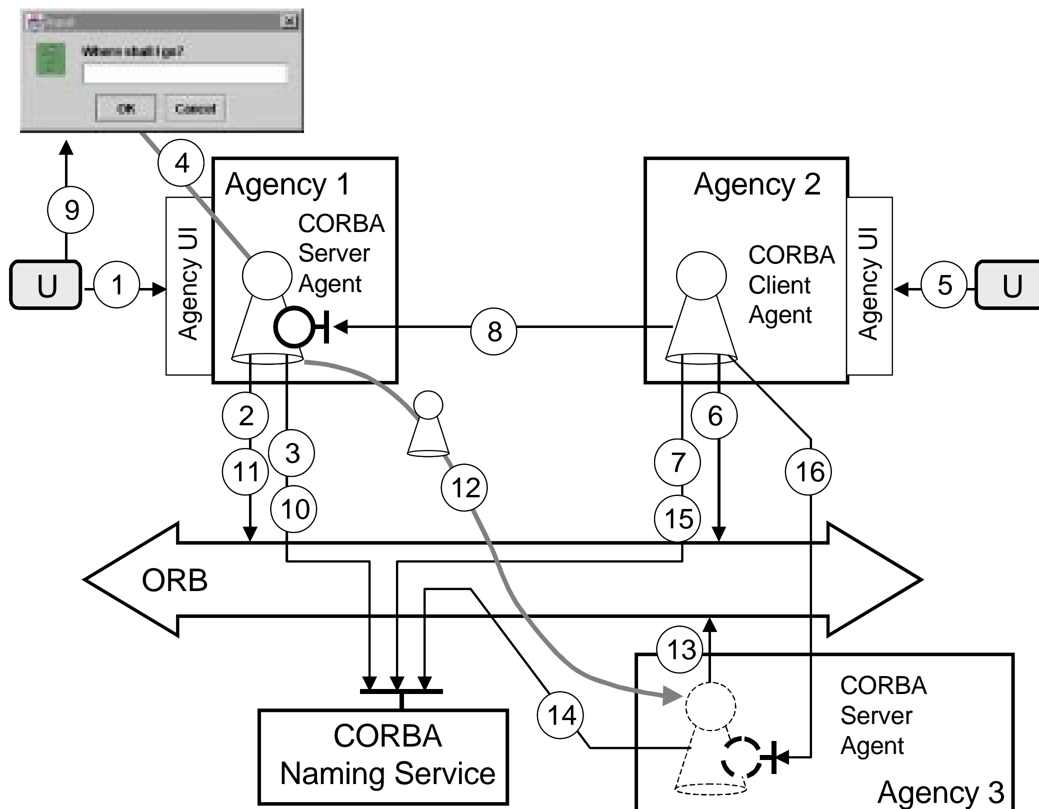



Figure 28: CORBA Agent Scenario

Create the CORBAServerAgent in one of the running agencies (1).

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.corbaCom.CORBAServerAgent
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

The CORBAServerAgent initializes the ORB, resolves the Naming Service IOR, creates its CORBA object, and connects it to the ORB. All these steps are summarized by step (2) of Figure 28. After binding the object to the Naming Service (3), the server agent creates a graphical dialog window, requesting a new location from the user (4).

Create the CORBAClientAgent in one of the running agencies (5).

If you are using the textual user interface of the agency, please create the agent by means of the following command:

```
cr a examples.corbaCom.CORBAClientAgent
```

If the agent's classes are not included in the Java CLASSPATH environment variable, you have to provide a code base as argument of the creation command. Please refer to the User's Guide for detailed information.

The CORBAClientAgent initializes the ORB and resolves the Naming Service IOR (6). After requesting the IOR of the server agent's CORBA object from the Naming Service (7), the client agent periodically (i.e., once per second) invokes the method `getAgencyName()` of the server agent's CORBA object (8) and prints the result on the text console of the local agency.

Type in a new location into the server agent's GUI. After pressing the OK button (9), the server agent unbinds its CORBA object from the Naming Service (10), disconnects it from the ORB, and shuts down the ORB (11). After its migration (12), the server agent again performs the same steps that have been performed in step (2) and (3) after its initial creation (13)/(14). The client agent, still periodically invoking the server method, recognizes that the CORBA object is not available any more. Thus, the client agent again contacts the Naming Service and requests the new IOR of the CORBA object (15). Then the client uses this IOR for establishing a connection to the CORBA object at its new location, and finally the client continues invoking the server method (16).

You can also move the client agent to another location. Since this agent does not provide an own user interface, please use the UI of the local agency.

# A Acronyms

| | |
|---|---|
| ADS | Agency Domain Service |
| API | Application Programming Interface |
| CORBA | Common Object Request Broker Architecture |
| GUI | Graphical User Interface |
| IIOP | Internet Inter-ORB Protocol |
| IOR | Interoperable Reference |
| JDK | Java Development Kit |
| JVM | Java Virtual Machine |
| MASIF | Mobile Agent System Interoperability Facility |
| OMG | Object Management Group |
| ORB | Object Request Broker |
| RMI | Remote Method Invocation |
| SSL | Secure Socket Layer |
| TUI | Textual User Interface |
| UI | User Interface |

# B Glossary

**Active**

one possible >*state* of an >*agent* or >*place*. An agent or place is active when it is currently executing its task, i.e., when the corresponding Java thread is running. Other possible states are >*suspended* (possible for agents and places) and >*flushed* (possible only for agents). After their creation, agents and places are active. For further information about states, please refer to Section 5.5.

**Address**

A Grasshopper address refers to a >*communication receiver* of the desired destination agency, region registry, or external object. Note that multiple communication receivers can be created on single agencies, region registries, and external objects. The purpose may be the need to support different protocols.

A Grasshopper address covers the following components:

- *protocol type*: Type of the protocol to be used for the migration. The following protocols are supported: socket, rmi, iiop, socketssl, rmissl, grasshopperiiop.
- *host name*: Name or IP address of the destination host
- *agency/registry/object name*: Name of the destination agency, region registry, or external object
- *port numbe*r: Number of the port at which the communication receiver of the destination agency is listening.
- *place name*: Name of the destination place. This component is optional. If no place name is specified, the agent migrates to the default place „InformationDesk“ which exists in every Grasshopper agency.

Represented as a String, a complete Grasshopper address has the following format:

```
protocol://hostName:portNumber/agencyName/placeName
```

Depending on the concrete scenario, the address may be simplified by skipping several components. Please refer to Section 5.4 for further details.

**ADS**

see >*agency domain service*.

**Agency**

the runtime environment for Grasshopper >*agent*s. An agency is realized as a Java process, running on its own Java Virtual Machine (JVM). All hosted agents are running inside >*place*s maintained by the agency. An agency provides the required functionality for supporting the execution and management of the hosted agents, including a >*communication service*, >*security service*, >*persistence service*, and access to a management API and an >*agency domain service*. Besides, an agency provides graphical and/or textual user interfaces for administration purposes.

**Agency Domain Service (ADS)**

a registration facility, supporting the localization of Grasshopper agents. Two kinds of agency domain services are supported: Grasshopper-specific >*region registries* and LDAP servers. When starting an agency, the user can (optionally) associate the agency with a running agency domain service. The entire set of all agencies that are registered at the same ADS build a >*region*. During its runtime, an agency automatically registers all hosted agents at this agency domain service, and it de-registers the agents after their removal or after their migration to another location. By accessing the domain service's API, agents are able to search for each other. Besides, the >*proxies* of the Grasshopper >*communication service* access the agency domain service in order to enable location-transparent interactions between agents. An agency domain service provides graphical and/or textual user interfaces for administration purposes.

**Agency Identifier**

a data type for uniquely identifying an >*agency*. Grasshopper uses a common >*identifier* structure for agencies, >*agents*, and listeners. Concerning an agency, a combination of host name and >*agency name* can be used instead of the unique identifier, e.g., in order to create an agency >*proxy* (see Section 9.11.4). The precondition of this alternative way of addressing an agency is that all agencies running on the same host have different names.

**Agency Name**

a user-defined name of an agency, specified during the agency's creation. Agency names do not have to be unique in the scope of the entire environment, but an agency name should only be used once on each host. The reason for this convention is that a combination of host name and agency name may be used for addressing purposes instead of the unique >*agency identifier*.

**Agent**

a self-contained software component which is responsible for autonomously carrying out one or multiple tasks on behalf of a user or another software entity. An Grasshopper agent is implemented in terms of Java classes. An agent's „core" class is referred to as >*agent class*. During its runtime, a Grasshopper agent is realized as a Java thread, running inside a >*place* of an >*agency. >Mobile agents* are able to migrate from one place to another, while >*stationary agents* reside at their creation place for their entire runtime.

**Agent Class**

the „core" of a Grasshopper agent, characterized by inheriting one of the classes `MobileAgent`, `StationaryAgent`, `PersistentMobileAgent`, or `PersistentStationaryAgent`. The agent class implements the `live()` method which defines the agent's active behavior. The `live()` method runs inside the agent's own thread.

**Agent Identifier**

a data type for uniquely identifying an >*agent*. Grasshopper uses a common >*identifier* structure for >*agencies*, agents, and listeners. For further details, please refer to Section 5.1.

**Agent State**

the mode of existence of an >*agent*. All Grasshopper agents can exist in the states 'active' and 'suspended'. >*Persistent agents* can additionally exist in the state 'flushed'. Agents may change their state several times during their runtime. After their creation, agents are active. Please refer to Section 5.5 for further information about states.

**Agent Type**

Grasshopper supports two general types of agents: >*mobile agents* and >*stationary agents*. Both agent types may optionally support persistence (see >*persistence service*), thus resulting in four agent types all together. The type of an agent is allocated during the agent's implementation by deriving the >*agent class* from one of the super classes `MobileAgent`, `StationaryAgent`, `PersistentMobileAgent`, or `PersistentStationaryAgent`.

**Agent Name**

a user-defined name of an agent. Agent names do not have to be unique. For the unique identification of an agent, the automatically generated

*>agent identifier* has to be used. Please refer to Section 5.2 for further details.

**Agent Platform**

the entire distributed agent environment. The Grasshopper platform consists of a set of *>agencies* and one or more *>agency domain service*.

**Agent System**

a synonym for the term *>agency*. Among others, the term 'agent system' is used in the context of the *>MASIF* specification.

**Asynchronous Communication**

a communication mechanism between clients and servers. After invoking a method on the server, the client does not have to wait for the termination of the server method. Instead, the client continues performing its own task. There are different possibilities for the client to retrieve the method result: it can periodically poll for the result, block its task execution until the result arrives, or subscribe to be notified about the result arrival. Asynchronous communication is one possible mode of the Grasshopper *>communication service*. For further information, please refer to Section 9.5.

**Client Agent**

an *>agent* acting as communication client. A client agent invokes methods on a *>server agent* which may run on the same or a remote agency. A Grasshopper agent may act as client and server at the same time.

**Code Base**

a network location that maintains Java class code. Grasshopper supports two kinds of code bases: file systems and HTTP servers. In the context of Grasshopper, code bases are particularly required for the class code of *>agents*. A Grasshopper *>agency* uses Java class loading mechanisms in order to dynamically retrieve the required (agent) classes

- File systems

  The classes of an agent may be maintained in the file system of an agency. In this case, the code base, represented as String, must have the following format:

  `file:/<directory-path>`

  where `<directory-path>` represents a path that leads to the directory in which the agent's class files are stored. Single directories of the

path are separated with slash ('/') characters. Note that on Windows machines, the letter of the maintaining device has to be specified:

`file:/<driveLetter>:/<directory-path>`

- Http servers

  The classes of an agent may be maintained on an Http server. In this case, the code base, represented as String, must have the following format:

  http://<domain-name>/<path>

  where `<domain-name>` and `<path>` are structured in the usual way (i.e., domain components separated with a dot ('.') character, and path components separated with a slash ('/') character).

Further details about the class loading mechanism are provided in Section 5.3.

**Communication Receiver**

realized the server side access point of a communication relationship that has been established via the Grasshopper >*communication service*. A communication receiver is represented by a Grasshopper >*address* which defines the server's protocol type, IP address, object name, and port number. By default, each Grasshopper >*agency* and >*region registry* has one communication receiver that uses the plain >*socket* protocol. Additional communication receivers can be added at start-up time or during runtime. Please refer to the User's Guide for further information.

**Communication Service**

a core service of Grasshopper >*agencies* and >*region registries*. The communication service supports local and remote interactions between Grasshopper components (>*agents*, agencies, region registries). Besides, the service enables external applications to communicate with the Grasshopper environment. The communication service enables synchronous and asynchronous (see Section 9.5), unicast and multicast (see Section 9.9), static and dynamic method (see Section 9.7) invocations. By means of >*proxy objects*, Grasshopper agents are able to communicate with each other in a location-transparent way. That means, >*client agents* and >*server agents* may migrate during a communication session, while the communication service keeps track of the agents and transparently redirects the method invocations and the return values to the actual locations (see Section 9.14).

**CORBA**

see >*common object request broker architecture*

**Common Object Request Broker Architecture (CORBA)**

a comprehensive architecture for the realization of distributed applications, developed by the >*Object Management Group* (OMG). For further details, please refer to `http://www.omg.org`.

**Data State**

the set of variable values that a >*mobile agent* carries with it during a >*migration*. The data state of a Grasshopper mobile agent consists of all instance variables that are not declared transient (see Section 6.3). Before a migration, the data state is >*serialized*, i.e., transformed into a data stream. After transferring this stream to the destination agency, the original object structure is re-created and delivered to the associated agent. In order to improve the migration performance, the data state should be minimized. A set of hints for defining a data state is provided in Section 6.3.

**Dynamic Method Invocation**

a mechanism for client/server interactions where the client is able to construct a method call without having access to the corresponding server (or proxy) class. Dynamic method invocations can be performed synchronously or asynchronously. Dynamic method invocation is one feature of the Grasshopper >*communication service*. For further information, please refer to Section 9.5.

**Dynamic Proxy Generation**

the generation of a >*proxy object* during the runtime of the associated application, without the need for a manually created >*proxy class*. Currently, Grasshopper supports dynamic proxy generation in a JDK 1.3 environment. If a JDK 1.2 environment is used, proxy classes have to be created manually before running the associated application. The manual proxy creation can be achieved by using the Grasshopper >*stub generator*. Please refer to Section 9.2 for further details.

**Execution Block**

a part of the program logic of a >*mobile agent* that is entirely executed at a single location. In the context of Grasshopper, the `live()` method of a mobile agent can be divided into a set of execution blocks. Invocations of the agent's `move()` method should only be performed at the end of an execution block. Before the agent's migration, a counter variable (belong-

ing to the agent's >*data state*) may be initialized, indicating which execution block to perform after the arrival at the new location. Please refer to Section 6.4 for more information about the concepts of execution blocks.

**Execution State**

a set of information indicating the exact point in execution of a process or thread. The standard Java language does not offer the possibility to extract the execution state of a process or thread, and thus (standard) Java-based mobile agent platforms are usually not able to take advantage of >*strong migration*. The current point in an >*agent*'s execution must be mapped onto the agent's >*data state*, e.g., in terms of a counter variable that indicates the number of the >*execution block* that has to be performed after the next migration. Please refer to Section 6.4 for more information about this concept.

**Flush**

the procedure of persistently storing a Grasshopper >*agent* in the file system of the hosting agency. The purpose of flushing an agent is to save agency resources. A flushed agent is automatically re-activated when another entity tries to access it. The functionality of flushing an agent is associated with the Grasshopper >*persistence service*. An agent can only be flushed if the following two preconditions are fulfilled: 1.) The persistence service of the hosting agency is active; 2.) the >*agent class* is derived from one of the classes `PersistentMobileAgent` or `PersistentStationaryAgent`. Please refer to Chapter 10 for further details.

**Flushed**

one possible >*state* of an >*agent*. A flushed agent does not anymore exist as a living object or running thread inside the hosting agency. Instead, the agent has been persistently stored inside the agency's file system. Other possible states are >*suspended* (possible for agents and places) and >*active*. For further information about agent states, please refer to Section 5.5. The functionality of flushing an agent is associated with the Grasshopper >*persistence service*. An agent can only be flushed if the following two preconditions are fulfilled: 1.) The persistence service of the hosting agency is active; 2.) the >*agent class* is derived from one of the classes `PersistentMobileAgent` or `PersistentStationaryAgent`. Please refer to Chapter 10 for further details.

**Home Location**

the >*location* at which a Grasshopper agent has been created.

**Identifier**

enables the unique identification of Grasshopper >*agents*, >*agencies*, and listeners in the entire distributed environment. A Grasshopper identifier consists of the following components:

- a prefix, describing the kind of component that is associated with the identifier (Agent, AgentSystem, Listener). Note that the prefix 'listener' is reserved for internal usage only.
- the Internet address or name of the host on which the identifier has been created
- the date on which the identifier has been created: "yyyy-mm-dd"
- the time on which the identifier has been created: "hh-mm-ss-msms"
- the number of copies of the corresponding agent

Represented as String, a complete Grasshopper identifier has the following format:

<prefix>#<ip-address>#<date>#<time>#<copy-number>

Example of a Grasshopper identifier:

"Agent#123.456.789.012#1999-11-19#15:59:59:0#0"

More information about Grasshopper identifiers can be found in Section 5.1.

**iiop (Grasshopper protocol type)**

one possible protocol used by the Grasshopper >*communication service*. iiop uses CORBA >*Internet Inter-ORB Protocol*. The protocol type 'iiop' is part of a Grasshopper >*address*. Note that, in order to communicate via iiop, the server side (represented by an >*agency*, >*region registry*, or external application) must provide a >*communication receiver* that supports this protocol. iiop required a CORBA runtime environment.

**IIOP**

see >*Internet Inter-ORB Protocol*

**InformationDesk**

the default >*place* that exists in every Grasshopper agency. If an agent wants to migrate and does not provide a specific place of the desired destination agency, the agent automatically moved to the InformationDesk place.

**Internet Inter-ORB Protocol (IIOP)**

a protocol specified by the >*Object Management Group* (OMG) in the context of the >*Common Object Request Broker Architecture* (CORBA). IIOP enables interactions between CORBA client and server objects that are implemented in different languages and residing remotely in different computing environments. For more information, please refer to the CORBA specification, available from the OMG (`http://www.omg.org`).

**Location**

in the context of Grasshopper, the network location of an >*agency*, a >*place* inside an agency , or a >*region registry*. The location of a Grasshopper >*agent* is equal with the location of the place in which the agent is currently running. A location is specified in terms of an >*address*.

**Manual Proxy Generation**

the creation of a >*proxy class* by using the Grasshopper >*stub generator*. Currently, Grasshopper supports >*dynamic proxy generation* in a JDK 1.3 environment. If a JDK 1.2 environment is used, proxy classes have to be created manually *before* running the associated application. Please refer to Section 9.2 for further details.

**MAFAgentSystem**

a CORBA interface for >*agencies*, specified in the >*Mobile Agent System Interoperability Facility* (MASIF) specification. Please refer to the MASIF standard for detailed information.

**MAFFinder**

a CORBA interface for >*region registries*, specified in the >*Mobile Agent System Interoperability Facility* (MASIF) specification. Please refer to the MASIF standard for detailed information.

**MASIF**

see >*Mobile Agent System Interoperability Facility*

**Migration**

the movement of a >*mobile agent* from one >*agency* to another. During the migration, not only the agent's class code is transferred, but also important internal information. Two kinds of migration can be distinguished: >*strong migration* and >*week migration*. Please refer to Section 6.1 for further details.

**Mobile Agent**

an >*agent* that is able to move from one >*agency* to another during its runtime. By dividing `live()` method of a Grasshopper mobile agent into several >*execution blocks*, the agent is able to perform different tasks at different locations. Please refer to Chapter 6 for detailed information about the characteristics of Grasshopper mobile agents.

**Mobile Agent System Interoperability Facility (MASIF)**

the first >*mobile agent* standard of the >*Object Management Group* (OMG). The idea behind MASIF is to improve the interoperability between mobile agent platforms of different manufacturers. Please refer to the MASIF standard for detailed information, available via download from the OMG's FTP server. Please look for the ORBOS document with the number 97-10-05.

Please note that, in contrast to previous Grasshopper releases, the current release does not cover the MASIF functionality in its kernel. Instead, a MASIF library is available as a Grasshopper extension. Please have a look at the IKV Web sites.

**Mobility**

the ability of a >*mobile agent* to migrate from one >*agency* to another during its runtime.

**Multicast Communication**

a communication mechanism between clients and servers. A client addresses a set of servers by performing just a single method call. Multicast communication is one possible mode of the Grasshopper >*communication service*. For further information, please refer to Section 9.5.

**Name**

In the context of Grasshopper, names are associated with >*agents*, >*agencies*, and >*region registries*. Names are user-defined, and they do not have to be unique in the distributed environment. Concerning agencies, names should be unique on single hosts, since the combination of host name and agency name may be used for addressing agencies. Please refer to Section 5.2 (agent names) and to Section 9.11.1/Section 9.11.4 (agency names).

**Object Management Group (OMG)**

founded in April 1989 by eleven companies, the OMG began independent operations as a not-for-profit corporation. The OMG's objective is to develop technically excellent, commercially viable and vendor-indepen-

dent specifications for the software industry. Its best-known achievement is the development of the >*Common Object Request Broker Architecture*, comprising its worldwide standard specifications: CORBA/IIOP, Object Services, Internet Facilities and Domain Interface specifications. Please refer to `http://www.omg.org` for further details.

**OMG**

see >Object Management Group

**Object Request Broker (ORB)**

a communication channel, supporting RPC interactions between distributed software components. Each component belongs to the category client and/or server. The object request broker manages the connection establishment. A well-known specification in this context is the >*Common Object Request Broker Architecture* of the >*Object Management Group*.

**ORB**

see >*Object Request Broker*

**Persistence Service**

a part of the core functionality of a Grasshopper >*agency*. An agency's persistence service is responsible for continuously storing the >*data states* of all >*agents* that are running on the agency. In contrast to other parts of an agency's core functionality which are active by default and cannot be deactivated, the persistence service must explicitly be activated during the agency's start-up. The reason is that the persistence service has a negative impact on an agency's performance. Thus, if persistence is not required, this service should not be activated.

Note that only those agents can be persistently stored whose >*agent class* inherits one of the Grasshopper super classes `PersistentMobileAgent` or `PersistentStationaryAgent`. Please refer to Chapter 10 for further details.

**Persistent Agent**

a Grasshopper >*agent* whose >*agent class* inherits one of the super classes `PersistentMobileAgent` or `PersistentStationaryAgent`. Such Grasshopper agents can be persistently stored if the >*persistence service* of the hosting agency is active.

**Place**

a logical entity inside a Grasshopper >*agency*. Each agency has at least one

place, named 'InformationDesk'. Additional places can be added at the agency's start-up or later during its runtime. Grasshopper >*agents* always run inside a place of an agency. Mobile agents can migrate from one place to another. The destination place of an agent's >*migration* may exist on the same or a remote agency. A user can define a security policy for each single place of an agency. Please refer to the User's Guide for more information.

**Place State**

the mode of existence of a >*place*. Places can exist in the states 'active' and 'suspended'. They may change their state several times during their runtime. After their creation, places are active. Please refer to Section 5.5 for further information about states.

**Proxy Class**

a Java class that has been created via the Grasshopper >*stub generator*. The required input of the stub generator is a >*server interface*. A >*client (agent)* needs an instance of the proxy class, i.e., a >*proxy object*, in order to interact with a >*server (agent)* via the Grasshopper >*communication service*. A proxy class, created via the stub generator, is needed if Grasshopper runs in a JDK 1.2 environment. In JDK 1.3 environments, a proxy object can be dynamically generated during the runtime of the associated application. In this case, no proxy class is required. (Note that in a JDK 1.3 environment the dynamic proxy generation is performed in any case. Even if a proxy class is accessible, it will not be used.) Please refer to Section 9.2 in order to learn about the differences between dynamic and manual proxy creation, and about how to create a proxy class by using the Grasshopper stub generator.

**Proxy (Object)**

a Java object that represents a >*server (agent)* at the location of the corresponding >*client (agent)*. When using the Grasshopper >*communication service*, a client has to locally create a proxy object that corresponds with the derived server. After this, the client invokes methods on the proxy object, and the communication service forwards these invocations to the remote server. The method result is transferred back to the client via the proxy. A proxy object can by created either as an instance of a >*proxy class* that has been generated with the Grasshopper >*stub generator* (required in JDK 1.2 environments, or dynamically via the Java reflection mechanism (possible in JDK 1.3 environments). (Note that in a JDK 1.3 environment the dynamic proxy generation is performed in any case. Even if a proxy class is accessible, it will not be used.) Please refer to Section

9.2 in order to learn about the differences between dynamic and manual proxy creation, and about how to create a proxy class by using the Grasshopper stub generator.

**Region**

a set of Grasshopper >*agencies* that are registered at the same >*agency domain service (ADS)*. One advantage of having a region is that all agents running inside the region can be easily located. Concerning the >*communication service*, a >*client (agent)* need not be aware of the current location of the desired >*server (agent)*. Instead, the communication service automatically contacts the agency domain service for determining the server location. Each agency automatically registers all currently hosted agents at the ADS, and it de-registers the agents after their removal or their migration to another location. Of course, a mobile agent can migrate to an agency that belongs to another region than the agency in which the agent is currently running. However, in this case the complete >*address* of the destination agency must be provided by the agent, while the migration between agencies belonging to the same region just requires the specification of the destination host and destination >*agency name*.

**Region Registry**

one special type of >agency domain service. A region registry is a Grasshopper-specific registration service that is responsible for maintaining information about >*agents* and >*agencies*. Beside region registry, Grasshopper supports LDAP servers as agency domain services.

**Remote Method Invocation (RMI)**

a Java communication mechanism, enabling remote interactions between Java objects that are running on different Java Virtual Machines.

**Resumed**

one possible >*state* of an >*agent* or >*place*. (see >*resumption*). For further information about states, please refer to Section 5.5.

**Resumption**

the procedure of re-activating a suspended >*agent* or >*place*. After the resumption of an agent, the agent continues performing its task which has been interrupted by the agent's >*suspension*. The resumption of a place resumes all agents inside the place. For further information about states, please refer to Section 5.5.

**rmi (Grasshopper protocol type)**

one possible protocol used by the Grasshopper >*communication service*. rmi uses Java >*Remote Method Invocation*. The protocol type 'rmi' is part of a Grasshopper >*address*. Note that, in order to communicate via rmissl, the server side (represented by an >*agency*, >*region registry*, or external application) must provide a >*communication receiver* that supports this protocol.

**RMI**

see >*Remote Method Invocation*

**rmissl (Grasshopper protocol type)**

one possible protocol used by the Grasshopper >*communication service*. rmissl uses >*Remote Method Invocation*, protected via >*Secure Socket Layer*. The protocol type 'rmissl' is part of a Grasshopper >*address*. Note that, in order to communicate via rmissl, the server side (represented by an >*agency*, >*region registry*, or external application) must provide a >*communication receiver* that supports this protocol. rmissl requires external security packages. Please refer to the User's Guide for information about these packages.

**Secure Socket Layer (SSL)**

SSL is one of the most widely used security protocols on the Internet and can be used to protect almost all traffic over TCP/IP networks. The Grasshopper >*communication service* is able to protect all remote interactions (covering for instance agent communication and agent migration) via SSL, presupposed that additional security packages have been installed. Please refer to the User's Guide for information about the required packages.

**Security Service**

one part of the core functionality of Grasshopper agencies. Grasshopper distinguishes between external and internal security:

- External security protects all remote interactions that are performed by using the Grasshopper >communication service. For this purpose, the communication service makes use of X.509 certificates and the >*Secure Socket Layer* protocol. In order to take advantage of protected interactions, a >*communication receiver* must be used which supports one of the secure protocols >*socketssl* or >*rmissl*. Note that these protocols are only available, if external security packages have been installed. Please refer to the User's Guide for information about these packages.
- Internal security protects the resources inside a Grasshopper >*agency*

by defining access rights for >*agents*. Internal security is active by default and does not require the installation of any external packages. Please refer to the User's Guide for information about how to configure the internal security of Grasshopper.

**Serialization**

the procedure of transforming a Java object structure into a data stream. Serialization is a fundamental requirement for >*agent* >*migration*, since it enables the transfer of an agent's >*data state* from one location to another.

**Server Agent**

an >*agent* acting as communication server. A server agent offers methods to a >*client agent* which may run on the same or a remote agency. A Grasshopper agent may act as client and server at the same time. The methods that are to be accessible for client agents must be declared in the server agent's >*server interface*.

**Server Interface**

a Java interface that has to be implemented by the >*agent class* of a Grasshopper >*server agent*. The server interface declares those methods of a server agent that are to be accessible for >*client agents*. The server interface is used as input for the Grasshopper >*stub generator*. The corresponding output is a >*proxy class*. Please refer to Section 9.1 to Section 9.3 for further details.

**Server Proxy**

see >*proxy (object)*

**socket (Grasshopper protocol type)**

one possible protocol used by the Grasshopper >*communication service*. socket uses a plain socket protocol, representing the lowest level of programming to the TCP/IP layer of a network. The protocol type 'socket' is part of a Grasshopper >*address*. Note that, in order to communicate via socket, the server side (represented by an >*agency*, >*region registry*, or external application) must provide a >*communication receiver* that supports this protocol. (Concerning agencies and region registries, a socket communication receiver is running by default.)

**socketssl**

one possible protocol used by the Grasshopper >*communication service*. socketssl uses a plain socket protocol, representing the lowest level of programming to the TCP/IP layer of a network., protected via >*Secure Socket*

*Layer*. The protocol type 'socketssl' is part of a Grasshopper >*address*. Note that, in order to communicate via socketssl, the server side (represented by an >*agency*, >*region registry*, or external application) must provide a >*communication receiver* that supports this protocol. socketssl requires external security packages. Please refer to the User's Guide for information about these packages.

**SSL**

see >*Secure Socket Layer*

**State**

the mode of existence of an >*agent* or >*place*. All Grasshopper agents can exist in the states 'active' and 'suspended'. >*Persistent agents* can additionally exist in the state 'flushed'. Places can exist in the states 'active' and 'suspended'. Agents as well as places may change their states several times during their runtime. After their creation, agents and places are active. Please refer to Section 5.5 for further information about states.

**Stationary Agent**

a Grasshopper >*agent* that is running on the same >*place* for its entire life time. In contrast to this, >*mobile agents* are able to migrate to different locations.

**Strong Migration**

a kind of >*migration* where an agent moves together with its whole >*execution state*. After a strong migration, the agent continues processing its task exactly at the point at which it has been interrupted before the migration. Please refer to Section 6.1 for further details.

**stubgen**

the name of the shell script that starts the Grasshopper >*stub generator.*

**Stub Generator**

a Grasshopper tool for generating >*proxy class*es. The required input of the stub generator is a >*server interface*. Please refer to Section 9.2.1 for information about how to use this tool.

**Suspended**

one possible >*state* of an >*agent* or >*place*. (see >*suspension*). For further information about states, please refer to Section 5.5.

**Suspension**

the procedure of temporarily interrupting an >*agent*'s task execution. After the suspension of an agent, the agent's thread is stopped, while the agent still exists inside the >*agency* as a (passive) Java object. The suspension of a place suspends all agents inside the place. The re-activation of agents and places is called >*resumption*. For further information about states, please refer to Section 5.5.

**Synchronous Communication**

a communication mechanism between clients and servers. After invoking a method on the server, the client is blocked until the termination of the server method. Synchronous communication is one possible mode of the Grasshopper >*communication service*. For further information, please refer to Section 9.5.

**Weak Migration**

a kind of >*migration* where an agent maintains its >*data state* when travelling from one location to another. An agent's data state consists of internal variable values that are serialized at the agent's old location, transferred across the network, and provided to the agent again at the new location. The agent programmer has to decide which variables are to be part of the data state. Please refer to Section 6.1 for further details.

# C Index


**A**

access control 236
active 42, Annex - 3
Address Annex - 3
address 40, Annex - 3
- complete structure 41
- host name 41
- minimal structure 42
- object name 41
- place name 41
- port number 41
- protocol type 40

ADS Annex - 3
agency Annex - 4
- access
  - local 144
  - remote 145
- access by agents 139
- address 40
- identifier 140, Annex - 4
- information 139
- listening to 146
- location 40, 140
- name 140, Annex - 4
- proxy 145
- type 140

agency domain service 163, Annex - 4
- access 163
  - local 168

agent Annex - 5
- address 40
- class Annex - 5
- client Annex - 6
- clone 67
- code base 37
- constructors 24
- copy 67
- creation 23
  - parameters 25
  - via API 25
  - via UI 25
- data state 52
- description 33
- identification 35
- identifier 35, Annex - 5
- interface name 33
- life cycle 42
- location 40
- migration 49
  - procedure 51
  - strong 50
  - weak 50
- mobile 13, Annex - 12
- name 33, Annex - 5
- names & descriptions 36
- persistent 13, Annex - 13
- platform Annex - 6
- properties 32
- removal 25
- server Annex - 17
- state 34, 42, Annex - 5
- stationary 13, Annex - 18
- system Annex - 6
- thread 19
- type Annex - 5
- types 14, 33

asynchronous communication 90, Annex - 6
- dynamic calls 113
- result handling 91
  - blocking 94
  - notification 96
  - polling 95

**C**

certificate 235
class
- ActionAgent 61
- AgencyClientAgent 152
- AgencyInfo 140
  - identifier 140
  - location 140
  - name 140
  - type 140
- Agent 15
  - action() 15, 61
  - afterCopy() 16

- beforeCopy() 16
- beforeRemove() 16, 26
- copy() 16
- getAgentSystem() 16
- getDescription() 16
- getInfo() 16
- getName() 16
- getProperties() 17
- getProperty() 17
- getRegion() 17
- getType() 17
- init() 17, 24
- live() 17, 19
  - structure 54
- log() 17
- remove() 17
- setProperties() 18
- setProperty() 18

AgentInfo 31
- AgentPresentation 33
- AgentSecurityRelated 33
- AgentSpecification 33
- code base 32
- home location 32
- identifier 32
- last location 32
- location 32
- properties 32
- state 34

AgentPresentation 33
- agent name 33
- agent type 33
- description 33
- interface name 33

AgentSecurityRelated 33
AgentSpecification 33
AsyncClientAgent 103
AsyncServerAgent 98
BoomerangAgent 55
ClientAgent 85
CopyAgent 69
CORBAClientAgent 252
CORBAServant 247
CORBAServerAgent 248
DynamicClientAgent 122
DynamicServerAgent 115
ExternalAccessAgent 212
ExternalApplication 207
ExternalCommService 204
- deregisterObject() 205
- registerObject() 205
- shutdown() 205
- startReceiver() 204

FutureResult 91
- addResultListener() 93
- getResult() 91
- getResult() 92
- getTimeout() 92
- isAvailable() 93
- isUserException() 93
- removeResultListener() 93
- setTimeout() 93

GHListener 156
GHSecurityManager 236
HelloAgent 19
MigratingClientAgent 193
MigratingServerAgent 189
MobileAgent 18
- afterCopy() 68
- afterMove() 18, 52
- beforeCopy() 67
- beforeMove() 18, 51
- getType() 18
- move() 18

MulticastClientAgent 133
MulticastServerAgent 131
PersistentMobileAgent 219
- afterLoad() 219
- beforeFlush() 219
- beforeSave() 219
- flush() 219
- getFlushTimeout() 219
- getSaveInterval() 219
- save() 219
- setFlushTimeout() 219
- setSaveInterval() 219

PersistentStationaryAgent 219
- afterLoad() 219
- beforeFlush() 219
- beforeSave() 219
- flush() 219
- getFlushTimeout() 219
- getSaveInterval() 219
- save() 219
- setFlushTimeout() 219
- setSaveInterval() 219

PlaceAccessAgent 229
PlaceService 227
PrintInfoAgent 45
PrintStringAgent 27
RegionClientAgent 175
Serializable 52
ServerAgent 83
ServerObject 210
SleepyAgent 220
StationaryAgent 18
getType() 18
TestDataPacket 117
class diagram
AgencyInfo 140
Agent 14
AgentInfo 31
IAgentSystem 141
IRegion 164
IRegionRegistration 166
client agent Annex - 6
clone 67
code base 32, 37, Annex - 6
communication receiver Annex - 7
communication service 75, Annex - 7
asynchronous com. 90
client side 81
dynamic com 112
external clients 204
external interactions 204
external servers 204
general usage 76
location transparency 76
migration handling 187
multicast com. 127
proxy
concept 75
creation 77
dynamic 80
manual 78
server side 77
static com. 112
synchronous com. 90
unicast com. 127
contact 4
copy 67
procedure 67
CORBA 241, Annex - 8


## D


data state 52, Annex - 8
defining the 53
dynamic communication 112
async. calls 113
generic proxy 113
method calls 113
primitive types 114
user defined classes 113
dynamic method invocation Annex - 8
dynamic proxy generation Annex - 8


## E


example
ActionAgent 61
AgencyClientAgent 149
BoomerangAgent 55
class loading 11
CopyAgent 68
fault tolerance 10
HelloAgent 19
overview 6
PrintInfoAgent 44
PrintStringAgent 26
RegionClientAgent 173
running the 10
scenario
async. communication 97
AsyncClientAgent 100
AsyncServerAgent 98
running the 109
CORBA 246
CORBAClientAgent 252
CORBAServerAgent 248
running the 256
dynamic communication 114
DynamicClientAgent 118
DynamicServerAgent 115
running the 125
external communication 206
ExternalAccessAgent 211
ExternalApplication 206
running the 214
migration 189
MigratingClientAgent 191
MigratingServerAgent 189
running the 199

multicast communication 131
MulticastClientAgent 132
MulticastServerAgent 131
running the 136
simple communication 83
ClientAgent 85
running the 87
ServerAgent 83
special places 227
IPlaceService 228
PlaceAccessAgent 228
PlaceService 227
property file 228
running the 231
SleepyAgent 219
execution block Annex - 8
execution state Annex - 9
external communication 204
external security 235

**F**

file
place property 225
filter 181
flush 217, Annex - 9
flushed 43

**G**

grasshopperiiop 41
group interface 128
group proxy 128
creation 129

**H**

home location 32, Annex - 9

**I**

identifier 32, Annex - 10
IDL 241
iiop 41, Annex - 10
InformationDesk Annex - 10
installation requirements 5
interface
CI_CORBAServerAgent 247
IAgent 19
IAgentSystem 141, 218
addSystemListener() 143
copyAgent() 142
createAgent() 23, 142
createPlace() 143
flushAgent() 142, 218
flushAgentAfter() 142, 218
getAgentState() 142
getInfo() 144
getPlaceState() 143
hasPersistence() 144, 218
invokeAgentAction() 142
listAgents() 142
listMobileAgents() 142
listPlaces() 143
listStationaryAgents() 142
moveAgent() 142
reloadAgent() 142, 218
removeAgent() 143
removePlace() 143
removeSystemListener() 144
resumeAgent() 143
resumePlace() 143
saveAgent() 143, 218
saveAgentEvery() 143, 218
suspendAgent() 143
suspendPlace() 143
IAsyncServerAgent 100
IDirectoryService 165
IDynamicServerAgent 116
IExternalAccessAgent 214
IFutureResult 91
IGroup 128
add() 128
getMembers() 128
getResult() 128
invoke() 128
remove() 128
setType() 129
IListeningAgent 155
IMigratingServerAgent 191
IMobileAgent 19
IMulticastServerAgent 132
IPersistent 19
IPlaceService 228
IRegion 163, 164
getAgentState() 164
getPlaceState() 165
listAgencies() 165

listAgents() 164
listMobileAgents() 164
listPlaces() 165
listStationaryAgents() 164
lookupCommunicationServer() 165
lookupLocation() 164
IRegionRegistration 163, 165
addSystemListener() 167
getAgentState() 166
getPlaceState() 167
listAgencies() 167
listAgents() 166
listMobileAgents() 166
listPlaces() 167
listStationaryAgents() 166
lookupCommunicationServer() 167
lookupLocation() 166
removeSystemListener() 167
IServerAgent 84
IServerObject 211
IStationaryAgent 19
ISystemListener 146, 170
agencyAdded() 170
agencyRemoved() 170
agentAdded 146
agentAdded() 170
agentChanged() 146, 170
agentRemoved() 146, 170
beforeRemove() 146, 171
getIdentifier() 147, 171
placeAdded() 146, 170
placeChanged() 146, 171
placeRemoved() 146, 171
ISystemListenerProvider 165
internal security 235, 236
invoke action() 61

## L

last location 32
listener
identifier 147, 171
result 96
system 146, 170
registration 147, 171
location 32, 40, Annex - 11
location transparency 76

## M

MAFAgentSystem Annex - 11
MAFFinder Annex - 11
manual proxy generation Annex - 11
MASIF Annex - 11
method
action() 15, 61
add() 128
addResultListener() 93
addSystemListener() 143, 167
afterCopy() 16, 68
afterLoad() 219
afterMove() 18, 52
agencyAdded() 170
agencyRemoved() 170
agentAdded() 146, 170
agentChanged() 146, 170
agentRemoved() 146, 170
beforeCopy() 16, 67
beforeFlush() 219
beforeMove() 18, 51
beforeRemove() 16, 26, 146, 171
beforeSave() 219
copy() 16
copyAgent() 142
createAgent() 23, 142
createPlace() 143
deregisterObject() 205
flush() 219
flushAgent() 142, 218
flushAgentAfter() 142, 218
getAgentState 142
getAgentState() 164, 166
getAgentSystem() 16
getDescription() 16
getFlushTimeout() 219
getIdentifier() 147, 171
getInfo() 16, 144
getMembers() 128
getName() 16
getPlaceState() 143, 165, 167
getProperties() 17
getProperty() 17
getRegion() 17
getResult() 91, 128

getSaveInterval() 219
getTimeout() 92
getType() 17, 18
hasPersistence() 144, 218
init() 17, 24
invoke() 128
invokeAgentAction() 142
isAvailable() 93
isUserException() 93
listAgencies() 165, 167
listAgents() 142, 164, 166
listMobileAgents() 142, 166
listPlaces() 143, 165, 167
listStationaryAgents() 142, 164, 166
live() 17, 19
structure 54
log() 17
lookupCommunicationServer() 165, 167
lookupLocation() 164, 166
move() 18
moveAgent() 142
placeAdded() 146, 170
placeChanged() 146, 171
placeRemoved() 146, 171
registerObject() 205
reloadAgent() 142, 218
remove() 17, 128
removeAgent() 143
removePlace() 143
removeResultListener() 93
removeSystemListener() 144, 167
resumeAgent() 143
resumePlace() 143
save() 219
saveAgent() 143
saveAgentEvery() 143, 218
serProperty() 18
setFlushTimeout() 219
setProperties() 18
setSaveInterval() 219
setTimeout() 93
setType() 129
shutdown() 205
startReceiver() 204
suspendAgent() 143
suspendPlace() 143
migration 49, Annex - 11
handling 187
procedure 51
strong 50, Annex - 18
weak 50, Annex - 19
mobile agent 13, Annex - 12
mobility Annex - 12
multicast communication Annex - 12
group interface 128
group proxy 128
result handling 130
termination mode 129
and 129
incremental 130
or 129

**N**

name Annex - 12

**O**

object request broker Annex - 13
OMG Annex - 13
ORB Annex - 13

**P**

permission 236
persistence service 217, Annex - 13
flush 217
reload 218
save 217
persistent agent 13, Annex - 13
place Annex - 13
service 225
special 225
place property file 225
place state Annex - 14
properties 32
protocol 40
type 40
grasshopperiiop 41
iiop 41
rmi 41
rmissl 41
socket 40
socketssl 41
proxy

concept 75
creation 77
dynamic 80
manual 78
generic 113
group 128
proxy class Annex - 14
proxy object Annex - 14

## R

region Annex - 15
region registry Annex - 15
access 163
local 168
remote 169
listening to 170
proxy 169
reload 218
remote method invocation Annex - 15
result listener 96
resume Annex - 15
resumption Annex - 15
rmi 41, Annex - 16
rmissl 41, Annex - 16

## S

save 217
search filter 181
searching 180
secure socket layer 235, Annex - 16
security service 235, Annex - 16
serialization Annex - 17
server agent Annex - 17
server interface Annex - 17
server proxy Annex - 17
socket 40, Annex - 17
socketssl 41, Annex - 17
source code
ActionAgent 61
AgencyClientAgent 152
AsyncClientAgent 104
AsyncServerAgent 98
AsyncServerException 100
BoomerangAgent 55
CI_CORBAServerAgent 247
ClientAgent 85
CopyAgent 70
CORBAClientAgent 253
CORBAServerAgent 248
DynamicClientAgent 122
DynamicServerAgent 115
ExternalAccessAgent 212
ExternalApplication 207
GHListener 157
HelloAgent 19
IAsyncServerAgent 100
IDynamicServerAgent 116
IExternalAccessAgent 214
IListeningAgent 155
IMigratingServerAgent 191
IMulticastServerAgent 132
IServerAgent 84
IServerObject 211
MigratingClientAgent 194
MigratingServerAgent 189
MulticastClientAgent 133
MulticastServerAgent 131
PrintInfoAgent 45
PrintStringAgent 27
RegionClientAgent 175
ServerAgent 84
ServerObject 210
SleepyAgent 220
TestDataPacket 117
special places 225
SSL 235
ssl Annex - 16
state 42, Annex - 18
active 42
diagram 44
flushed 43
suspended 43
static communication 112
stationary agent 13, Annex - 18
strong migration 50, Annex - 18
stub generator Annex - 18
usage 79
stubgen Annex - 18
See stub generator
suspended 43, Annex - 18
suspension Annex - 19
synchronous communication 90, Annex - 19
system listener 146, 170

**T**

termination mode 129
  and 129
  incremental 130
  or 129

**W**

weak migration 50, Annex - 19

**X**

X.509 235