Grasshopper Basics And Concepts

Release 2.2

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

Grasshopper, Release 2.2, Basics and Concepts (Revision 1.0), March 2001.

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

Contents

List of Figures

List of Tables

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 **Basics and Concepts** of mobile agent technology in general and of the Grasshopper platform in particular. However, it is assumed that the principles of mobile agent technology are not completely new to the reader, since this document is not meant as a mobile agent tutorial. The intention is just to give the reader some fundamental background information before using the Grasshopper platform.

1.2 Document Structure

This document is subdivided into the following seven chapters.

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

CHAPTER 2, **What is Grasshopper?**, gives a very brief description of the Grasshopper platform.

CHAPTER 3, **Concepts**, describes the basic terms associated with the Grasshopper environment, such as agents, agencies, and regions. Besides, the Grasshopper communication, security, and persistence mechanisms are explained in detail.

CHAPTER 4, **Frequently Asked Questions**

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 comprises 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 realise 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 the reader in finding specific information.

1.4.1 Fonts

The following font types are used through this manual:

Font	Description
Proportional Font	Used for standard text
Proportional Italic Font	Used to emphasise words or to indicate the first appearance of new terms that can be found in the glossary (Annex 0 of this manual) or in background information boxes marked with the corresponding icon (cf. Section 1.4.2)
`Fixed Font`	Used to identify source code, email addresses and http addresses.

Font	Description
Fixed Italic Font	Indicates commands that a user has to type.
<same font with <> around>	Indicates parts of the command that the user has to substitute by concrete values, e.g. a file name
Fixed Bold Font	Indicates output of the system that is printed into a console window or comments within a source code listing.

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.

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 by the reader. However, it may be interesting for readers who want to know more about the concepts realised by the Grasshopper platform. This background information is additionally highlighted by means of a shaded frame.

This icon indicates useful tips and tricks that facilitate the usage of Grasshopper.

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 fast 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 send an email to `grasshopper2@ikv.de`

In order to retrieve the comments of other Grasshopper users and participate in discussions, please consult the Grasshopper community discussion board at http://www.grasshopper.de/community.

Additional information can be retrieved from http://ww.grasshopper.de/.

2 What is Grasshopper?

Mobile agent technology is a relatively new area in the field of distributed applications. Mobile agents are software components which are able to migrate actively from one physical network location to another. By moving to locations where required information or logic is hosted, mobile agents are able to take advantage of local communication instead of interacting remotely via the network.

Grasshopper is a mobile agent platform that is built on top of a distributed processing environment. In this way, an integration of the traditional client/server paradigm and mobile agent technology can be achieved.

Grasshopper is developed compliant to the first mobile agent standard of the Object Management Group (OMG), i.e. the Mobile Agent System Interoperability Facility (MASIF). The MASIF standard has been initiated in order to achieve interoperability between mobile agent platforms of different manufacturers.

3 Concepts

This chapter provides background information about the concepts of the Grasshopper platform. Among others, this background information comprises the Grasshopper agent environment as well as realised communication and security mechanisms.

3.1 Distributed Agent Environment

This section describes the structure of the Grasshopper Distributed Agent Environment (DAE). The DAE is composed of regions, places, agencies and different types of agents. Figure 1 depicts an abstract view of these entities.

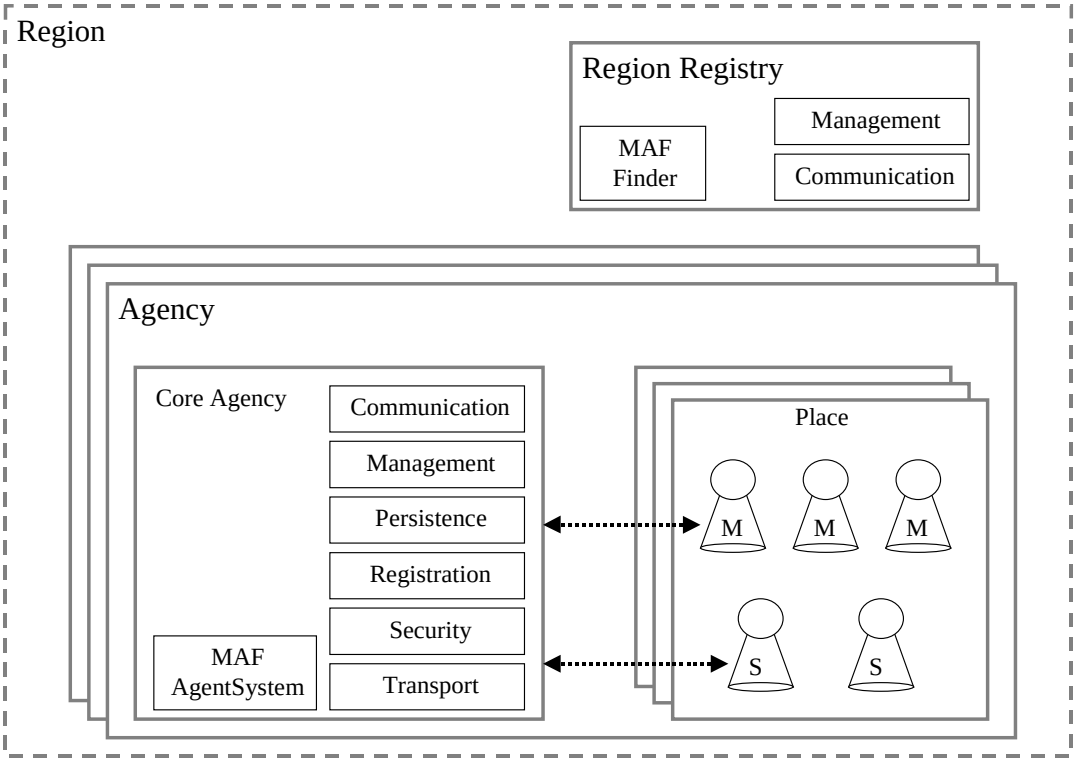


Figure 1: Hierarchical Component Structure

3.1.1 Agents

Up to now, there is no unique definition of a (software) agent. However, agents can be characterised by a set of attributes. The only attribute that is commonly accepted is *autonomy*. Taking this into account, an agent is a computer program that acts autonomously on behalf of a person or organisation.

Two types of agents act in the Grasshopper context, i.e. stationary agents and mobile agents.

3.1.1.1 Mobile Agents

Mobile agents are able to move from one physical network location to another. In this way, they can be regarded as an alternative or enhancement(!) of the traditional client/server paradigm. While client/server technology relies on remote procedure calls across a network, mobile agents can migrate to the desired communication peer and take advantage of local interactions. In this way, several advantages can be achieved, such as a reduction of network traffic or a reduction of the dependency of network availability. Note that mobile agent technology is often regarded as a replacement of the client/server paradigm. In contrast to this, by building Grasshopper upon a distributed, object-oriented middleware, an integration of both approaches is achieved.

Note that an integration of mobile agent technology and client/server technology includes that an agent may be realised either as client or as server. By doing this it is possible for a client agent to migrate to a (traditional) server or for a server agent to migrate to a (traditional) client, thus taking advantage of local interactions instead of communicating via RPC. On the other hand it is possible for two mobile agents to communicate remotely, i.e. not to meet at a common location. It is very important to evaluate for each scenario both possibilities, i.e. to decide whether to use migration plus local interactions or to use remote interactions instead.

There is a significant difference between mobile agents and simple "traditional" mobile code. This difference can be described by two kinds of mobility, i.e. *remote execution* and *migration*.

Remote Execution

Remote execution, known from traditional mobile code, means that a program is sent to a remote location *before* its activation. The program remains at this location during its entire life time.

Migration

Migration means that a program (a mobile agent) is able to change its location *during* its execution. A mobile agent may start its execution at location A, migrate to location B, and *continue* its execution at location B exactly at the point at which it has been interrupted before the migration.

3.1.1.2 Stationary Agents

In contrast to mobile agents, stationary agents do not have the ability to migrate actively between different network locations. Instead, they are associated with one specific location.

3.1.1.3 Agent States

During their life cycle, agents can reside in different states. Grasshopper compliant agents can normally be in one of the following states: active, suspended, or flushed.

Active

An agent is active if it is currently performing its task. An active agent can be suspended or deactivated.

Suspended

An agent is suspended if its task execution is temporarily interrupted. However, the agent remains instantiated and continues its task execution when it is resumed. A suspended agent is unable to communicate and thus not reachable by other parties.

Flushed

A flushed agent is not active any more. Instead, all relevant internal information is permanently stored, e.g. on a hard disk. A flushed agent can be activated again. This means, a new instance of the agent is created, this new instance is supplied with the execution-relevant data, and the agent continues its task execution. In contrast to the suspended state, a flushed agent behaves

the same way as an active one from the viewpoint of other agent. On incoming communication requests the agent will be reactivated automatically.

3.1.2 Agencies

An agency is the actual runtime environment for mobile and stationary agents. At least one agency must run on each host that shall be able to support the execution of agents. A Grasshopper agency consists of two parts, i.e. the core agency and one or more places.

3.1.2.1 Core Agency

Core Agencies represent the minimal functionality required by an agency in order to support the execution of agents. The following services are provided by a Grasshopper core agency:

Communication Service

This service is responsible for all remote interactions that take place between the distributed components of Grasshopper, such as location-transparent inter-agent communication, agent transport, and the localisation of agents by means of the region registry. All interactions can be performed via CORBA IIOP, Java RMI, or plain socket connections. Optionally, RMI and plain socket connections can be protected by means of the Secure Socket Layer (SSL) which is the de-facto standard Internet security protocol. The communication service supports synchronous and asynchronous communication, multicast communication, as well as dynamic method invocation.

For detailed information about the Grasshopper communication service, please refer to Section 3.3.

Registration Service

Each agency must be able to know about all currently hosted agents and places, on the one hand for external management purposes and on the other hand in order to deliver information about registered entities to hosted agents. The registration service is developed to achieve this.

Besides, the registration service of each agency is connected to the region registry (cf. Section 3.1.3) which maintains information of agents, agencies and places in the scope of a whole region.

Management Service

Management services are developed to allow the monitoring and control of agents and places of an agency by external (human) users. Among others, the following functionality is supported:

- Create, remove, suspend and resume agents and places
- Get information about specific agents and places
- List all agents residing in a specific place
- List all places of an agency

Apart from this, configuration management enables human users to specify system, trace, security and communication properties. For detailed information about the configuration of the Grasshopper platform, please refer to the User's Guide.

Transport Service

This service supports the migration of agents from one agency to another. At the destination agency, the agent continues its task processing exactly at the point where it has been interrupted before the migration. The transport service handles the externalisation and internalisation of agents, and the coordination of the actual transfer which is performed by the communication service.

Security Service

Grasshopper supports two kinds of security mechanisms, i.e. external and internal security.

External security protects remote interactions between the distributed Grasshopper components, i.e. agencies and region registries. For this purpose, X.509 certificates and the Secure Socket Layer (SSL) are used. SSL is an industry standard protocol that makes substantial use of both symmetric and asymmetric cryptography. By using SSL, confidentiality, data integrity, and mutual authentication of both communication partners can be achieved.

Internal security protects agency resources from unauthorised access by agents. Besides, it is used to protect agents from each other. This is achieved by authenticating and authorising the user on whose behalf an agent is executed. Due to the authentication/authorisation results, access control policies

are activated. Internal security within Grasshopper is mainly based on the security mechanisms provided by JDK.

For detailed information about the Grasshopper security service, please refer to Section 3.4.

Persistence Service

The Grasshopper persistence service enables the storage of agents and places, i.e. the internal information maintained inside these components, on a persistent medium. In this way it is possible to recover agents or places when needed, e.g. when an agency is restarted after a system crash. Two types of persistence are distinguished, namely implicit and explicit persistence.

Implicit persistence: When the **persistence** service is activated, places are automatically persistent when they are created. That means, a place exists even after the agency has been shut down, and the place will be available again after restarting the agency. The agency owner may also enable the automatic saving of all agents when the agency shuts down. Note that implicit **persistence** is not visible for agent programmers. Instead, it is configured and activated by agency administrators.

Explicit persistence: Three mechanisms can be distinguished:

An agent is persistently stored periodically after a certain time interval without suspending its task execution. The time interval is specified by the agent itself, and after this the agent need not care about the maintenance of its information, since this is done automatically by the persistence service. This mechanism is useful to enable the recovery of agents when an agency is restarted, e.g. after a system crash.

Agents may order the persistence service to terminate them after a certain time of idle processing, i.e. after they have not been used by other entities for a certain period of time. However, the agents remain registered within the agency and the region registry, so that they can be restarted if other entities try to access them. This mechanism is useful to save agency resources.

Agents can be stored explicitly at any time on their own behalf or on behalf of a human user, e.g. the agency administrator. This is useful e.g. if the agency has to be temporarily terminated.

The core functionality explained above can be enhanced in a comfortable way by means of that agents can be created in (or move to) places of an agency.

3.1.2.2 Places

A place provides a logical grouping of functionality inside an agency. For example, there may exist a communication place offering sophisticated communication features, or there may be a trading place where agents offer or buy information or service access. The name of the place should reflect its purpose. For example, in every agency exists by default a place named "InformationDesk". Every agent with no determined place is transported to the "InformationDesk" where it can look for further information.

3.1.3 Regions

The region concept facilitates the management of the distributed components in the Grasshopper environment, i.e. agencies, places, and agents. Agencies as well as their places can be associated with a specific region, i.e. they are registered within one region registry (cf. Section 3.1.3.1). Each registry automatically registers each agent that is currently hosted by an agency associated with the region. If an agent moves to another location, the corresponding registry information is automatically updated. A region may comprise all agencies belonging to a specific company or organisation, thus facilitating its management.

3.1.3.1 Region Registries

A region registry maintains information about all components that are associated with a specific region. When a new component (i.e. an agency, place, or agent) is created, it is automatically registered within the corresponding region registry. While agencies and their places are associated with a single region for their entire life time, mobile agents are able to move between the agencies of different regions. The current location of mobile agents, i.e. the agency and place in which they are residing, is updated in the corresponding region registry after each migration. By contacting the region registry, other entities (including agents and human users) are at any time able to locate agents, places, and agencies residing in a region. Besides, a region registry facilitates the connection establishment between agencies or agents. For instance, agent A which wants to communicate with agent B is able to establish a connection just by knowing the identifier of agent B. The Grasshopper communication service automatically determines the current location of agent B by contacting the region registry and establishes the connection. The same applies to agent migration: An agent is able to migrate just by knowing the name of the desired destination agency. The host name, port number, and

supported transport protocol of the destination is automatically detected by the source agency by contacting the region registry.

Note that the Grasshopper environment can be established even without a region registry. However, in this case agents and agencies must know all information that is required for remote interactions, such as host name, port numbers, communication protocols, etc.

If the Grasshopper environment shall be established *with* a region registry (what is strongly recommended), the registry must be started *before* the first agency is created. The registry is not able to register agencies which have been created before the creation of the registry.

3.2 MASIF

Mobile agents represent a relatively new technology. Today, a large number of mobile agent platforms exist which differ widely in architecture and implementation. These differences prevent any interoperability between MA platforms. However, interoperability between systems of different vendors or manufacturers is a fundamental requirement in order to fulfil the needs of the upcoming open service market. Since the basis for interoperability is the standardisation of the used technology, one requirement regarding mobile agent platforms is standard compliance.

Currently, the most important standardisation body in the area of mobile agents is the Object Management Group (OMG) with its *Mobile Agent System Interoperability Facility (MASIF)* specification. The MASIF standard has been adopted as new OMG technology in February 1998. It comprises several important aspects, such as agent management, mobility, naming, and tracking. Probably, further agent standards will be initiated by the OMG which will all together build a widely accepted standardisation framework for mobile agent technology.

In order to meet future requirements and to provide openness to platforms of different vendors, Grasshopper is developed compliant to the OMG MASIF standard. To do so, you will have to download the Grasshopper MASIF Add-on from http://www.grasshopper.de/.

3.3 Communication Concepts

This section explains the communication concepts of the Grasshopper platform. These concepts are realised by means of the Grasshopper *communication service (CS)* which is an essential part of each core agency (cf. Section 3.1.2.1). The communication service allows location-transparent interactions between agents, agencies, and non-agent-based entities.

Alternatively to the communication service, Grasshopper can use its OMG MASIF-compliant CORBA interfaces for remote interactions (cf. Section 3.2). For this purpose, each agency provides the interface *MAFAgentSystem*, and each region registry provides the interface *MAFFinder*. Both are defined in the MASIF standard which is available on the OMG FTP server. Note that the following sections only describe the Grasshopper communication service. Detailed information about MASIF can be found in the standard itself.

3.3.1 Multi-protocol Support

Remote interactions are generally achieved by means of a specific protocol. The CS supports communication via the *Internet Inter-ORB Protocol* (IIOP), Java's *Remote Method Invocation* (RMI), and *plain socket connections*. To achieve a secure communication, RMI and the plain socket connection can optionally be protected with *the Secure Socket Layer* (SSL) (cf. Section 3.4).

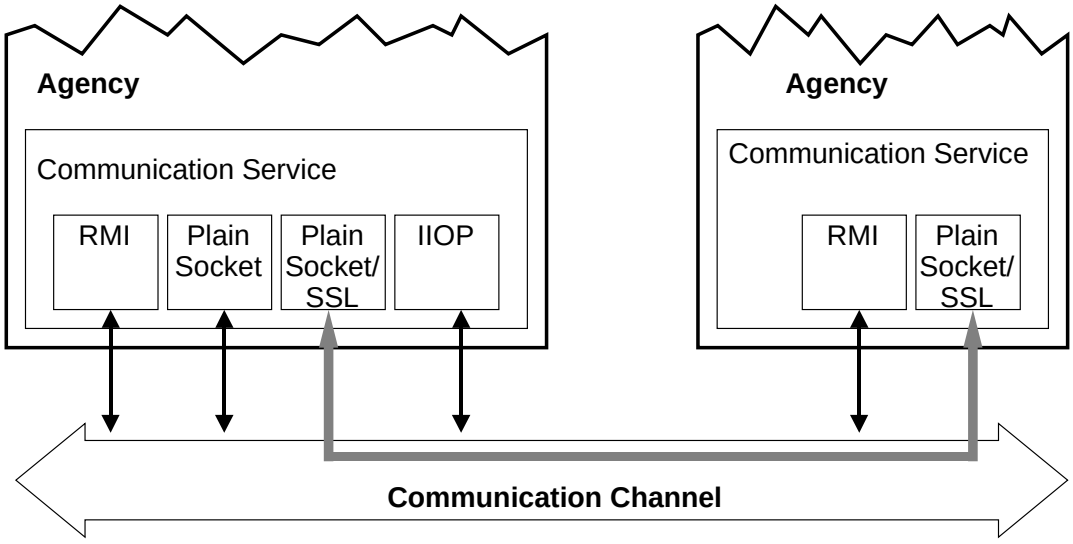


Figure 2: Multi-Protocol Support

- **CORBA IIOP:** The CORBA 2.0-compliant Internet Inter-ORB Protocol can be used in all environments which support CORBA, independent of a vendor-specific ORB implementation. It uses the standard-compliant mechanism to connect to an object using a CORBA Naming Service. Note that CORBA IIOP communication is possible only if both client and server are running in a CORBA-enabled environment (see the Installation chapter in the User's Guide).

- **MAF IIOP:** This protocol is a specialisation of CORBA IIOP developed for agent system interaction. It is introduced in the MASIF standard and provides the connectivity between agent systems of different vendors. Thus, MAF IIOP does not use the Grasshopper communication service and connects to the MASIF interface of the peer agency directly. MAF IIOP has the same requirements as CORBA IIOP. Note that MAF IIOP communication is possible only if both client and server are running in a CORBA-enabled environment (see the Installation chapter in the User's Guide).

- **RMI:** Java Remote Method Invocation (RMI), introduced in JDK 1.1, enables Java objects to invoke methods of other Java objects running on another Virtual Machine (VM). Since this protocol is included in every JDK1.1-compliant VM, all Grasshopper agencies support this protocol by default without any further installation or configuration effort.

- **RMI with SSL:** Using this protocol, RMI is running over sockets protected with the Secure Socket Layer (SSL) protocol. SSL provides secure transport of all data. To run this protocol, the user probably needs to have an additional security package installed (see the Installation Chapter in the User's Guide). For detailed information about the Grasshopper security concepts, please refer to Section 3.4 or additional documents if available on the Grasshopper website.

- **Plain Sockets:** The fastest way of remote interactions is the communication via plain sockets to a specific port of the target host. This technique is robust and avoids the overhead of a distributed object model (apart from the Grasshopper model). Plain socket communication is possible in each Internet-enabled environment, and it is the default protocol used by Grasshopper agencies.

- **Plain Sockets with SSL:** Using this protocol, plain socket connections are protected via SSL. The preconditions for usage are the same as those mentioned for RMI/SSL. For detailed information about the Grasshopper security concepts, please refer to Section 3.4.

Inside a region, Grasshopper is dynamically able to determine the protocols supported by a desired communication peer and to select the most suitable protocol for the remote interactions.

Since the supported communication protocols are realised via a plugin interface, developers can easily integrate new communication protocols by writing their own protocol plugins. In this way Grasshopper is open for future requirements that may come up in the changing communication world.

As an example how to combine several protocols to fulfil specific requirements consider a network in a large company. Usually, this network is connected with the Internet via a router and protected with a firewall running on this router. Thus, only a fixed number of ports are visible to users outside the company's network (called intranet). Grasshopper can use this intranet strategy to protect users from malicious agents by providing one single agency as access point to the intranet, allowing only secure interactions to the outer world. Inside the intranet, agencies may interact by taking advantage of fast (but insecure) connections. Figure 3 shows this example network configuration.

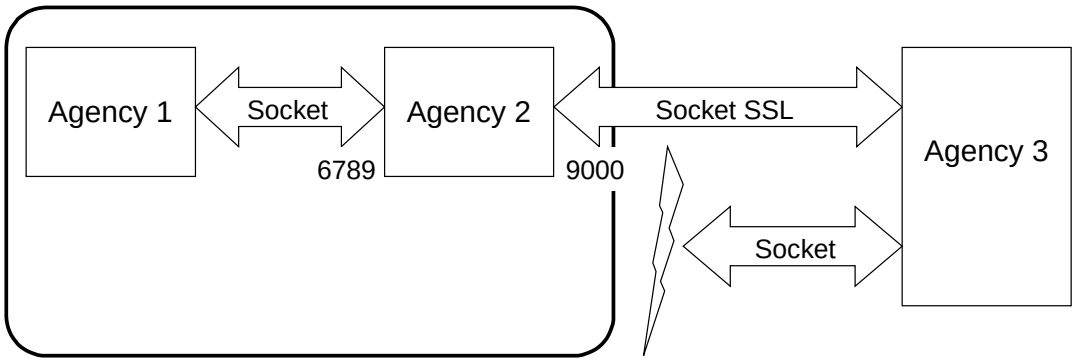


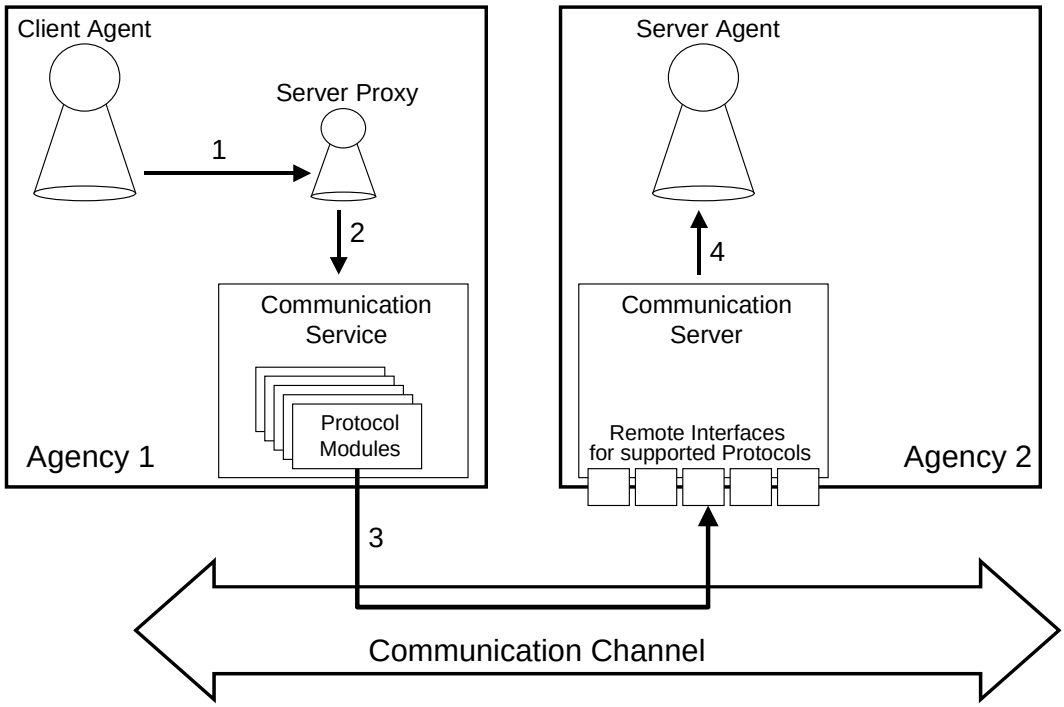
Figure 3: Secure vs. Insecure Protocols

In this example, agency 2 has started two communication receivers. The first one, a plain socket receiver, listens on port 6789 for incoming communication requests. This port is protected by the firewall, i.e. only hosts from inside the company can connect to this port. The second receiver uses the SSL-socket protocol and listens on port 9000 which is accessible to all Internet hosts world wide. Because the owner of agency 2 cannot be sure which Internet user will send an agent to its agency, Internet users are able to send agents with the SSL protocol only. This protocol provides the authentication of the client, so that the owner of agency 2 can decide whether the communication should be permitted or not. If, for example, an attacker tries to send an agent by using an insecure socket-protocol, the attempt fails at the firewall. On the other side, the agency 2 owner trusts all clients from inside the company. Thus, clients from the intranet can use the faster socket-protocol.

3.3.2 Location Transparency

On the one hand the communication service is used by the Grasshopper system, e.g. for agent transport or for locating entities within the DAE. On the other hand, agents can use the CS to invoke methods on other agents. This is done location-transparently, i.e. the agent need not care about the location of the desired communication peer. Within the agent code, remote method invocations look exactly like local method invocations on objects residing on the same Java Virtual Machine.

This is achieved by means of so-called *proxy objects* (or *stubs*) that are directly accessed by a client. The proxy object forwards the call via the ORB to the remote target object, i.e. the server. In this way, these proxy objects are equivalent to the client stubs used by CORBA implementations. Figure 4 shows this concept on a rather abstract level.



1, 2, 4 Local Java method invocation

3 Remote method invocation via one of the supported protocols

Figure 4: Location Transparent Communication

The usage of the local proxy object offers a simple API to agent programmers and hides all internal negotiations and requests. Please note that location-transparent communication requires a running region registry and works inside the same region only. Communication between agents running in the same agency is not constrained to this pre-condition, i.e. without a region

this is still possible. For communication with an object outside the client region, the client has to specify the target location in a URL syntax. This functionality is equivalent to the functionality of current CORBA implementations.

3.3.3 Communication Modes

In the context of Grasshopper, inter-agent communication may be performed in several modes. Grasshopper supports the following communication modes:

- synchronous communication
- asynchronous communication
- dynamic communication
- multicast communication

3.3.3.1 Synchronous Communication

Usually, when a client invokes a method on a server, the server executes the called method and returns the result to the client which then continues its work. This style is called *synchronous* because the client is blocked until the result of the method is sent back. Figure 5 illustrates this concept.

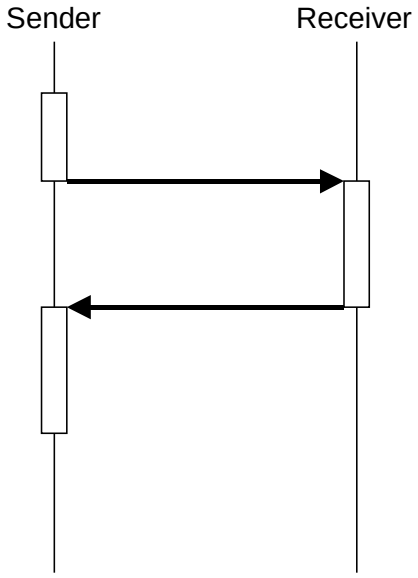


Figure 5: Synchronous Communication

3.3.3.2 Asynchronous Communication

When using asynchronous communication, the client does not have to wait for the server executing the method. Instead the client continues performing its own task. There are several possibilities for the client to get the result of the invoked method: It can periodically ask the server whether the method execution has been finished, wait for the result whenever it is required, or subscribe to be notified when the result is available. Figure 6 shows these three possibilities.

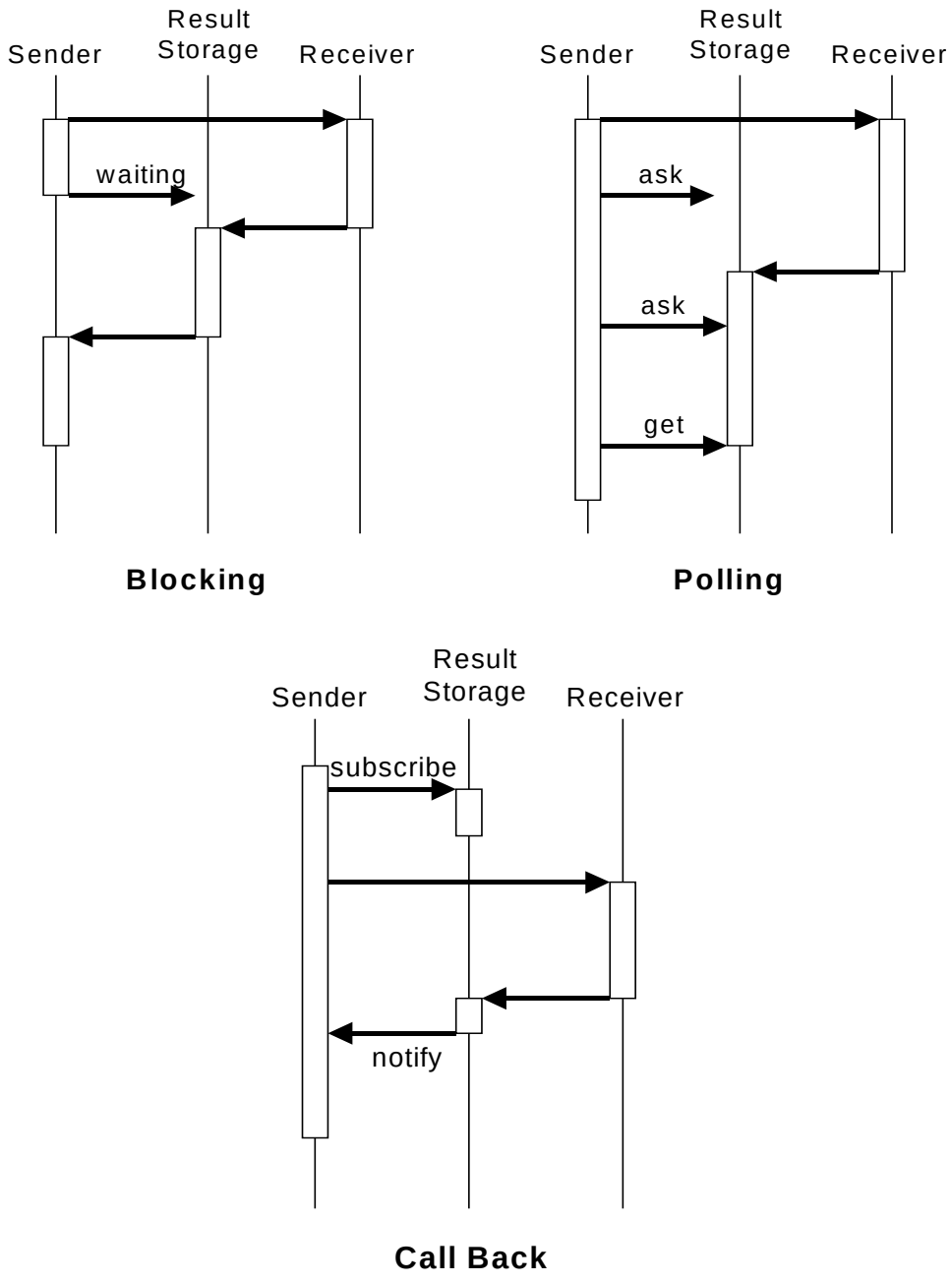


Figure 6: Asynchronous Communication

3.3.3.3 Dynamic Communication

This mechanism is useful if the client does not have access to a server proxy. The client is able to construct a message at runtime by specifying the signature of the server method that shall be invoked. Dynamic messaging can be used both synchronously and asynchronously.

3.3.3.4 Multicast Communication

Multicast communication enables clients to use parallelism when interacting with server objects. By using multicast communication, a client is able to invoke the same method on several servers in parallel.

Group proxies provide a framework for designing and implementing partitioned concurrent activities in Java. A *Group* consists of an arbitrary number of members. Group proxies maintain some kind of collection by enabling objects to *join* and *leave* groups dynamically. They also encapsulate the thread-based mechanisms needed to implement Java analogues of execution constructs found in concurrent, parallel languages. Internally, concurrent communication is implemented by the creation of parallel running threads where each of them performs a specific communication task. For this purpose, the group request is split and addressed to the appropriate server in the first phase, the so-called *scatter phase*. In the case that all the threads perform invocations without return values, only the scatter part firing up the tasks applies. But when actions must be synchronised or results must be collected, a co-termination policy is required. The policy depends on the semantics of the communication of interest. Three common policies are:

- OR Termination
- AND Termination
- Incremental Termination

A policy characterises how the results returned by the server objects are collected and how the group client can access these results. This phase is called *gather phase*. Figure 7 shows the roles involved in a multicast communication.

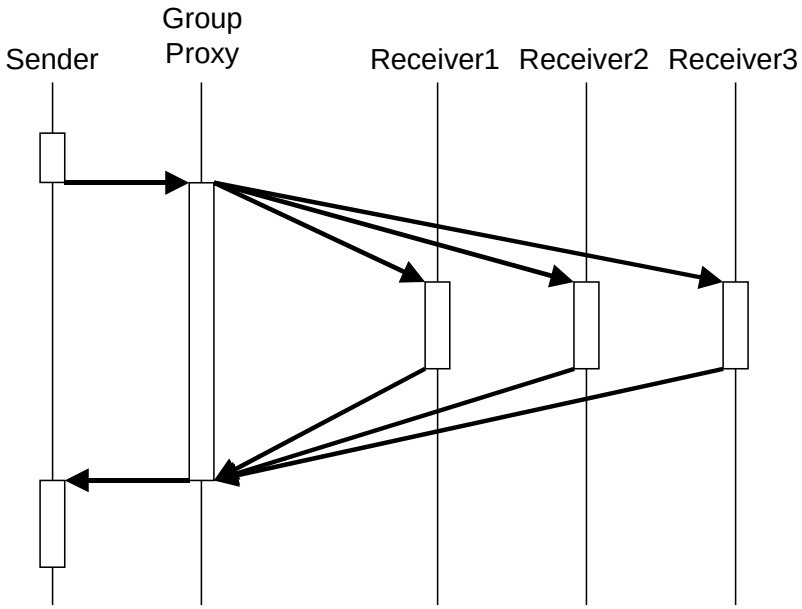


Figure 7: Multicast Communication

OR Termination

Figure 7 also depicts the gather policy problem of multicast communication: May the client continue its execution while the server methods are running? If not, at which point shall the client be restarted? The first possibility is to suspend the client until a result arrives, i.e. the client continues processing its task when any communication thread terminates. The client does not care about which server has sent the result. All results related to this communication session that arrive after the first one are discarded (cf. Figure 8).

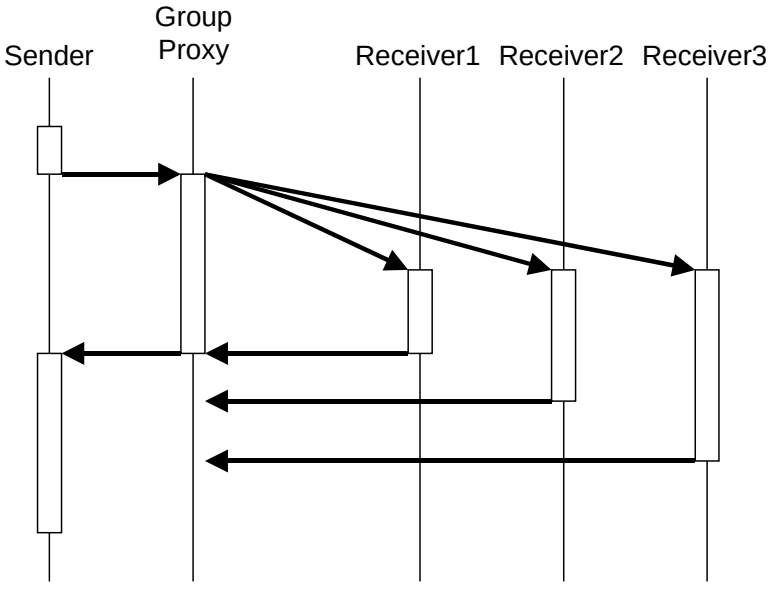


Figure 8: OR Termination

AND Termination

Another possibility is to suspend the client until all results have arrived. In this case, all communication tasks must be terminated before the client can continue its execution. A timeout duration specifies the maximum time for waiting. This avoids the death of the client in the case that a server does not send a result. After restart, the client can access the results sent by the group members.

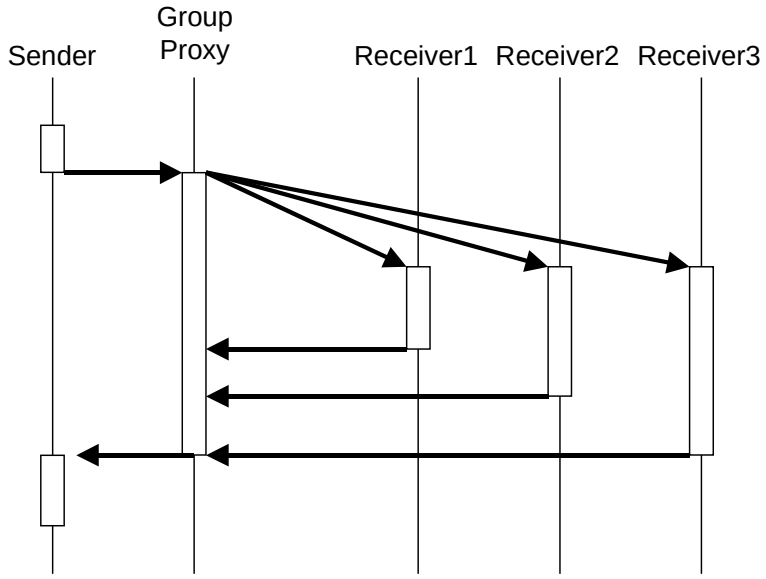


Figure 9: AND Termination

Incremental Termination

The policies described above suspend the client during the communication session. The communication service also supports a policy that allows the execution of the client immediately after sending a request to the group. Like the polling concept in asynchronous communication (cf. Figure 6), the client can check whether a reply from a specific server is available or not.

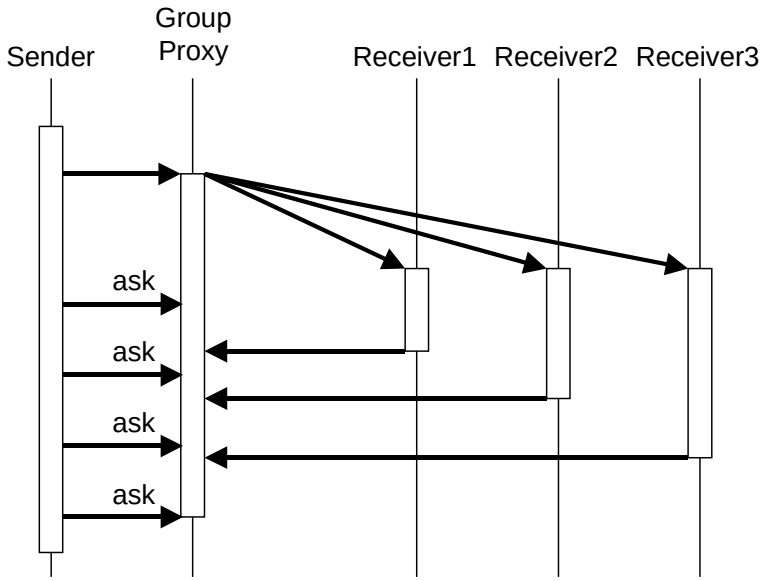


Figure 10: Incremental Termination

3.3.4 Grasshopper URL

Generally, a Grasshopper location is specified in a URL-like notation of the form

```
<protocol>://<host>:<port>/<agency>/<place>
```

`protocol`:	acronym of the desired protocol (cf. Table 3)
`host`:	name or IP address of the destination host
`port`:	number of the port at the destination site (optional)
`agencyName`:	name of the destination agency
`placeName`:	name of the destination place (optional)

Name	Acronym	Default Port Range
Plain socket	`socket`	7000 – 7020[1]
Java RMI	`rmi`	7050[2]

[1] Note that each socket communication receiver requires its own port on each host. If the default port is already in use, the port number is automatically incremented with 2.

Name	Acronym	Default Port Range
CORBA IIOP	`iiop`	depends on CORBA implementation
Plain socket with SSL	`socketssl`	8000 – 8020[1]
Java RMI with SSL	`rmissl`	8050[2]
MAFIIOP	`mafiiop`	depends on CORBA implementation

Table 3: Supported Communication Protocols

Because of RMI internal reasons it is not possible to run RMI and RMI-SSL interactions at the same time within the same agency.

The place name as well as the port number are optional. If no place is specified, the default place "InformationDesk" is contacted which exists within each agency. The default port number depends on the desired protocol (see Table 3). Note that the port number of CORBAIIOP and MAFIIOP depend on the used CORBA implementation. Thus, if one of these protocols shall be used, a port number must be specified.

The location of a region registry is specified in a corresponding way:

```
protocol://host:port/registryName
```

The `registryName` should be set to name of the region registry if the MASIF compliant parts of Grasshopper are used. Otherwise the `registryName` has no meaning.

The location specification explained above requires that the communication client knows which protocols are supported at the server (i.e. destination) site. If the client does not have this information, the meta-protocol `grasshopperiiop` may be used. In this case, the communication service automatically contacts the region registry in order to get information about the protocols that are supported at the destination site. Of course, these addi-

[2] This number specifies the port of the RMI registry within an agency. Note that more than one RMI or RMI-SSL receivers from several agencies can register to the same RMI registry. If the agency in which the RMI registry is running terminates, all registered RMI receivers lose their communication ability.

tional interactions are time consuming. Thus, if possible, a concrete protocol should be specified when constructing a location.

3.4 Security Concepts

This chapter introduces the basic concepts necessary to understand the security concepts of the Grasshopper platform.

In open distributed environments such as Grasshopper a number of serious security threats exist that must be considered when designing an effective security policy.

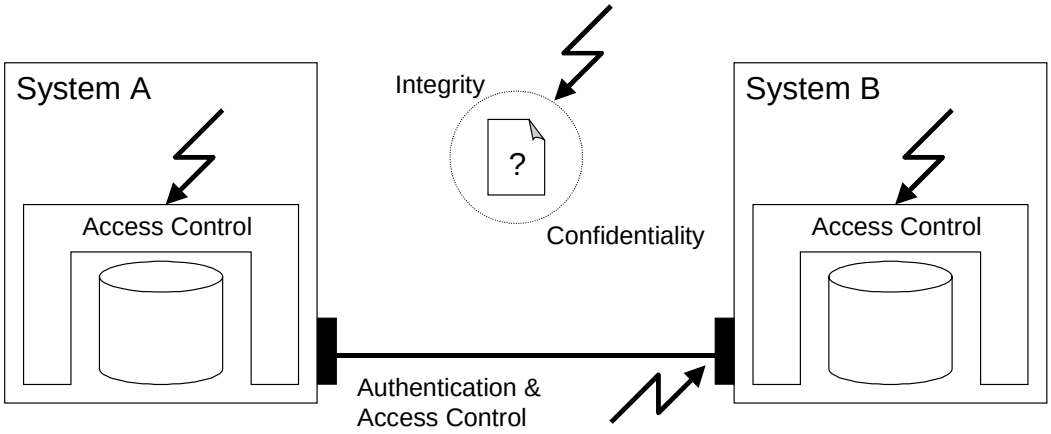


Figure 11: Potential Security Attacks

To address these threats, the Grasshopper security services must provide the following features:

- **Confidentiality**: When an agent transports confidential data, it should not be of any potential use for anybody else than the communication partners. Disclosure or eavesdropping of the data can be fatal. Since the transmission often takes place over communication media that cannot be physically secured or that are out of the control of the communicating partners, the transmitted agent must be encrypted while in transit. Thus it is made useless for anybody, except for someone who knows how to decrypt the agent (which should be only the designated server).

- **Integrity**: Upon reception, it must be obvious if the agent was modified or corrupted, be it by means of transmission errors or intentional acts of vandalism. An attacker must not be able to modify agents without the server noticing it. If in doubt, the server can try to reconstruct the agent or ask the client to repeat the transmission.

- **Authentication**: In a communication session, authenticity of the exchanged data requires authentication of the communication partners. If important data is transmitted, each communication party should be aware of the real identity of the communication peer. The nature of computer networks makes it much easier for an impostor to pretend to be someone else than in real life, where "authentication" is straightforward. Scenarios, where an attacker pretends to have the identity of someone else are called masquerading. Once successful, the attacker can easily pass further security checks. Therefore, authentication is one of the key requirements to securing distributed systems. Another attack that can be covered by an authentication service is the replay of previously intercepted data.

- **Access Control**: An agent system generally has access to a certain amount of resources, such as file system, network, CPU time, memory, etc. These resources must be protected from unauthorised access. Usually this is achieved through an access policy that grants access to system resources based upon different levels of trust. When an agent manages to get unauthorised access, it can destroy data or perform so-called denial-of-service attacks, where agent system resources are allocated again and again until the system is overloaded and breaks down.

- **Auditing**: In order to get an overview about security relevant events, the security services mentioned above, especially authentication and the access control service, should be able to keep track of whenever agents try to access system resources or the system itself, as well as authentication failures or when transmission faults occur. These events should be logged to a file so they can be analysed later.

3.4.1 Cryptography

Most scenarios where cryptography is deployed involve a sender and a receiver of a message. The sender wants to send the message securely and make sure that an eavesdropper cannot read it. Before it is sent, i.e. in its actual clear form, the message is plain text. The process of rendering the message unreadable, i.e. converting it to cipher text, is called encryption. The reverse procedure, i.e. converting cipher text into plain text, is called decryption. The term "plain text" should not be confused with "plain ASCII text". The plain text messages can have any form, e.g. a text file, a video stream, or, in the case of software agents, simply binary data.

A cryptographic algorithm, or cipher, is a mathematical function used for encryption and decryption. However, keeping the cipher secret would be

rather ineffective. Therefore, in addition to the algorithm, a key is used and kept secret. Both encryption and decryption use this key in combination with the cipher. Now the algorithm can be published, analysed and standardised. This can be compared to a door lock. Everybody can get the technical specifications of a door lock, but without the key this knowledge is useless.

3.4.1.1 Symmetric Algorithms

Symmetric algorithms are algorithms where the key used for decryption can be calculated from the key used for encryption and vice versa. Usually encryption and decryption key are the same. This presupposes that sender and receiver agreed on a key before securely exchanging messages. Here comes the problem inherent to symmetric algorithms. In a scenario, where sender and receiver are located relatively close, they can easily exchange the key physically prior to the actual communication, e.g. on a floppy disk. But in a larger networked application, this prerequisite is often not given. Key exchange must be done via other "more trusted" media like telephone or mail. Even worse, if the receiver does not know the sender, how can the receiver be sure of receiving the correct key? Furthermore, in a scenario with many secure point-to-point transmissions, the number of keys grows exponentially, thus adding a vast overhead to the application.

3.4.1.2 Asymmetric Algorithms

Asymmetric or public-key algorithms use two different keys for encryption and decryption: a public and a private key. For each communication partner there exists such a key pair, where the private key is kept secret while the public key is made available to the public, e.g. through a directory service or public-key certificates. This requires that the private key cannot be calculated from the public key, at least not in a reasonable amount of time. The keys act as a sort of one-way lock, i.e. once a message is encrypted using the public key of a person, it can only be decrypted by the corresponding private key. Therefore, only that person can convert the cipher text back to plain text (presumed no one else knows the private key). That way, the problem of key distribution can be solved and the overhead is reduced since public and private key can be used several times.

As a matter of fact, public-key algorithms are generally a thousand times slower than symmetric algorithms, so in real-life applications a combination of both techniques, called hybrid crypto-systems is often used: public-key algorithms are used first to exchange a symmetric session key, which is then used to encrypt and decrypt the actual message.

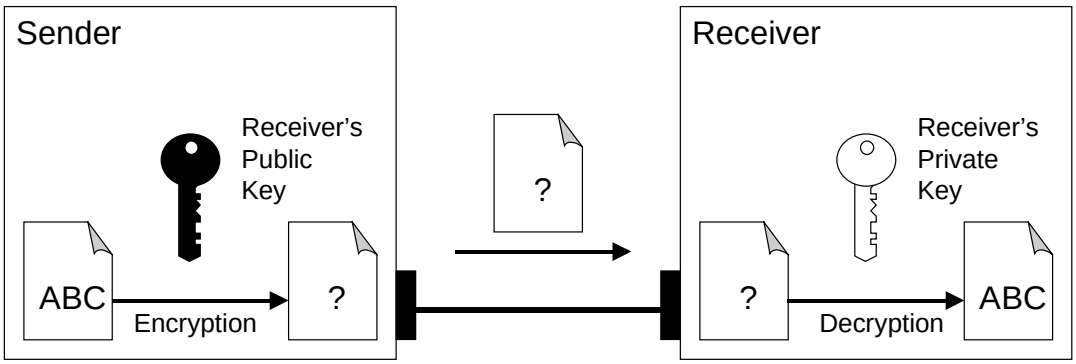


Figure 12: Providing Confidentiality Using Public Key Algorithms

3.4.1.3 One-Way Hash Functions

One-way hash functions, often also called digital fingerprint, message digest or message checksum, are central to modern crypto-systems. They operate on a message and produce an output of fixed length. Their importance for cryptography is based upon the following common characteristics:

- Given a message M, it is relatively easy to compute h(M).
- Given h(M), it is hard to compute M. Hard means mathematically infeasible in a reasonable amount of time.
- Given M, it is hard to find another message M', such that h(M) = h(M').

Therefore it can be determined when a message was modified or corrupted during transmission.

3.4.2 Authentication

Not only privacy and integrity can be ensured through the use of crypto-systems, but also authentication. If sender and receiver share a common secret, only known to them, the secret acts as the authentication information, as a proof of the other communication party's identity. In the case of symmetric algorithms, this would be the session key. However, much more popular is the use of public-key algorithms to ensure authentication. If the sender wants to prove that it is really it who sends the agent, it digitally signs it prior to transmission. In most cases this denotes the act of encrypting the agent with the private key. Now anyone who receives the agent but is not sure if it really comes from the actual sender can decrypt it with the sender's public key. If the decryption succeeds, the receiver can be sure that only the sender could have encrypted it. This is called verifying a digital signature.

In practical implementations, public-key algorithms are not used to sign the whole agent. Instead, it is far more efficient to generate a one-way hash of the agent together with a timestamp, sign them and transmit them separately. The receiver again generates a hash from the agent and compares it with the received hash.

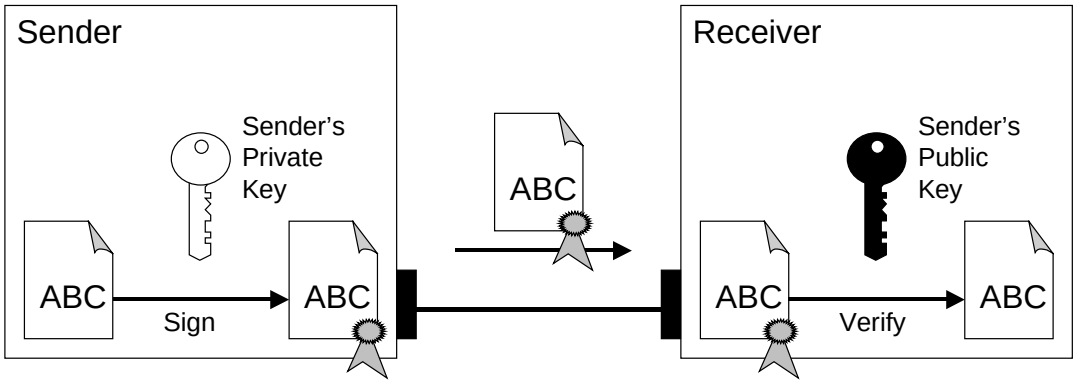


Figure 13: Providing Authentication Using Public Key Algorithms

3.4.3 X.509 Certificates

The ITU-T authentication framework introduces the notion of public-key certificates, data structures for storing and exchanging public keys. A trusted third party, called certification authority (CA), assigns a unique distinguished name to each user and issues a digitally signed certificate, consisting of:

- *Version number*: Identifier for the certificate format.
- *Serial number*: Unique to each user of a CA.
- *Algorithm identifier*: Identifies the algorithm used to sign the certificate.
- *Issuer*: The distinguished name of the CA.
- *Period of validity*: A time period, consisting of a Not-Before and Not-After date, between the certificate is valid.
- *Subject*: The distinguished name of the user.
- *Subject's public key*: The public key of the user.
- *Signature*: The signature of the CA.

By signing the certificate, the certification authority states that the personal data, contained in the distinguished name of the subject, corresponds to the

public key. Now, what happens during an authentication scenario using X.509 certificates? Prior to the communication, the receiver gets the sender's certificate from a public database and verifies the signature. If both sender and receiver share the same certification authority, this is trivial. The receiver knows that the sender certificate is issued by a trusted instance and therefore trusts the certificate itself. If not, it is more complicated. The receiver can either trust the sender's CA anyway or check if it is certified by yet other certification authorities. Thus, a tree-like structure of trust is built. CAs certified by higher-level CAs can again certify other CAs. The receiver can walk up the tree until reaching a certification authority that is well-known and trusted.

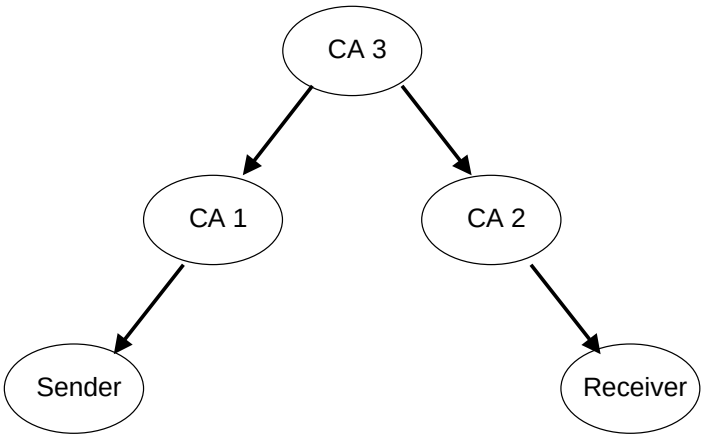


Figure 14: An X.509 Certification Tree

3.4.4 Access Control

Access Control denotes the process of controlling access of an entity to a system or to resources within that system. If access to a specific resource is denied, it is the task of auditing to log this for further evaluation. Access control is closely linked to authentication, i.e. the actual access decision is often based upon the identity of the entity performing the access. Within the system where the access occurs, the initiator's identity can be associated with a set of access rights. Therefore, if an attacker succeeds in masquerading as another user, it obtains the access permissions granted to that user. In this case, the attacker has unauthorised access to system resources and can subsequently invoke denial-of-service attacks or destroy data.

3.4.4.1 Protection of Resources

The resources of a distributed system that must be protected include:

- *CPU time*: In a multi-user and multitasking operating system, a single user or process must not be able to get hold of the complete CPU, otherwise the work of other users or processes may be disturbed.

- *Memory*: The amount of allocated memory to each process should be limited. Furthermore, the address spaces of different processes must be protected from one another.

- *File system*: Obviously, in multi-user environments, such as UNIX, files must be protected from unauthorised access.

- *Networking capabilities*: Since networking resources such as sockets represent the connection of a system to the outside world, they are vulnerable and subject to access control.

3.4.4.2 Access Control Policies

Access control policies represent the security requirements in a certain security domain. They usually consist of a set of rules acted upon by Access Control Decision Functions whenever a check is performed. There can be different categories of access control policies, as identified by ITU Recommendation X.800. A rule-based access control policy applies to all users of a system, regardless of any privileges or access levels. In contrast, identity-based access control policies assign different access rights to different identities. Of course this presupposes proper authentication. Identity-based access control policies are often implemented using Access Control Lists (ACL), where an identity is associated with a set of permissions it is granted. Two particular types of identity-based policies are group-based and role-based access control policies. In the former, the respective permissions are valid for a group of people, e.g. an organisation or enterprise, while in the latter, access rights are assigned to specific roles that can be held by different users at different times, e.g. database administrator or super user. Furthermore, one can distinguish between administratively imposed or dynamically selected access control policies.

3.4.5 Security in Grasshopper

This section describes the realisation of a Grasshopper Security Service intended to meet the requirements stated above.

3.4.5.1 External Security

To provide the required security, Grasshopper makes use of X.509 certificates (see above) and the Secure Sockets Layer (SSL). SSL is an industry-standard protocol that makes substantial use of both symmetric and asymmetric cryptography. Widely deployed in client-server products of leading vendors, including Netscape, Microsoft and IBM, SSL provides confidentiality, data integrity and mutual authentication of client and server.

- **Confidentiality**: All communication between client and server will be handled over a secure socket, encrypted with a symmetric key and an encryption algorithm negotiated in a handshake prior to the actual SSL session. Although the IP packets can still be intercepted, encryption renders them useless for eavesdroppers. Currently, Grasshopper is able to use any available encryption algorithm.

- **Integrity**: Message Authentication Codes (MACs) can prove that a message was not modified during transport, be it by vandals or transmission errors. These MACs are calculated for each SSL packet using hash functions.

- **Authentication**: The purpose of authentication is that both communication parties convince each other of their identity. During the SSL handshake, client and server exchange personal data and their public keys packaged together in the form of X.509 certificates. The authentication process requires both parties to digitally sign protocol data with their private keys. The certificate itself does not authenticate, but the combination of certificate and correct private key does. Currently, Grasshopper is able to use every available X.509 compliant certificate type.

So what are the prerequisites for doing secure communication? Each agency should have at least a personal certificate and a private key. The key is required for signing data during the SSL handshake (512 bit). This data is hold in the Grasshopper security context that is loaded and initialised at start-up.

Now, how can agencies specify their desire to communicate securely? The receiver agency does so by running only secure servers (Socket/SSL or RMI/SSL), which are configured in the communication preferences, while the sender/client can selecting a appropriate protocol if it wants to communicate securely or not. If both agencies specified the same, everything is fine. If the sender specified secure communication, but the receiver is only running insecure servers, the communication fails and a security exception is thrown on the sender side. If the sender wants to communicate insecurely, but the receiver has only secure servers, this fails, too.

What happens next? In the SSL handshake, sender and receiver exchange their personal certificates containing the public key. When the Grasshopper agency receives a chain of certificates (this includes a single certificate where the chain has length 1), it verifies the consistency of the chain, checks if each certificate is within its validity period and searches in its database of known signers, to check whether the subject belonging to the certificate is known. If yes, the SSL session is handled as specified in the associated behaviour (see above). If it is not known, the next certificate is taken from the chain and examined the same way. If the algorithm reaches the end of the chain without encountering a known certificate, the connection is rejected, if not configured otherwise.

If this step is passed on both sides, sender and receiver agency mutually authenticate each other by signing random data with the private key and sending it to the peer, who verifies the signature with the public key. After this, the SSL handshake has been successfully completed, and the secure channel established, meaning that sender and receiver are authenticated and a session key was exchanged. Subsequently, all data transmitted through the socket is encrypted and decrypted using this key and secured by MACs, thus providing confidentiality and integrity for the remaining session.

But what does this mean? A secure session between two agencies is established. This session can be cached and reused later. In this context, it is of vital importance to understand the fact that it is not agents which are authenticated but the agency administrator, or at least the person who specified the personal certificate used for authentication. Different agents from different owners and creators can travel through the same authenticated secure socket. However, in typical applications this can be considered as a minor drawback.

3.4.5.2 Internal Security

Internal security is all about protecting resources of the agency from unauthorised access by agents. Furthermore, it is useful to protect agents from one another. But why is access control so important? Well, think of malevolent agents as viruses. If you do not protect yourself against them, they could destroy data, shut down your agency, kill other agents and so on.

Regarding access control, Grasshopper is strongly oriented towards the security mechanisms of JDK 1.2. It makes use of an codesource-based and identity-based access control policy, which is initialised at start-up. In Grasshopper, an access control policy is an access control list comprising several entries, one for each subject treated in this policy, where a subject can be a codebase or a principal. With each subject, there is associated a set of permissions, granting access to all important parts of the Grasshopper agency.

Note that it is not possible to explicitly deny permissions. Only "positive" permissions are allowed.

You might ask, how the agent is linked to the name of the subject in the policy? When an agent tries to make a system access, e.g. by opening a file using the `java.io` package, an access controller is consulted to make the access decision. In fact, each time a system access happens the access controller is invoked, but it is capable of distinguishing whether the access was made by an agent or by trusted system code, e.g. the Grasshopper core. If the access came from an agent, the access controller extracts the agent's origin and owner from the agent itself. With this information, it contacts the Access Controller to extract the set of permissions valid for this subject. It is then checked if the permission to perform the access is contained in the set of permissions granted to the subject. If not, an access control exception is thrown.

It follows that effective access control presupposes authentication and secure transmission of the agent. Why is this so? When an agent from a remote agency makes a system access, the agent's signature is taken as a token to retrieve the proper permissions from your local access control policy. This signature is set at the agent's creation time in the remote agency, and it must be ensured that it was not modified during agent transport to your agency. Otherwise, for example, some attacker could masquerade as a friend to whom you have granted read/write access to your home directory.

Grasshopper uses the notion of protection domains, first introduced in JDK 1.2. Each agent is loaded by a class loader associated to a protection domain, thus properly separating different security domains. Classes from your `CLASSPATH` variable are considered as system classes and therefore trusted. They are not subject to any permission checks and therefore run outside the "sandbox". It is of utmost importance to always keep this in mind. If you let someone tamper with your `CLASSPATH` variable, the access control mechanisms can be tricked. Only agent classes loaded from a codebase, such as `file://` or `http://` , are subject to access control.

When programming your own agents that perform access to resources, please take care to enclose the code where the access happens in try-catch blocks in order to catch the access control exception if the necessary permission is not granted. If, for example, for performance optimisation, access control is not desired it can be disabled via a command-line option.

3.5 Persistence

Persistence is a very important topic regarding distributed applications. Objects are sent from one computer to another and often have a long life span. That is especially true for mobile agents. The following undesirable scenarios have to be taken into account:

- An agent moves from one agency to another. The transmission fails for some reason so that the agent never arrives at its destination.
- An agent is residing within an agency whose host computer crashes or shuts down unexpectedly (e.g. due to a power failure).
- There are many agents residing within an agency, with most of them waiting for external events without performing any task, thus just wasting system resources. The host computer could run out of resources (especially memory) if more agents want to migrate into that agency.

While the first scenario can be avoided by buffering the agent until the arrival has been confirmed, the remaining two need another approach. A copy of the agent object has to be maintained on a durable (i.e. persistent) medium, e.g. a hard disk. If the agency system crashes, persistent agents can be reloaded from this medium after the agency has been restarted (keyword: saving). Besides, idle agents (i.e. agents just waiting for an event without executing any task), need not remain instantiated. Instead, they could be stored permanently and then removed from the agency's RAM in order to save resources (keyword: flushing). If a request for a flushed agent arrives, the agent can be re-instantiated in order to handle the request.

Grasshopper handles all the topics mentioned above if persistence is enabled.

4 Frequently Asked Questions

This chapter contains answers for some Frequently Asked Questions. The chapter is organised to help the Grasshopper user to find answers in different usage areas. Note that additional information is provided in the Glossary (cf. Annex A).

4.1 Mobility

How can agents move?

Grasshopper agents are implemented in Java. Thus, their code can be executed on every host that supports Java - independent of the hardware architecture. Grasshopper agents can either use the Grasshopper-specific communication service or optionally a CORBA 2.0 compliant ORB for MASIF conformant migration. To move an agent to another agency residing on the same or another host can be initiated by invoking the `move()` method of an agent. This method can be invoked by the moving agent itself or even by other entities. A detailed description can be found in the Programmer's Guide.

4.2 Communication

How does inter-agent communication work?

Grasshopper agents can invoke methods of other agents in a location-transparent way. Therefore, agents need not know the actual location of their communication peer. For the agent programmer, a method invocation of a remote object can be implemented like a method invocation of an object residing in the same Java Virtual Machine (JVM). Grasshopper supports synchronous, asynchronous, and multicast communication.

What communication protocols are supported?

The Grasshopper communication service supports the following protocols:

- CORBA IIOP
- Java Remote Method Invocation (RMI)

- Plain Socket Protocol
- RMI with SSL
- Plain Socket Protocol with SSL

4.3 Security

Questions associated with Grasshopper security are separated into the following sections:

- Certificates and Encryption (Section 4.3.1)
- Permissions and Access Control (Section 4.3.2)

4.3.1 Certificates and Encryption

Does Grasshopper support strong cryptography?

Yes. Communication in Grasshopper can be secured using the Secure Socket Layer (SSL) protocol. By default, 1024 bit RSA keys are used for key exchange and authentication and 128 bit RC4 keys are used for encryption and decryption of the actual data.

What's the deal with U.S. export restrictions and patents?

The U.S. government restricts the use of U.S. products which secure their data with strong cryptography to U.S. citizens only. Export versions of U.S. products using cryptography are constrained to weak cryptography. Grasshopper uses an Austrian toolkit named IAIK that supports both weak and strong cryptography. RSA and RC4 are patented by RSA Security Corp. Please keep this in mind when using Grasshopper in the U.S.

What are symmetric key algorithms?

In symmetric key algorithms, the keys for encrypting and decrypting the data are identical. Although much faster than asymmetric key algorithms, they raise the problem of exchanging the key prior to a communication session. The most prominent members of this family are DES and RC2/RC4.

What are asymmetric key algorithms?

Invented in 1976 by Diffie and Hellman, asymmetric or public key algorithms solve the problem of key distribution by having a key pair: private and public key. The public key can be published, while the private key should be kept secret. Once data is encrypted with the public key of a person, it can be only decrypted with the corresponding private key. Although overcoming the key exchange problem, asymmetric key algorithms are much slower than symmetric. A well-known example is RSA.

What is RSA?

Named after its inventors, Rivest, Shamir and Adleman, the RSA public key algorithm works for both encryption and digital signatures. It is based upon the difficulty of factoring very large numbers. Although RSA's security has been never fully proven by crypt-analysts, it is widely used and very popular.

Suppose Alice wants to send Bob a message, then there are two ways of using RSA:

- For encrypting data: Alice takes Bob's public key and encrypts the data. Now, only Bob can decrypt the data with his private key.
- For digitally signing data: Alice takes her private key and encrypts the data. When Bob gets the message and succeeds in decrypting it with Alice's public key, he can be sure that it could only have been encrypted by Alice, given the fact that only Alice knows her private key. Note that everybody else can decrypt the message with Alice's public key and read it.

In real life scenarios, a combination of both techniques is often used.

What is RC4?

RC4 is a variable-key-size symmetric cipher developed in 1987 by Ron Rivest for RSA Data Security, Inc. For several years it was proprietary until the source code was posted anonymously to a mailing list and quickly spread throughout the Internet. It is about 10 times faster than the popular DES (Data Encryption Standard) cipher.

What is SSL?

The Secure Sockets Layer (SSL) is an industry-standard protocol that makes substantial use of both asymmetric and symmetric cryptography. It delegates encryption and authentication procedures from the application to the transport layer. Widely deployed in client-server products of leading vendors in-

cluding Netscape, Microsoft and IBM, SSL provides privacy, data integrity and mutual authentication of client and server.

- *Privacy*: All communication between client and server goes over a secure socket, encrypted with a symmetric key and an encryption algorithm negotiated in a handshake prior to the actual SSL session. Although the IP packets can still be intercepted, encryption renders them useless for eavesdroppers.

- *Integrity*: Message Authentication Codes (MACs) can prove that a message was not modified during transport, be it by vandals or transmission errors. These MACs are calculated for each SSL packet using hash functions.

- *Authentication*: The purpose of authentication is that both communication parties convince each other of their identity. During the SSL handshake, client and server exchange personal data and their public keys packaged together in the form of X.509 certificates. The authentication process requires both parties to digitally sign protocol data with their private keys. The certificate itself does not authenticate, but the combination of certificate and correct private key does.

What is a X.509 certificate?

A X.509 certificate is a data structure containing personal data about an entity (person or organisation), such as name, e-mail address, location, country and the public key. Furthermore it contains a digital signature of an issuing entity, testifying that the certificate data belongs to that entity. Usually, certificates are issued by so-called certification authorities (CAs) and are valid only for a certain time. Thus, certificates act as a digital version of an ID, passport or driver's license. The combination of certificate and corresponding private key (which is NOT inside the certificate) authenticates you to someone who needs proof of your identity.

What is a certificate chain?

Certificates can build a tree-like structure, where each node certificate issued its child certificates. In an authentication process, usually the verifier gets a chain of certificates. If he does not trust a leaf certificate, he goes up the tree until he finds a well-known certificate. The so-established trust model is transitive, i.e. if there is one certificate that is considered trusted, all child and grandchild certificates are, too.

What is a certification authority (CA)?

A certification authority issues certificates to users, thus guaranteeing that the personal data and the public key in that certificate belong together. A CA usually has the reputation of being a trustworthy entity, so it acts as a trusted third party in authentication processes.

What is a self-signed certificate?

A certificate chain normally ends at a top-level CA certificate. This certificate is self-signed since it has no ancestors.

How does authentication work in Grasshopper?

Each Grasshopper agency has a list of certificates which are associated to a certain behaviour. If, in an SSL handshake, the agency receives a chain of certificate, this chain is traversed from bottom to top using the following algorithm:

1. Take the current certificate from the received chain.

2. Check if the certificate was signed by the next one in the chain.

3. Check if the certificate is already/still valid.

4. Check if the certificate owner is found in the list of known signers.

5. If yes, accept, reject or ask user, based upon the specified behaviour.

6. If not, take next certificate and go back to step 2.

7. If last certificate is reached, reject automatically or ask user about unknown top-level certificate. The user can accept or reject it. It may optionally be saved in the database of known signer certificates.

Who is authenticated?

Since SSL sessions are established between agencies, the agency owner or administrator is authenticated with his certificate. He can differ from the person who wrote the agent code or instantiated the agent. This is an important fact to keep in mind!

Does Grasshopper support X.509v3 certificates?

Yes.

How can I get a personal certificate chain?

You can use the standard keytool shipped with the JDK.

Can I have more than one personal certificate chain?

Yes. In your private keystore you can import several personal certificate chain. However, in authentication processes during SSL handshakes, only one of these chains is taken.

Where are Grasshopper security relevant files stored?

Grasshopper uses your private keystore and Java policy file. They are stored in your home directory by default.

- `.java.policy`: The access control policy for Grasshopper (see below).
- `.keystore`: All private keys and certificates.

Are private keys stored password encrypted?

Yes. They are packaged together with a personal certificate chain and can be exported password protected.

4.3.2 Permissions and Access Control

Why is access control important?

Without access control, an agent executing in your agency can easily change or destroy your files, influence agency system threads, establish network connections, shutdown the agency. Therefore, access to these system resources must be protected.

What is an access control policy?

An access control policy is a database or a file with different entries containing a subject and a set of associated permissions. E.g. it could contain an entry granting user Bob access to the file `/tmp/foo`.

Who is identified by the subject in the policy?

When an agent tries to access a resource, it does so on behalf of its user. Upon creation, each agent is associated with an owner, currently one certifi-

cate of the user who started the agency, i.e. the agency owner or administrator. So if user Bob starts agency A and creates an agent that travels to agency B and writes a file to the remote file system, there should be an entry in the policy file of agency B granting Bob permission to write to that specific file.

When is access control effective?

Although access control should also work when SSL is disabled, it is important to keep in mind, that an effective access control presupposes authentication, confidentiality and data integrity. To illustrate this, suppose you have granted Alice the right to access every single file in your file system. When an agent from Alice enters your agency, it must be ensured that it is really Alice who sent you the agent. Otherwise an impostor could masquerade as Alice and delete your home directory!

What is a permission?

A permission is associated with a user or subject and usually contains three entries:

- *type*: the type of the permission
- *target*: the resource that is to be accessed
- *actions*: the actions that are invoked on this resource, e.g. read/write

What is a protection domain and what do class loaders have to do with it?

To provide a sort of "sandbox", each agent in Grasshopper has its own class loader. This distinguishes agent classes from system classes loaded from CLASSPATH. When the access controller is invoked, it examines the current execution context and checks for class loaders. If no class loader is found, only system classes are in the stack, and the access controller returns quietly. If there are class loaders, access from within another protection domain is performed. A protection domain is a construct that can be used to encapsulate and groups different set of permissions. For example, there can be a system domain which has unlimited access, while a domain representing all agents of Bob has restricted access.
The algorithm for the access decision is as follows:

- Iterate through each class loader, and determine its associated protection domain.
- Check if the necessary permission to complete the access is contained in the protection domain’s set of permissions.

- If there is one protection domain in the stack that does not have the necessary permissions, throw an AccessControlException.

What's the deal with `$CLASSPATH`?

Classes from the system's CLASSPATH are considered trustworthy in Grasshopper, i.e. they have access to all resources. It is very important to always be aware of this fact. An agent executed from your CLASSPATH has system privileges and can do everything, from deleting your files to shutting down your agency or deleting other agents.

What should I care about when writing own code?

When you write agents accessing system resources, please be sure to enclose the code parts performing the access in a try-catch block, in case the access is denied.

How can I use the access controller within my own code?

The Grasshopper access control policy can be extended in the form of introducing new permissions, which can be checked in your own code. Suppose you write an agent offering services to other agents. While service A should be accessible by all agents, service B should only be accessible by agents from Alice or Bob. You can easily define a security permission "access.B" and grant Alice and Bob this permission in your policy. At the beginning of the method(s) realising service B, you invoke

```
AccessController.checkPermission(new
  de.ikv.grasshopper.security.Se-
  curityPermission("access.B"))
```

Now, only agents from Alice or Bob can pass beyond this point.

4.4 Installation

On what platforms Grasshopper is available?

Grasshopper is implemented entirely in Java. It can run on every platform that supports JDK 1.2 or higher. The Grasshopper package is delivered with installation procedures developed for UNIX and Windows 95/98/NT/2000. Since version 2.1 there is additionally a PersonalJava version.

What additional software packages are required to run Grasshopper ?

- Java Development Kit (JDK) or Runtime Environment 1.2 or higher.
- optional: security packages (e.g. IAIK iSaSiLk 3.0).
- optional: CORBA 2.0 compliant runtime environment. Without this environment, no IIOP communication is possible.

Where I can get these additional packages ?

- JDK 1.2 or higher from JavaSoft: http://www.javasoft.com/
- optional: the Java security package IAIK iSaSiLk 3.0 from the Technical University of Graz: http://www.iaik.at/

How is the installation process started ?

Grasshopper is delivered by means of a self-extracting file. For starting the installation procedure, please run the downloaded script (UNIX users) or the executable (Windows users). For PersonalJava you will have to copy the cab file to your device and start it afterwards.

Can I run Grasshopper on other operating systems than Solaris, like Linux or BSD Unix ?

Yes, but Grasshopper has been tested only on Solaris. When you use a UNIX-like operating system, you should try the Solaris distribution.

4.5 Platform Usage

How can I start an agency?

Use the start-up assistant `bin/Grasshopper` without any argument ans select *Agency* as the system type to start. Try the `-h` option to get more help.

How can I start a region registry?

Use the start-up assistant running `bin/Grasshopper` without any argument and select *Region* as the system type to start. Try the `-h` option to get more help.

Is it necessary to start a region registry?

Grasshopper 2 agencies can also be executed without a running region registry. This does not influence the creation and transport of agents. But without a region registry location-transparent communication is not possible, i.e. you have to explicitly specify a complete URL, including the desired protocol as well as host name and port number of the communication peer.

What is the meta-protocol `grasshopperiiop`?

By using this meta-protocol when specifying a location (URL), the Grasshopper system will automatically determine the communication protocol. Note that this is only possible if a region registry is available.

Does every agency need a separate region registry ?

No, each new region registry establishes a new region. If a region registry is still running in your environment, specify the host name and port number of this registry when starting a new agency.

What is the agent catalogue ?

The agent catalogue offers a user-friendly agent creation tool. It is similar to `Bookmarks/Favourites` known from Web browsers. After an agent is created for the first time, the code base and the class of the agent can be stored in the catalogue. When a user wants to create the agent again, he/she only has to select the corresponding catalogue entry.

Where can I find example agents ?

Grasshopper comprises several example agents in order to allow the user to verify the correct installation of the platform.
The source code of documented example agents is included in the `examples` directory of the Grasshopper installation. By using the agent catalogue, one can run these examples in an easy way. Additionally, several unsupported examples can be found on the Grasshopper web-site.

A Acronyms

CGI	Common Gateway Interface
CORBA	Common Object Request Broker Architecture
CS	Communication Service
DAE	Distributed Agent Environment
DPE	Distributed Processing Environment
IIOP	Internet Inter-ORB Protocol
ISO	International Standardisation Organisation
JDK	Java Development Kit
JFC	Java Foundation Classes
JRE	Java Runtime Environment
MA	Mobile Agent
MASIF	Mobile Agent System Interoperability Facility
OMG	Object Management Group
ORB	Object Request Broker
OS	Operating System
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SSL	Secure Socket Layer
VM	Virtual Machine

B Glossary

activation

the procedure in which the → *deactivation* of an → *service* is revoked. After its activation, a service is instantiated again and continues its task execution. Regarding the implementation, the object(s) associated with the service are created anew, and the corresponding Java thread is activated.

active

one possible → *state* of a → *service* that results out of the service's → *usage state*. A service is active if it is currently executing its task, i.e. if the corresponding Java thread is active. The other resulting state values are → *suspended* and → *deactivated*.

agency

the runtime environment for mobile and stationary → *agent*s. Each agency runs on its own Java virtual machine. An agency consists of a → *core agency* and one or more → *place*s. Agents execute their tasks within a place, and → *mobile agent*s are able to migrate from one place to another. A set of agencies can be grouped into a → *region*. Associated with each agency is a unique → *agency identifier*. In the context of Grasshopper, agencies are regarded as → *service*s.

agency identifier

enables the unique identification of an → *agency*. It is generated automatically during the creation of an agency and remains valid for the agency's entire lifetime. For information about its structure, please refer to → identifier.

agent

a self-contained software element which is responsible for autonomously carrying out one or multiple tasks. An agent acts actively in a → *distributed agent environment* on behalf of a human user or other software components. Associated with each agent is a unique → *agent identifier*. In general, agents run within a → *place* of an → *agency*. In the context of Grasshopper, two kinds of agents are distinguished: → *mobile agent*s are able to move actively from one place to another, whereas → *stationary agent*s can only be moved to another place by human users. The active movement of a mobile agent is called → *migration*. Grasshopper agents are realised as Java threads. In the context of Grasshopper, agents are regarded as → *service*s.

agent class

A Grasshopper agent consists of one or more Java classes. One of these classes builds the core of the agent and is referred to as agent class. Among others, the agent class contains the method → *live*() which specifies the actual task of the agent.

agent identifier

enables the unique identification of an → *agent*. It is generated automatically during the creation of an agent and remains valid for the agent's entire lifetime. For information about its structure, please refer to → identifier.

agent state

the mode of existence of an → *agent*. This mode is specified by means of the → *state* attribute of the agent. An agent can be → *active*, → *suspended*, or → *deactivated*.

asynchronous communication

a communication mechanism between clients and servers. After invoking a method of the server, the client does not have to wait for the server executing the method. Instead the client continues performing its own task. There are several possibilities for the client to get the result of the invoked method: It can periodically ask the server whether the method execution has been finished, wait for the result whenever it is required, or subscribe to be notified when the result is available. The counterpart is → *synchronous communication*.

autonomy

one of the most fundamental characteristics of an → *agent*. Once activated by a human user or another software component, an agent is able to execute its task more or less independently, after this returning the demanded results to its initiator.

clone

To clone an → *agent* means to create an exact copy of an already existing agent instance, i.e. to create a *clone* of this agent. This clone comprises the same internal information, e.g. the same → execution state, and thus it starts its task execution exactly at the point that the original agent instance had reached when the clone was created. Each agent is able to create a clone of itself. Note that the clone is always created within the same → *place* in which the original agent is currently executing. (see also → copy)

Common Object Request Broker Architecture

the Common Object Request Broker Architecture of the Object Man-

agement Group (OMG). CORBA is probably the best known architecture for → *object request broker* s. For further information, please refer to `http://www.omg.org`.

communication service

a → *core service* provided by each Grasshopper → *agency*. The communication service enables → *agent*s to communicate with each other in a location-transparent way. Currently, the communication service supports synchronous and asynchronous method invocations.

copy

Each → *agent* is able to create a copy of itself. This copy comprises the same internal information, e.g. the same → *execution state*, and thus it starts its task execution exactly at the point that the original agent instance had reached when the copy was created. Note that, in contrast to a → *clone*, a copy need not be created in the same → *place* in which the original agent is currently residing. It can be created even in remote places.

CORBA

cf. → *Common Object Request Broker Architecture*

core agency

the part of an → *agency* that comprises the functionality which is inevitably required for the execution and management of → *agent*s. Apart from this core, each agency comprises one or more → *place*s.

core service

the services that are comprised by the → *core agency*. These services realise the functionality which is inevitably required for the execution and management of → *agent*s.

DAE

cf. → *distributed agent environment*

deactivated

one possible → *state* of a → *service* that results out of the service's → *usage state*. If a service is deactivated, it does not exist anymore in form of a real "living" object. Instead the service, i.e. its code and (if the concrete service is a → *mobile agent*) its → *execution state* are permanently stored, e.g. on a hard disk. If a deactivated service shall be instantiated again in order to continue its task execution, its → *activation* has to be initiated. If a service is deactivated, its → *usage state* is set to US_UNKNOWN. The other resulting values of a service's state are → *suspended* and → *active*.

deactivation
: the procedure in which a → *service* is halted. The object(s) associated with the service are removed and (is the service is an agent) the execution state is permanently stored. To revoke the deactivation the service has to be activated again (cf. → *activation*), i.e. the corresponding object(s) are instantiated anew.

de-registration
: the procedure in which an → *agency*, → *agent*, or → *place* entry is removed from the → *region registry*.

distributed agent environment (DAE)
: the environment in which → *agent*s execute their tasks. The DAE consists of various agencies (cf. → *agency*) that are distributed throughout a network. Agencies can be grouped to → *region*s, and each single agency comprises one or more → *place*s. → *mobile agent*s are able to migrate from one place to another place of the same or a different agency.
The Grasshopper DAE is built on top of a → *distributed processing environment* . In this way, an integration of → *mobile agent technology* and the traditional client/server paradigm is achieved.

distributed processing environment (DPE)
: supports the remote interaction between distributed software components. These software components can be divided into the categories client and server. During an interaction, a client invokes a service offered by a server. It is possible that a single component takes the client and the server role relative to distinguished other components.

DPE
: cf. → *distributed processing environment*

dynamic method invocation
: a mechanism for client/server communication. This mechanism is useful if the client does not have access to a specific server → *proxy*. The client is able to construct a message at runtime by specifying the signature of the server method that shall be invoked. Dynamic messaging can be used both synchronously and asynchronously.

execution block
: one code segment within the method → *live*() of a → *mobile agent*. Each execution block is completely executed in one → *place*, and the last statement of each block is the → *move*() method. That means, after performing a complete execution block, the agent migrates to another place. After the → migration the next execution block is per-

formed. This concept has been introduces by Grasshopper since Java does not support the transfer of an agent's execution stack.

execution state

indicates which part of an → *agent*'s code is currently executed. Besides, the execution state comprises values of important variables of the agent's code. The execution state is especially important for → *mobile agent*s. When a mobile agent migrates from one place to another, its code and its execution state is transferred to the destination location. After its arrival, the agent is instantiated again and supplied with its execution state. This enables the agent to continue its task execution exactly at the point where it has been interrupted before the migration.
Note: Do not mix up the meaning of an agent's execution state with a service's → *state*, i.e. its mode of existence.

identifier

enables the unique identification of distinguished entities within the → *distributed agent environment* . In the context of Grasshopper, especially → *agent*s and agencies (cf. → *agency*) are supplied with a unique identifier. An identifier consists of the following five components:
- the prefix "Agent", "Service", "Listener", or "Unknown" (Note that in this context an agency is regarded as a service.)
- 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 at which the identifier has been created:
"hh:mm:ss:msmsms"
- the number of clones of the identifier

IIOP

cf. → Internet Inter-ORB Protocol

InformationDesk

the "default" → *place* that exists within each Grasshopper → *agency*. If an agent migrates from one agency to another without specifying the desired destination place, it is sent to the information desk place of the target agency.

Internet Inter-ORB Protocol

IIOP has been specified by the Object Management Group (OMG) in the context of the → *Common Object Request Broker Architecture*

(CORBA). IIOP enables interactions between objects implemented in different languages and residing in distinguished environments.

live()
: the most fundamental method of each Grasshopper agent. This method specifies the actual task of the agent. In case of a → mobile agent, the live() method is subdivided into several → *execution block*s.

localisation
: the process in which an → *agency*, → *agent*, or → *place* entry is requested from the → *region registry*. The localisation of these components is essential for the management of the → *distributed agent environment* and for the task execution of → *mobile agent*s.

location
: specifies a physical address within the → *distributed agent environment* . For instance, the location of an → *place* comprises a host name, a port number , the name of the agency to which the place belongs, and the place name. A location consists of the following five components:
- the protocol scheme
- the IP address of the host
- the port (omitted by default)
- the name of the → *agency*
- the name of the → *place* (omitted for agency locations)

MASIF
: cf. → *Mobile Agent System Interoperability Facility*

migration
: the procedure in which a → *mobile agent* moves from one → *place* of the → distributed agent environment to another. In contrast to → *remote execution*, migration allows an agent to continue its task execution at the destination location exactly at the point where it has been interrupted before the migration. For this purpose, not only the agent's code, but also its → *execution state* is transferred.

mobile agent
: an → *agent* that is able to migrate from one → *place* of the → *distributed agent environment* to another. (cf. → *migration*). In contrast to mobile agents, → *stationary agent*s cannot migrate by themselves. they are more or less associated with a single place, and they can only be moved by their owner.

Mobile Agent System Interoperability Facility

the Mobile Agent System Interoperability Facility. The MASIF specification is the first → *mobile agent* standard of the Object Management Group (OMG). It has been initiated in November 1995 by means of a request for proposal, and it has been accepted as new OMG technology in February 1998. MASIF has been developed by Crystaliz, General Magic, GMD FOKUS, IBM Japan, and the Open Group. The idea behind MASIF is to enable interoperability between mobile agent platforms of distinguished manufacturers. One main objective was to specify interfaces that can easily be integrated into already existing platforms. Currently, MASIF only provides a minimal set of functionality. However, further requests for proposal will probably be initiated in order to continue mobile agent standardisation.

mobile agent technology

the way to realise distributed software applications by means of → *mobile agent*s. Mobile agent technology can be regarded as an alternative to or as an enhancement of the traditional client/server paradigm. While the client/server paradigm is mainly based on remote communication between components that are distributed throughout a network (i.e. clients and servers), mobile agent technology allows to move service logic dynamically and on-demand to the network locations where it is currently needed. Since the mobile agent platform Grasshopper is built on top of an → *object request broker* , it achieves an integration of both technologies.

mobility

the fundamental capability of a → *mobile agent*. Mobility allows a mobile agent to migrate from one → *place* of the → *distributed agent environment* to another. (cf. → *migration*).

move()

the method of a Grasshopper → *mobile agent* that initiates its → *migration* from one → *place* to another.

object request broker (ORB)

supports remote communication via RPC between distributed software components. Each communicating component belongs either to the category client or server. Naturally, in distinguished communication scenarios one single component can switch between the client and the server role. The object request broker manages the connection establishment between a client and a server. Usually an ORB can be regarded as basis of a → *distributed processing environment* .

ORB

cf. → *object request broker*

place

one specific area within an → *agency*. Each agency consists of a core (cf. → *core agency*) and one or more places. At least the place → *InformationDesk* exists in each agency, representing the default entry point for → *mobile agent*s. A place can be defined in order to group specific agency capabilities. For instance, a telecommunication place could provide adapter services for the access of telecom hardware devices such as switches, or a post place could comprise email or fax services. Thus places represent more or less logical, conceptual entities. Note that the functionality of a specific place can only be accessed by an agent if this agent currently resides within this place.

proxy

Regarding remote client/server interactions, a proxy (object) is responsible for the connection establishment between clients and servers. The proxy is created at the client's site, and its methods are invoked locally by the client. The proxy connects itself with the server site and invokes the corresponding methods remotely on the server object, eventually via a sort of communication service.

region

a subset of the complete → *distributed agent environment* . A region is a logical entity that groups a set of agencies (cf. → *agency*) that belong to a certain authority, e.g. a company or administrative domain. Associated with each region is a → *region registry* that allows the → *registration*, → *de-registration*, and → *localisation* of → *agent*s, agencies (cf. → *agency*), and → *place*s.

region registry

a component within the → *distributed agent environment* that allows the → *registration*, → *de-registration*, and → *localisation* of → *agent*s, agencies (cf. → *agency*), and → *place*s. A region registry is associated with each → *region* within the DAE. Each registry runs on its own Java virtual machine.

registration

the procedure in which an → *agency*, → *agent*, or → *place* entry is added to the → *region registry*. The registration of these entities is the precondition for their → *localisation*.

remote execution

allows to create and activate an object on a remote host. It is even

possible that, during its task execution, the object itself initiates its own execution on another host. That means that the object is removed from the current host and created on the remote one. However, note that the task execution starts on each new host from the beginning on. In contrast to this, → *migration* allows to execute different parts of the object's (→ *mobile agent*'s) task on distinguished hosts.

remote method invocation

Java RMI enables remote interactions between Java objects residing on different virtual machines, possibly running on different hosts. A Java program can make a call on a remote object once it obtains its reference. This can be achieved by looking up the remote object in the bootstrap-naming service provided by RMI. RMI is part of the Java Development Kit (JDK) provided by Sun.

resumption

the procedure in which the → *suspension* of a → *service* revoked. After its resumption the service thread is activated anew, i.e. the service continues its task execution.

RMI

cf. → remote method invocation

secure socket layer

SSL is the de facto standard Internet security protocol. Originally, it has been designed by Netscape. SSL is one of the most widely used security protocols on the Internet, and currently it is even gaining momentum in the International Standards Organisation (ISO). It can be used to protect almost all traffic over TCP/IP networks. (See also → *socket*)

service

a software component that offers certain functionality to other entities. Usually a Grasshopper service consists of one or more (Java) objects and is realised by means of a Java thread. → *agent*s as well as agencies (cf. → *agency*) are regarded as services, derived from the super class `service`. At each point of its life time, a service resides in a specific → *state*.

socket

Sockets represent the lowest level of programming to the TCP/IP layer of a network. In many ways they are similar to file handles in traditional programming languages. Within a program you open a socket, read from the socket, write to the socket, and close the socket. The main difference is that instead of a file handle being associated

with a file on a disk drive, a socket is associated with another program that is also reading and writing data on the socket. The communicating programs may run on different hosts.

SSL

cf. → *secure socket layer*

state

the mode of existence of a → *service*. A service can be → *active* (i.e. the service objects are instantiated and the service thread is running), → *suspended* (i.e. the service objects are instantiated and the service thread has been stopped) or → *deactivated* (i.e. the service objects are not instantiated).
Note: Do not mix up the meaning of a service's *state* with the meaning of the → *execution state* of a → *mobile agent*.

stationary agent

an → *agent* that is more or less associated with a single → *agency*. In contrast to a → *mobile agent*, a stationary agent is not able to migrate actively from one → *place* of the → *distributed agent environment* to another. However, even a stationary agent can change its → *location* if this is initiated by an external entity, such as the agent owner.

stub generator

a separate tool of Grasshopper for the generation of → *proxy* classes. Instances of these classes, the so-called proxies (or proxy objects) are accessed locally by a client in order to invoke methods of a remote server.

suspended

one possible → *state* of a → *service* that results out of the service's → *usage state*. If a service is suspended, the associated object(s) are still existing, but their task execution is temporarily interrupted, i.e. the service thread is stopped. If a suspended service shall continue its task execution, its → *resumption* has to be initiated. If a service is suspended, its → *usage state* is set to US_SUSPENDED. The other values of an service's state are → *deactivated* and → *active*.

suspension

the procedure in which the task execution of a → *service* is temporarily interrupted. In contrast to the → *deactivation* of a service, the associated object(s) are not removed. Instead, only the service thread is stopped. To revoke the suspension the service has to be resumed again (cf. → *resumption*), i.e. the thread has to be activated anew.

synchronous communication

a communication mechanism between clients and servers. The client invokes a method of the server and is blocked until the execution of the method is finished. After retrieving the result of the method, the client continues its own task. The counterpart is → *asynchronous communication.*

usage state

The usage state can have the value US_ACTIVE, US_SUSPENDED, US_FLUSHED or US_UNKNOWN. The usage state indicate whether a service is currently → active (usage state = US_ACTIVE), → suspended (usage state = US_SUSPENDED), or → usage state = US_UNKNOWN).

C Index