LARS Community Edition

Agent and Messaging Handbook

Version 3.1

Contents

1 Introduction

1.1 Who should read this document?

This document is written for agent programmers and LARS administrators.

1.2 Overview

The document contains all the necessary information about LARS agents and messaging:

The following Topics are covered:

- A brief introduction to Agents, Multiagent-Systems, LARS and LARS agents.
- A general discussion on Messaging, the different Message Types and other basic message features supported by LARS.
- A list of all Agents with an explanation of the services they offer.

- In-depth descriptions of How-To connect any client to a LARS platform.
- Configuration of LARS.
- Guides on How-To use the LARS Cockpit, which is an application that helps to administer a LARS Platform.

2 Living Agents Runtime System

2.1 Definitions

The general trend of growing complexity of present software systems leads in most cases to increased difficulty in developing high performance, low-maintenance and adequately functioning products. Agents, which can be viewed as electronic digital assistants, can eliminate this problem. Living Systems Agents can assist System Users in the sense that they can make decisions independently, but on behalf of the User and proactively relieve Users from routine assignments.

2.1.1 What is an Intelligent Software Agent?

To be honest, this topic like it was in the early days of object-oriented software design is not easy to answer, because within the agent society there exists no universally accepted definition of an intelligent software agent. But the existing definitions are very closely related to each other, as shown in the following definitions:

(1) An intelligent software agent is a *software object* that *proactively* operates *on behalf of its human master* in performing a *delegated task*.

(2) An *intelligent software agent* is a *program*, that is *situated* in some *environment* and that is capable of *flexible autonomous* action in order to meet its *design objectives*.

The following characteristics can be deduced from the above definitions:

- *autonomy*:
  Agents are able to act independently without the intervention of other systems or humans - they have control over their internal state and over their behavior.

- *reactivity*:
  Agents perceive their environment and are able to respond to changes.

- *pro-activeness*:
  Agents exhibit goal-directed behavior by taking the initiative. Therefore they decide and act along a solution path in order to achieve predefined goals.

- *collaboration*:
  Agents are able to interact with other agents (and possibly humans).

- *domain expertise*:
  Agent's perceptions and actions imply and contain complete and detailed domain specific knowledge / industry know how.

The characteristics just mentioned show that Agents differ from usual objects in object-oriented software in the sense that an agent is able to make decisions autonomously within the designated strategy defined by the user.

2.1.2 What is a Multiagent System?

A multiagent system is a system, in which several interacting intelligent agents pursue sets of goals by performing sets of tasks.

2.1.3 What is LARS?

LARS (Living Agents Runtime System) is a multiagent system developed by the living systems AG. It is written entirely in Java.

2.1.4 What is a LARS Agent?

A LARS Agent is an intelligent software agent, that is programmed entirely in Java and that runs on a LARS platform.

A Mobile agent is able to transmit itself across a computer network (e.g. the internet) and continue its execution on a remote site.

2.2 Benefits of LARS Agents

The Benefits of the LARS Agent Technology are as follows:

(1) LARS agents can be seen as the natural representation for modeling a wide range of domains, because they can be used to model both active and passive entities.

(3) LARS agent technology is able to cover an extensive area of different technologies and application fields (e.g. collaborative commerce, telecommunications, distributed systems).

(4) Because of their slim design and their role-oriented nature, LARS agents are easy to model, develop and extend. Their ability to communicate makes them ideal user representatives (e.g. in collaborative commerce).

(5) LARS agents offer a configurable load balancing, which allows the usage of LARS in environments, where scaling issues are important.

(6) The messaging mechanism used in LARS abstracts a low-level communication protocols (changes of the low-level communication protocol e.g. RMI is possible at configuration time and does not require recompiling). LARS messaging system allows synchronous and asynchronous messages to be sent.

2.3 LARS Agent Lifecycle

A LARS agent runs through a predefined lifecycle. First its messenger including its message box and its thread are created and started by AgentManager. Then the following run levels are passed through:

0	EMBRYONIC	initial run level

1	`INTERPRET_METHODS_REGISTERED`	Reached after agent has initialized internal data structures for calling interpret methods automatically when receiving corresponding service requests (e.g. when receiving a message with the service "`do_something`", the method `protected boolean interpretDoSomething (Message)` is called).
2	`CONFIG_FILE_READ`	Reached after agent has read (but not yet interpreted) its config files from file system without any unforeseen exceptions/errors.
3	`ACTIONS_INITIALIZED`	Reached after agent has executed its initializeAction () method without any unforeseen exceptions/errors.
4	`CONFIG_FILE_INTERPRETED`	Reached after agent has interpreted its config files method without any unforeseen exceptions/errors

5	POOL_INITIALIZED	Reached after agent has passed the initializing of its thread pool without any unforeseen exceptions/errors. Note, that a pool is initialized only, if agent load balancing was configured in the appropriate start_agent message for AgentManager - the run level instead is reached independently of an agent running pooled or unpooled.
6	RUNNING	Reached after agent has executed its initializeMessages () method without any unforeseen exceptions/errors. As long as the agent is in run level RUNNING, it executes its executeWhileRunning () method. There it usually waits for incoming messages and interprets them[1].
7	STOPPED	Reached automatically, if unforeseen exceptions/errors occur in agent initialization phase[2] or is intentionally set to stop/shutdown the agent.
8	TERMINATED	Reached, if agent has executed its `terminateAction()`. In the `terminateAction` method, the agent sends a ServiceBroadcastMessage to the service providers for "`sender_rip`" to inform everyone, who is interested, about its death[3].

[1] If the agent runs in a pooled mode, it does not interpret the message itself but hands it over to one of its interpretation threads from its thread pool.

[2] The initialization phase involves all run levels that are less than RUNNING.

[3] One of those service providers is AgentMessageRouter, which then removes the Messenger including the inbox of the agent from the message router.

Note: To assure a deterministic behavior, agent run levels can only increase (e.g. it is impossible to set the run level RUNNING after the run level STOPPED has been reached).

3 Messaging

3.1 A Primer on sending Messages

As sending messages is essential for all agents, the following section tries to outline this process. If you already know how to send messages from agent to another agent within LARS, you can safely skip this section.

Usually, agents will have to transmit information to other agents at some point. This is done by sending messages with certain *services* (or subjects) and *contents*. Further on in this document, we will refer to the "requested service" as the "subject".

The service of a message prompts the concerned agent to perform a certain action. If the action needs supplementary information, this is transmitted in the content section of the message. For more complex information, the content section of an agent message will usually contain a Map.

Below is a sample code for generating and sending a message:

```
1. Map content=new HashMap();
2. content.put("value1", String.valueOf(359901));
3. content.put("value2", "bogus");

4. sendMessage(new SingleMessage("test", agentA, content));
```

Line 4 would result in a message with subject "test" sent to Agent A on the current platform. The Message has two parameters stored in a Map (*value1* and *value2*).

3.2 The different Message Types

Messaging is the technology used to interlink agents. This chapter describes the general messaging architecture, the message format and explains how to use asynchronous and synchronous messaging.

The LARS messaging system currently supports six different message types, which are shown in Figure 1. Each of these messages can be represented in XML.

All message classes are located in the `com.ls.lars.communication` package. A detailed description of the different message classes follows in the next paragraphs.

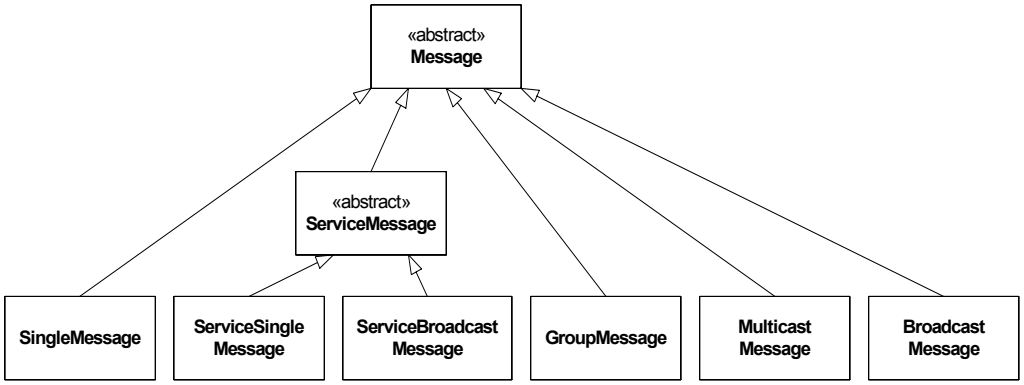

Figure 1: class diagram for LARS messages[4]

3.2.1 Message

All messages between agents or between agents and agent-like clients in the LARS (Living Agents Runtime System) inherit from the abstract `Message` class. The actual recipient is determined by the message router and depends on the type of the message.

The abstract message class has the following attributes:

Field	Description
Service **(String)**	Service that is accessed by the message. This attribute needs to be set by the creator of the message.

[4] Note, that the BroadcastMessage is not yet implemented in release 3.1.

	Note: there is a **standard for naming message services**: • a message service is written in lower case • if a service consists of multiple words, they are separated by an underscore ("_"), e.g. "set_constants" This standard is needed for automatic registration of "interpret" methods in agents.
sender **(String)**	Name of the agent that sent the message. This attribute is initially null and automatically set in the CommunicationTemplate at send time. Note: the sender attribute can be used to determine if a message is sent or unsent. If the sender is not null, the message is assumed to be sent, otherwise it is not sent.
forwardedBy **(List of Strings)**	List of all agents that forwarded this message. This attribute is maintained automatically and initially set to null. It remains null as long as this message is not forwarded by someone.
replyWith **(String)**	ID that is used to identify a reply to this message, see section 3.3.3 for details. This attribute needs to be set by the creator of the message. Otherwise it remains null. It can only be set while the message is "not yet sent".
inReplyTo **(String)**	ID that identifies this message as a reply to a previous message, see section 3.3.3 for details. This attribute is automatically set by the Message.createReply method, if the message, for which the reply is generated, contains a non-null replyWith attribute (otherwise the attribute remains null).
replyCounter **(short)**	Specifies the number of replies that have already been *generated* for this message. Note: this does not tell you anything about the replies being sent or not!
priority **(short)**	Priority with which the message is delivered. This attribute needs to be set by the creator of the message, otherwise it remains

	`Message.PRIORITY_NORMAL`.
qualityOfService **(short)**	The quality of service defines the suitable conditions in which an automatic reply should be sent to this message by the system. The System offers constants for different qualities in the classcom.ls.lars.communication.QualityOfService.
currentHops **(short)**	Number of messengers this message already passed. If this number is reaching a threshold (see next *expireHops*) it is considered to be bouncing and therefore removed from the net.
expireHops **(short)**	Number of messengers this message is allowed to pass, before it is considered to be bouncing and therefore removed by the routing components.
sentTime **(long)**	Time in milliseconds since 01.01.1970 specifying, when this message was sent out (unsent messages have a sentTime of 0).
content **(Object)**	The content of the message. Typically this is either a String containing the message as an (XML) string or it is a List containing a collection of Strings, Lists or Maps or it is a Map containing a collection of Strings, Lists or Maps. If the message is sent via a RemoteMessenger, it is important, that the content consists of serializableobjects.

In config files `service` and `content` are the only required entities. Additional mandatory parameters may exist depending on the specific type of the message.

3.2.2 SingleMessage

This message type is used for a peer-to-peer communication between agents or agent-like clients, where the name of the receiving agent is known. The SingleMessage inherits directly from the `Message` class and contains the following additional attributes:

Field	Description
receiver **(String)**	Short name (without LARS id and IP) or fully qualified name (containing the agent name, an "@", and the LARS id and IP) of an agent to

	whom this message shall be delivered.

For example:

```
<MESSAGE>
    <service>ping</service>
    <type>single</type>
    <receiver>amr</receiver>
    <sender>AgentExample@192.168.100.114-lars</sender>
    <sentTime>993634068273</sentTime>
    <currentHops>1</currentHops>
    <expireHops>10</expireHops>
    <priority>20</priority>
    <qualityOfService>0</qualityOfService>
    <content></content>
</MESSAGE>
```

Figure 2: Example of an XML-representation of a SingleMessage

3.2.3 Service Messages

Agents can register themselves as providers of services. This mechanism can be used to distribute a message to a service provider and to hide the agent name from the clients requesting for the service. The agent, which receives the service message, is dependent on the concrete platform configuration.

Note, that the ServiceMessage itself is abstract and only the ServiceSingleMessage (see below) and the ServiceBroadcastMessage (see below) can be instantiated.

The ServiceMessage contains only one additional attribute:

Field	Description
platform **(String)**	Name of the platform (LARS id and IP) where the service is requested. This is an optional attribute, which allows requesting a service from a specific platform instead of using the default one[5].

AgentMessageRouter is responsible for service Registration.

[5] The default platform for an agent-like client is the first platform, which is reached by the message; for an agent it is the local platform.

Registering a Service

An agent can register itself as a service provider by sending a '**register_service**' message to AgentMessageRouter containing the service(s) it is ready to provide.

Note: only the service provider itself is allowed to register himself as provider for a service.

```
<MESSAGE>
    <receiver>amr</receiver>
    <service>register_service</service>
    <content>log_system_information</content>
    <content>any_other_service</content>
</MESSAGE>
```

Figure 3: Example of a register_service message

Unregistering a Service

A single service can be unregistered by sending an '**unregister_service**' message containing the name(s) of the service(s) as content. If no agent is provides that service, the service list is deleted.

Unregistering all Services of an Agent

All services an agent has registered can be unregistered by sending an '**unregister_all_services**' message with no additional parameters.

ServiceSingleMessage

This service message is delivered to *only one* service provider (if one exists) of the specified service. It is used for accessing applications (agent-like clients) by the name of the service instead of by their agent-name.

An example of ServiceSingleMessaging:

Two agent-like clients offer the service "transfer_money_to_account".

A ServiceSingleMessage will arrive at only one of the clients. This allows shutting down and restarting the agent-like clients at runtime even though it is certain that the transfer_money_to_account service is accessible and only once by one message.

```
<MESSAGE>
    <service>transfer_money</service>
    <type>service_single</type>
    <platform>192.168.100.114-lars</platform>
    <sender>CustomerOrganization@192.168.100.96-lars</sender>
    <sentTime>993655353260</sentTime>
    <currentHops>1</currentHops>
    <expireHops>10</expireHops>
    <priority>20</priority>
    <qualityOfService>1047548</qualityOfService>
    <content>
        <transactionId>8977246</transactionId>
    </content>
</MESSAGE>
```

Figure 4: Example of an XML-representation of a ServiceSingleMessage

ServiceBroadcastMessage

A ServiceBroadcastMessage is delivered to *all* service providers of the specified service.

Usage example: When an agent dies, it sends ServiceBroadcastMessage to the service "sender_rip" to inform all those agents, that might be interested in an agent's death (e.g. AgentTimer, AgentCron, AgentMessageRouter to unregister the dying agent as their client).

```
<MESSAGE>
    <service>log_system_information</service>
    <type>service_broadcast</type>
    <platform>192.168.100.114-lars</platform>
    <sender>Cockpit.993655092615@192.168.100.96-lars</sender>
    <sentTime>993655353273</sentTime>
    <currentHops>1</currentHops>
    <expireHops>10</expireHops>
    <priority>20</priority>
    <qualityOfService>1047548</qualityOfService>
    <content></content>
</MESSAGE>
```

Figure 5: Example of an XML-representation of a ServiceBroadcastMessage

3.2.4 Group Messages

Group messages have been introduced in order to send messages to a whole group of agents. AgentMessageRouter provides the necessary services to manage message groups.

Field	Description

group **(String)**	Short name (without LARS id and IP) or fully qualified name (containing the group name, an "@", and the LARS id and IP) of the group to which this message shall be delivered.

For example:

```
<MESSAGE>
    <service>compute_amount</service>
    <type>group</type>
    <group>calculation</group>
    <sender>Cockpit.993655092615@192.168.100.96-lars</sender>
    <sentTime>993655353274</sentTime>
    <currentHops>1</currentHops>
    <expireHops>10</expireHops>
    <priority>20</priority>
    <qualityOfService>1047548</qualityOfService>
    <content>
        <customerRelation>friend</customerRelation>
        <productId>34089</productId>
    </content>
</MESSAGE>
```

Figure 6: Example for an XML-representation of a GroupMessage

Subscribing to a Message Group

Any agent can subscribe any other agent to a message group. To subscribe a single agent or a group of agents to a message group, a message must be sent to AgentMessageRouter. This message contains the group name as well as all agent names that should be subscribed. If the group does not exist, a new message group is generated automatically.

```
<MESSAGE>
    <receiver>amr</receiver>
    <service>subscribe_to_message_group</service>
    <content>
        <group>theGroupName</group>
        <agent>agentname1</agent>
        <agent>agentname2</agent>
    </content>
</MESSAGE>
```

The AgentMessageRouter returns 'subscribe_to_message_group_succeeded' if all agents are subscribed, otherwise it returns 'subscribe_to_message_group_failed'.

Unsubscribing from a Message Group

To unsubscribe a single agent or a list of agents from a message group the '**unsubscribe_from_message_group**' message is sent to AgentMessageRouter. A group is removed from the list of message groups if all agents are removed from that group (i.e. it becomes empty). Parameters are the same as with the subscribe message. The AgentMessageRouter returns 'unsubscribe_from_message_group_succeeded' if all agents could unsubscribe or 'unsubscribe_from_message_group_failed' in all other cases. If the group does not exist a warning message is logged, but a success message is returned.

Unsubscribing an Agent from all Message Groups

To remove an agent from all message groups where it is subscribed to (e.g. when shutting down the agent) the '**unsubscribe_from_all_message_groups**' message is provided. The only required passed is the name of the agent that should be unsubscribed. The AgentMessageRouter returns 'unsubscribe_from_all_message_groups_succeeded', if the agent could unsubscribe from all groups otherwise it returns 'unsubscribe_from_all_message_groups_failed'.

Removing a Message Group

A message group can be deleted by sending the '**remove_message_group**' message. The only required parameter is the name of the group to be removed. The AgentMessageRouter returns 'remove_message_group_succeeded' if the group is deleted.

Sending Messages to a Message Group

A message can simply be sent to a group of agents by specifying the group name as the receiver of the message.

```
<MESSAGE>
    <group>theGroupName</group>
    <service>ping</service>
</MESSAGE>
```

3.2.5 MulticastMessage

A multicast message is sent to a list of receivers. Each of the specified receivers gets a clone of the original message. The multicastMessage

inherits directly from the `Message` class and contains the following additional attributes:

Field	Description
Receiver (List)	List of recipients[6] of the agents to whom this message shall be delivered.

For example:

```
<MESSAGE>
    <service>recalculation_event</service>
    <type>multicast</type>
    <receiver>orbiter</receiver>
    <receiver>moon</receiver>
    <receiver>asteroid</receiver>
    <sender>spaceshuttle@192.168.100.96-lars</sender>
    <sentTime>993655625364</sentTime>
    <currentHops>1</currentHops>
    <expireHops>10</expireHops>
    <priority>20</priority>
    <qualityOfService>1047548</qualityOfService>
    <content>
        <speed>37.7859</speed>
        <objectId>77</objectId>
        <x>245.567</x>
        <y>567.255</y>
        <z>569.253</z>
    </content>
</MESSAGE>
```

Figure 7: Example of an XML-representation of a MulticastMessage

3.2.6 BroadcastMessage

The BroadcastMessage is not yet implemented in the release 3.2.

[6] short name (without LARS id and IP) or fully qualified name (containing the agent name, an "@", and the LARS id and IP)

3.3 Usage of basic Message features

3.3.1 Forwarding messages

Upon receipt of a message, An Agent may think that another Agent might be interested in the received message, in which case the first Agent would forward the received message.

This functionality is offered by two methods in the CommunicationTemplate and therefore exists in any agent or agent-like client.

The first method allows a received message to be forwarded to a single receiver:

```
void forwardMessage(Message message, String forwardReceiver)
```

The method forwards a message by creating a shallow copy of the message and then forwarding the copy to the new receiver. If the given message is not an instance of SingleMessage, it is converted into a SingleMessage. Then the original message type cannot be determined at the forwardReceiver any more.

The second method allows message forwarding to multiple receivers:

```
void forwardMessage(Message message, List forwardReceivers)
```

The method forwards a message by creating a shallow copy of the message and then forwarding the copy to the new receiver. If the given message is not an instance of MulticastMessage, it is converted into a MulticastMessage. Then the original message type cannot be determined at the forwardReceivers any more.

Note: For performance reasons both forwardMessage methods only make a shallow copy of the message's content, if you need to change parts of the message (e.g. the content), make a copy yourself before calling forwardMessage.

3.3.2 Asynchronous versus Synchronous communication

In contrast to simple object oriented designs, where synchronous communication between objects is used (by just calling a method and waiting for the result), in a Multi Agent System normally asynchronous communication is used. LARS offers both types of communication synchronous and asynchronous. The advantages and disadvantages of the different approaches are explained in detail in this section.

Asynchronous Messaging

Asynchronous messages can be sent through the send message method of the ItoLars interface.

```
Message message =
    new SingleMessage(  "finish_contract",
                        "AgentContractSupervisor",
                        content);
sendMessage(message);
```

Sending an asynchronous message does not involve blocking (i.e. the sending agent can get on with other work while waiting for a result).

Asynchronous communication should be used if an answer is not needed or when the overhead for administering the reply identifications (see the message attributes replyWith and inReplyTo) is low compared to the performance losses suffered during synchronous communication (because the calling agent is blocked while the called agent performs its tasks).

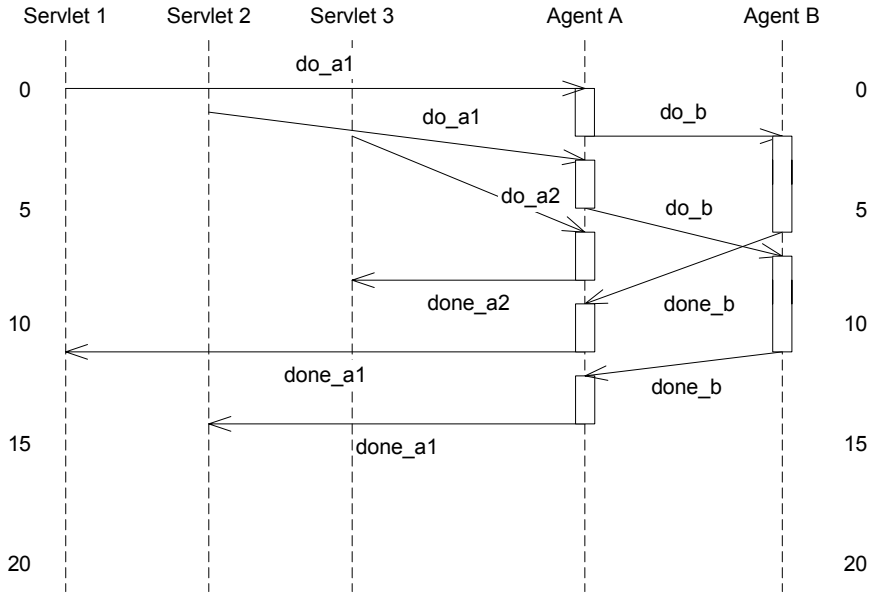

Figure 8: Example for Asynchronous Messaging

Synchronous Messaging

Synchronous Messaging should be used when an Agent needs to wait for the reply to a message (see disadvantages below). The IToLars interface `sendSynchronousRequest` method is used to send a synchronous message.

Please note that synchronous messages cannot work without a `replyWith` attribute.

```
Message message =
    new SingleMessage(  "finish_contract",
                        "AgentContractSupervisor",
                        content);

message.setReplyWith(uniqueReplyId);

try {
    replyMessage = sendSynchronousRequest(message, 4000);
} catch (TimeoutException tex) {
    ...
}
```

Sending a synchronous message involves blocking, i.e. the agent waits until a reply (usually sent by the receiver agent) is received. One can specify a timeout in milliseconds: If no Reply is received within that amount of milliseconds, a TimeoutException is thrown.

Disadvantages: Sending synchronous instead of the asynchronous messages shall be used if it is absolutely ensured that the call does not block an agent for a *long period*. Especially when the agent holds a very important position on the platform, this blocking could result in very bad runtime behavior of the *whole* system.

Example: Think of an agent A, who is to process multiple incoming requests from three connected servlets (see Figure 9).

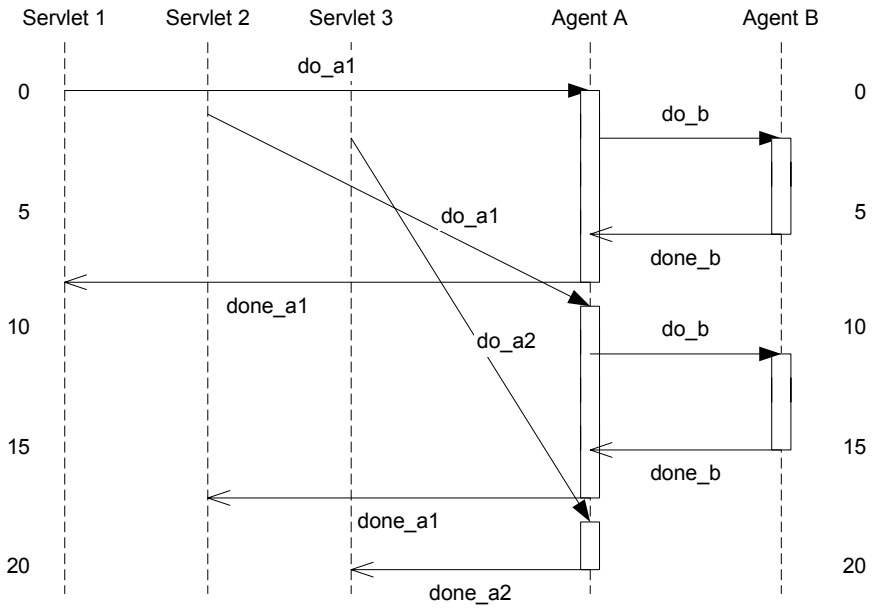

Figure 9: An Example of Synchronous Messaging

When a servlet requests an agent A to communicate with another agent B synchronously, agent A cannot process other messages while waiting

for the reply[7]. This means all other requests to agent A are blocked until it receives the response for the synchronous message or it receives a TimeoutException.

A more sophisticated way would be to send all messages asynchronously using the standard `sendMessage(Message)` method (see Figure 8, which is the same as Figure 9 but using asynchronous messages). In this case the agent is not blocked and thus free to perform any other operations, instead of wasting time while waiting for a response message.

3.3.3 Questions and Answers - How to identify Replies

Questions and Answers are matched in the LARS communication by using the two attributes `replyWith` and `inReplyTo` as in

KQML (Knowledge Query and Manipulation Language).

If the `replyWith` attribute of a message is not set, it means that the value returned by `getReplyWith()` is null (i.e. the sender does not expect a reply). On the other hand if the attribute has a value the message is considered to be a question and the sender expects a reply.

The reply would be a message with a parameter `inReplyTo`, which has a value identical to the question's `replyWith` value.

Example:

The following message, which is sent from agent A to agent B, expects a reply:

```
<MESSAGE>
    <receiver>agentB</receiver>
    <sender>agentA</sender>
    <service>ping</service>
    <replyWith>A-question-0815</replyWith>
</MESSAGE>
```

Agent B sends back the subsequent message:

```
<MESSAGE>
    <receiver>agentA</receiver>
    <sender>agentB</sender>
    <service>pong</service>
    <inReplyTo>A-question-0815</inReplyTo>
</MESSAGE>
```

[7] except that it is running in pooled mode, which is not possible for all agents

Note:

The attributes `replyWith` and `inReplyTo` make it possible to create complex communication patterns of questions and answers. It is very important when generating `replyWith` values to ensure that they are *unique* for the sending agent. This is because the replyWith parameter would be used later on to identify an incoming message as the reply. The `CommunicationTemplate` has a method `createReplyId()`.

3.3.4 Quality of Service

The quality of service attribute for a message determines which conditions are suitable for the system to automatically generate a reply for that message.

The default quality of service for a message is `QOS_REPLY_ERROR`, which means that a reply is required for any possible error that might occur after the message has been sent out. For a detailed description of all available constants have a look at the Java Doc of `com.ls.lars.communication.QualityOfService`.

Automatically generated replies

Different automatic replies can be generated (these are listed below) depending on the problem that brought about the reply:

- `ILarsConstants.SERVICE_DELIVERY_FAILED`: this means that the message given in the content of this message couldn't be delivered.
- `ILarsConstants.SERVICE_NOT_UNDERSTOOD`: this means that the interpretation of the message given in the content of this message failed, because it was not understood or an internal error happened in the agent.
- `ILarsConstants.SERVICE_ACCESS_DENIED`: this means that the interpretation of the message given in the content of this message was denied.
- `ILarsConstants.SERVICE_INTERPRETATION_SUCCESSFUL`: this means that the interpretation of the message given in the content of this message was successful.

All these replies have one thing in common in that they contain the message that brought about the reply as a map (returned by Message.toMap()) within the content of the reply.

Additionally the content of the reply contains the following tags:

- *reasonOfFailureText*: textual description of the cause of the failure
- *reasonOfFailureCode*: An integer that specifies the exact quality of service, which caused the reply (e.g`QualityOfService.QOS_REMOTE_MESSENGER_NOT_REACHABLE`)
- *routeFailedAt* (optional tag)[8]: specifies the name of the messenger, where the routing has failed. This tag is optional, because not all failure reasons deal with a specific messenger.

Note, that such an automatically generated reply has a quality of service value of `QualityOfService.QOS_NONE` to avoid message loops.

```
<MESSAGE>
    <service>delivery_failed</service>
    <type>single</type>
    <receiver>ac@192.168.52.7-lars</receiver>
    <sender>LARS_INTERNAL</sender>
    <sentTime>-1</sentTime>
    <currentHops>0</currentHops>
    <expireHops>10</expireHops>
    <priority>normal<priority>
    <qualityOfService>QOS_NONE</qualityOfService>
    <content>
        <receiver>AgentReminderSender</receiver>
        <sender>ac@192.168.52.7-lars</sender>
        <replyCounter>0</replyCounter>
        <reasonOfFailureCode>QOS_RECEIVER_NOT_AVAILABLE
        </reasonOfFailureCode>
        <sentTime>993708138462</sentTime>
        <inReplyTo />
        <service>notify_canceled</service>
        <content>
            content of the original request
        </content>
        <qualityOfService>QOS_REPLY_ERROR</qualityOfService>
        <type>single</type>
        <replyWith />
        <priority>10</priority>
        <expireHops>10</expireHops>
        <forwardedBy />
        <currentHops>1</currentHops>
        <routeFailedAt>AgentReminderSender</routeFailedAt>
        <reasonOfFailureText>
            no messenger and no forward found
        </reasonOfFailureText>
    </content>
</MESSAGE>
```

Figure 10: An Example of a delivery_failed message

[8] The *routeFailedAt* tag is optional, because not all failure reasons deal with a specific messenger.

Defining a composite quality of service

Generating any of the available constants and using binary operations can define a composite quality of service without a predefined constant.

For example a quality of service that requests system replies for an agent failure at interpret time and also for a routing problem would be generated as shown below in Figure 11.

```
Message message = ...;
int quality =
    (QualityOfService.QOS_AGENT_FAILURE
     | QualityOfService.QOS_ROUTING_FAILURE)

message.setQualityOfService (quality);
```

Figure 11: An Example of a newly defined composite quality of service

3.3.5 Messaging Architecture

A message is sent by the Agent's messenger and then routed to the recipient(s) by the message router. To prevent messages being lost if an agent is currently busy, each agent's messenger has a message box to store incoming messages.

Different messengers can be used to achieve communication depending on where the agent resides.

Agents running on the LARS platform would use a local messenger while Agents or agent-like clients running remotely would use a remote messenger. There are different types of remote messengers, which use the abstract from the underlying network protocol:

- *SocketMessenger* is used for sending XML data.
- If a socket connection is to be used, but XML communication is not needed, the much faster *JSocketMessenger*, which transports serialized messages over the net, is used.
- The *RMIMessenger* also uses serialization, but uses the underlying protocol RMI (Remote Method Invocation) instead of using socket communication directly.
- The *JMSMessenger* can also be used for communication between homogeneous and heterogeneous platforms. It has all the features of *RemoteMessenger* and it also has the ability to transmit persistent messages across the platforms. All it needs is an extra layer i.e. a JMS provider for creating as well as locating the queues. Queues are just containers of messages.

- Encrypted communication with serialized message objects is possible with the help *JSecureSocketMessenger*.
- Finally HTTP tunneling enabled by the HTTPMessenger can also be used.

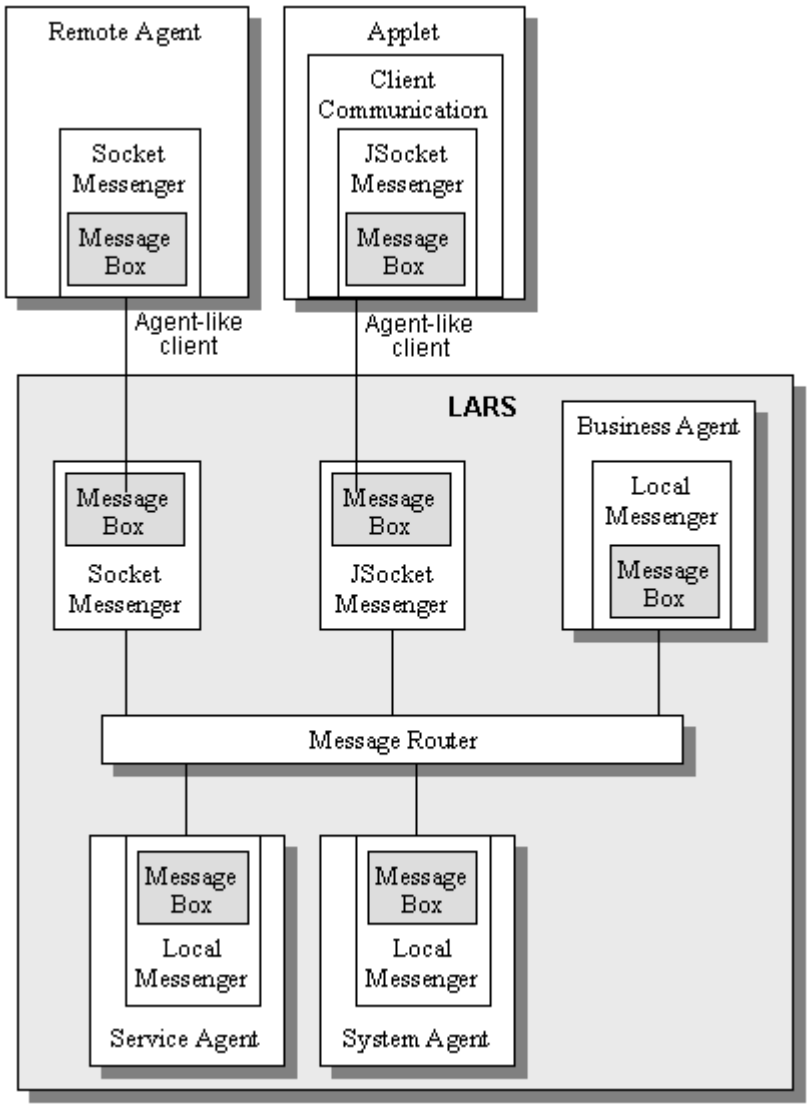

Figure 12: General messaging architecture of LARS

3.4 The base class for LARS communication: CommunicationTemplate

The CommunicationTemplate is the base class for all LARS agents and the agent-like clients (such as servlets, applets), respectively. This class provides the basic communication and logging functionalities needed by LARS agents.

3.4.1 Attributes

The CommunicationTemplate and every single agent or agent-like client provides the following attributes, which are essential for agent handling on the LARS platform:

- the *agentName*, which is necessary for uniquely identifying every single agent on a running LARS platform.
- the *homePlatform* identifier – the name of the platform where the current agent originated.
- the *startTime*, which can be used to determine the actual lifetime of an agent.
- a set of *connection Parameters*, which hold protocol specific information regarding the remote communication.
- the *logger* - every single agent can keep its own logging details while doing its job.

Agent naming

An Agent has a local name and a global name and this should be considered when the Agent is being referred to.

Local names are used for identifying agents uniquely on a single platform. For example the local names 'AgentStaticMaker' and 'AgentUpdateMaker' may be based on the same source code, but they can be uniquely identified by their local names.

An agent's global name is useful for identifying the agent uniquely across different LARS platforms. Remember that 'amr' is used for identifying the AgentMessageRouter on the local platform, however if your agent wants to communicate with the AgentMessageRouter on a remote platform, you could not simply use 'amr' without running into trouble.

Each agent's global name is a combination of the agent's local name and its home platform using the following format:

<*localName*>@<*homePlatform*>,

The *homePlatform* is built from the platform's IP address and the *platformId* as follows:

<*ipAddress*>-<*platformId*>

For example, AgentManager on the lars4 platform (IP address: 141.28.227.129). This agent's local name is '**am**', and its global name would be '**am@141.28.227.129-lars4**'.

Hint: There is a utility class `LarsNaming` in the package `com.ls.lars.communication`, which can be used to construct global names from locally ones, or to extract local names from globally ones.

3.4.2 Methods

The CommunicationTemplate implements the `IToLars` interface and therefore uses the following connection- and message-specific methods:

Connection-specific methods

- *void setConnectionParameters(Map)* – sets the connection parameters for connecting to a LARS platform
- Map getConnectionParameters() – returns the actual connection settings
- void connect() – connects to the LARS platform using the connection settings as given in the CommunicationTemplate constructor
- void disconnect() – disconnects from the LARS platform (if currently connected)
- boolean getConnectionStatus() – returns the current connection status

Message-specific methods

- *void sendMessage(Message)* – sends an asynchronous message to another agent/agent-like client
- *void forwardMessage*(*Message, String*) – forwards the given message object to the given receiver.
- *Message sendSynchrononousRequest(Message)* – sends a synchronous message to another agent and waits for the responding message.
- *Message sendSynchronousRequest(Message, long)* – sends a synchronous message to another agent and waits for the responding message. The second parameter specifies a timeout, after which this method returns null.

- *Message receiveBlockedMessage()* – returns the next incoming message, which was sent to the current agent.

- *Message receiveBlockedMessage(long)* – returns the next incoming message, which was sent to the current agent. The second parameter specifies a timeout, after which this method returns null.

4 Agents

4.1 Introduction

This chapter covers generic agents that are available within LARS. You will find a detailed description of agent interfaces, reactions and behaviors. Readers must note that agent description in this chapter differs from that in the standard Java class documentation. Message handling and agent communication over the LARS are the focus of the generic agent documentation.

Generic agents are divided into system agents and business domain specific agents. The business domain specific agents are further classified according to agent functionality.

In documenting generic agents, the following terms are used:

Function generally describes the agent's functionality, what it is designed for and what it can do.

Relation describes how other agents relate to the agent being described and visa versa.

Messages are the most extensive part of every agent description. Messages is a list of message services the agent is able to provide,

describing what the message service function is, how an agent reacts on receiving a message, including the response message service.

Config file describes the agent configuration file. Here you can find advice on writing config files to include information on how to avoid common pitfalls. In some cases config file samples are also presented.

4.2 System Agents

4.2.1 AgentTemplate

Child of: CommunicationTemplate

Related to: all other agents

Functionality: All living markets agents descend from this class, which has the core functionality of answering messages that cannot be understood by the intended agent. AgentTemplate sends a "pong" reply to a "ping" message.

Incoming Messages

All incoming messages are tested in order to determine whether a valid and corresponding `interpretXXX()` can be found. If no valid `interpretXXX()` is to be found, AgentTemplate's `performDefaultBehavior()` method is invoked.

The return type of `performDefaultBehavior()` indicates whether the incoming message was successfully interpreted or not. If the incoming message was not successfully interpreted, a 'not_understood' messages is sent to the originator of the incoming message.

All messages that AgentTemplate can handle are listed below.

set_access_privileges (Config file only)

Function: Sets the privileges for accessing the services of the current agent.

Parameters: map containing the privilege settings

Responses: none

Note: This message is used for defining access privileges in order to control which agent is allowed to request what services.

Service access control can be set in two ways:

- specificSetting refers to the case in which service access privileges are regulated.

- generalSetting refers to the case in which service access control is not regulated.

> All agents are allowed to access the service under consideration. generalSetting is the default setting when access privileges are not regulated. A message with set_access_privileges subject will be interpreted only when it is read from the config file or when the sender is LARS_INTERNAL.
>
> The default access privileges for all services are set to FULL_SERVICE_ACCESS, which can later be changed in AgentTemplate sub classes by modifying the *defaultServiceAccessPrivilege* member variable.

The general schema for specifying service access rules is shown below:

```
<generalSettings>[full_access | no_access]</generalSettings>

<specificSettings>
    <service>[{single_service_name} | *]</service>
    <accessRule>
        <permission>[allowed | denied]</permission>
        <basedOn>[agent-name | platform-name]</basedOn>
        <name>[{defined_agent} | {defined_platform}]</name>
    </accessRule>
</specificSettings>
```

An example explaining how to regulate service access privileges is shown below:

```
<MESSAGE>
    <service>set_access_privileges</service>
    <content>
        <generalSetting>no_access</generalSetting>
        <specificSetting>
            <service>{service_name}</service>
            <accessRule>
                <permission>allowance</permission>
                <basedOn>platform-name</basedOn>
                <name>HOME_PLATFORM</name>
            </accessRule>
        </specificSetting>
    </content>
```

```
</MESSAGE>
```

The above example denies access to all agents first. It then permits those agents who reside on the platform on which the provider of the service is running to access the service. The service name is specified between the service tags. If an asterisk "*" is specified between the service tags instead of a service name, a blanket access control to all services is achieved.

set_log

Function: Sets the name for the log file, the logging verbosity level and the kind of logging (e.g. log4j).

Parameters: *logFile, logLevel, logType* (Map)

Responses: none

Note: In releases prior to V2.5, all agents must receive set_log message first otherwise they will log (write) into System.out. In later releases agent log settings are included in AgentManager.cfg file.

register_service (Config file only)

Function: Registers this agent as a service provider agent for the given services.

Parameters: *service* (String or List) the service(s) this agent wants to become a service provider for.

Responses: none

Note: If this message is not read from a config file, a security violation is assumed and the message is ignored.

load_object (Config file only)

Function: Instantiates and configures an object of the given class to be stored in this agent's *dynamicObjectController*. Whether or not a configuration is to be performed depends on whether the new object implements `com.ls.util.objectcontrol.Iconfigurable or not`. If the new object implements `com.ls.util.objectcontrol.Iconfigurable`, method `configure() is called` and a *configuration* object passed to it as a parameter.

In the end the new object is stored in the *dynamicObjectController* under the given key.

Parameters: *class* (String): name of the class to instantiate

key (String): access key for the new object (needed to retrieve it from the dynamicObjectController)

configuration (optional, Map or String):

(if instance of Map): object-dependant XML structure used for the new object's configuration

(if instance of String): A config file containing the new object's configuration is used.

Responses: none

---

define_object (Config file only)

Function: Defines an object of the given class. A new instance of the defined object can then be retrieved with the *DynamicObjectController:getInstanceFromDefined()* method from this agent's *dynamicObjectController*. Whether or not a configuration is to be performed after an instantiation depends on whether the new object implements `com.ls.util.objectcontrol.Iconfigurable or not`. If the new object implements `com.ls.util.objectcontrol.Iconfigurable`, the `configure()` method is invoked and a configuration object passed to it as a parameter.

Parameters: *class* (String): name of the class to define/instantiate

key (String): access key for the object definition (needed to retrieve it from the dynamicObjectController)

configuration (optional, Map or String):

(if instance of Map): object-dependant XML structure is used for the new object's configuration.

(if instance of String): config file containing the new object's configuration.

Responses: none

set_run_level

Function: Sets the runlevel of the agent to a new value. The new runlevel can be set to TERMINATED, STOPPED or RUNNING.

Parameters: *runLevel (*String*)*

Responses: 'run_level_set' agent's run level is changed.

run_level_not_set agent's run is changed.

Both replies contain a map with:

agentName: name of the agent who requested to change the run level.

requestedRunLevel: requested run level or "lacking" if the request cannot be interpreted.

previousRunLevel: previous run levels prior to the run level change request.

currentRunLevel: Current run level following the receipt and processing of the change run level request.

ping

Function: Checks whether the agent is alive. If it is, it will reply with a "pong" message.

Parameters: *none*

Responses: 'pong' Agent is alive and working. Response message content: map with following entries

time, the current time

*location, t*he platform this agent does currently live on

not_understood

Function: Writes a warning log to the system log that the agent did not interpret the current message. Check for spelling errors in the message services or for a wrong content if you encounter this.

Parameters: all fields of a Message.toMap() of the message that was not understood plus the following fields:

reasonOfFailureText: Textual description of the reason for the failure

reasonOfFailureCode: code of the failure (see *QualityOfService*

Responses: none

delivery_failed

Function: Writes a warning log to the system log that a message for an agent could not be interpreted because the agent could not be reached.

Parameters: all fields of a Message.toMap() of the message that was not deliverable plus the following fields:

reasonOfFailureText: textual description of the reason for the failure.

reasonOfFailureCode: code of the failure (see *QualityOfService*).

routeFailedAt (optional): messenger or component, where the failure occurred.

Responses: none

access_denied

Function: Writes a warning log to the agent's log file, that a message for an agent could not be interpreted because the recipient denied the interpretation.

Parameters: all fields of a Message.toMap() of the message that was requesting the particular service plus the following fields:

reasonOfFailureText: contains a request denial message.

reasonOfFailureCode: code of the failure (see *QualityOfService*)

Responses: none

interpretation_successful

Function: Writes an info log entry to the agent's log file, that a message for another agent was interpreted correctly.

Parameters: all fields of a Message.toMap() of the message that was successfully interpreted plus the following fields:

reasonOfFailureText: contains an "interpretation returned true" text description.

*reasonOfFailureCode: code of failure (*see *QualityOfService)*.

Responses: none

get_revision_information

Function: Reads the revision information of the agent and sends a reply message with that revision information.

Parameters: *none*

Responses: 'revision_information' the revision information of the current agent.

log_inbox

Function: Reads the messages of the agent from the inbox and writes the messages into the log file of the agent.

Parameters: *none*

Responses: none

Note: It is advisable to set this message's priority to SYSTEM_PRIORITY.

set_pki_environment

Function: Sets the Public Key Infrastructure (PKI) depending on the given parameters.

Parameters: *pki_handler* specifies which PKIHandler class shall be used

pki_provider specifies which IPKIUtils-implementation shall be used (only known implementation is com.ls.pki.BaltimorePKIUtils)

Responses: none

Note: This is the base for all following PKI-related messages, because the PKI-classes are dynamically instantiated through this message.

set_private_key

Function: Sets the private key for the Public Key Infrastructure (PKI) to sign or encrypt contents.

Parameters: *keyfile_name* specifies the name of the private key file.

Location specifies the file-location of the private key file.

Password specifies the password of the private key file (this field is mandatory).

Responses: none

set_own_public_key_certificate

Function: Sets the own public key certificate for the Public Key Infrastructure (PKI). The own public key is not really used, but there might be reasons to distribute it.

Parameters: *keyfile_name* specifies the name of the public key certificate file.

Location specifies the file-location of the public key certificate file.

Responses: none

Note: The public key certificate file has to be PEM or DER formatted. Other formats are not supported!

set_foreign_public_key_certificate

Function: Sets a public key certificate for the Public Key Infrastructure (PKI) of a foreign platform.

Parameters: *keyfile_name* specifies the name of the public key certificate file

Location specifies the file-location of the public key certificate file.

keyID the ID to identify this foreign public key certificate

Responses: none

Note: The public key certificate file has to be PEM or DER formatted. Other formats are not supported!

set_pki_messages

Function: Sets one or more messages, which have to be signed, verified, encrypted, decrypted, signed and encrypted, or decrypted and verified.

Parameters:

Service: the service, for which a PKI-action has to be done.

Action: the action, which has to be carried out.

sending actions: sign, encrypt, sign_encrypt.

receiving actions: decrypt, verify, decrypt_verify.

data_key: only used for sending actions. It specifies, which message data-keys should be handled. If no data-key is specified, all content-keys of the message will be handled.

Responses: none

Example: An example to illustrate how to construct set_pki_messages messaged is show below. It is to be noted that the message entity in this example is child of a content entity that is a child of an outer message entity.

```
<message>
    <service>name of service</service>
    <action>name of action</action>
    <data_key>data_key1</data_key>
    <data_key>data_key2</data_key>
    <data_key>.........</data_key>
</message>
```

check_registered_services

Function: For monitoring the set PKI services (set_pki_messages). Logs the registered PKI services of this agent into the log file and returns a message "registered_services" with this information.

Parameters: none

Responses: 'registered_services'

signature_not_valid

Function: Used from class PKIHandler to redirect messages with invalid signatures. This method logs the message to the agent's log file.

Parameters: keys created by method toMap() of class Message.

verify_message: a detailed trace-message.

Responses: none

startup_constraint

Function: Defines interdependencies between the current agent and one or more other agents in the start-up phase. This message tells the current agent to wait for a given set of specified agents. As soon as all awaited agents are available on the local platform, the set of specified messages will be interpreted.

Parameters: *awaitedAgent* one or more agents to wait for, each single agent's name is specified within an 'awaitedAgent' tag.

MESSAGE the messages to be interpreted when all awaited agents are available on the platform.

Responses: none

Example:

```
<MESSAGE>
    <service>startup_constraint</service>
    <content>
        <awaitedAgent>AgentA</awaitedAgent>
        <awaitedAgent>AgentB</awaitedAgent>
        <MESSAGE>
            <service>do_this_or_that_first</service>
            <content>
                [message specific content goes here]
            </content>
        </MESSAGE>
        <MESSAGE>
            <service>do_another_task_afterwards</service>
            <content>
                [message specific content goes here]
            </content>
        </MESSAGE>
    </content>
</MESSAGE>
```

agent_connected

Function: Notifies the current agent that another agent is available on the local platform.

Parameters: *Agent's name* (String)
The string message containing the agent's name.

Responses: none

notified_agent, notify_canceled, notify_not_canceled, wake_up

Function: These messages are captured without functionality.

If your agent cooperates with AgentTimer, it should implement the methods:

interpretNotifiedAgent(),

interpretNodifyCanceled() and

interpretNotifyNotCanceled(), in order to deal with those services.

Parameters: none

Responses: none

send_as_configured (Config file only)

Function: Forwards one or more messages as being read from this message's content. For more flexibility in configuration the messages can contain $a in order to reference this agent's name

Parameters: one or more MESSAGES

Responses: none

Note: This message can be send only by setting up this agent's config file.

Config File

The AgentTemplate itself does not require a config file. For the PKI related tasks the PKI.cfg should be used to configure the required settings.

For handling config files, AgentTemplate comes with a config file reader that fills the *configHash* member of the class and runs the interpret() method for every message entry in the config file.

Note: Although the LARS agents can be configured with XML messages, the default config handler does not understand XML attributes. This means that you cannot use attributes when you set up configuration files for your agents. If you want to do so, you have to call the method *setIXMLHandler()* from within the constructor of your agent, passing it an instance of `com.ls.util.xml.XMLComplexHandler`, which is capable of processing XML attributes. For detailed information of this class, please consult the API documentation.

4.2.2 AgentManager

Child of: AgentTemplate

Related to: all other agents

Functionality: AgentManager is available on every LARS platform. It is responsible for creating and deleting agents from the platform, listing them and can serialize or rebuild serialized agents.

Incoming Messages

You can send messages with the following services to AgentManager. The required Input parameters are listed with the messages.

start_agent

Function: Creates a new agent for the specified *class* with the given *name,* which gets configured by *cfgFile*. In contrast to the new_agent message this method doesn't send any response. By specifying the *loadBalancing* parameter and passing a value >1, it is possible to occupy the load balancing functionality

Parameters: *class, name, cfgFile and loadBalancing* (Map)

cfgFile The configuration file or files for the agent.

When there is more than one configuration file, every file name is given an extra cfgFile tag. It is also possible to use the source tag, in which case the settings will only be valid for the specified configuration file. However it should be noted that the global source

settings would be overwritten. See below for a detailed explanation.

class The full class of the agent.

name The agent's name. The naming conventions should always be adhered to e.g. AgentTimer should always be named **at**.

loadBalancing There are two tags for loadbalancing i.e. 'minPoolSize' and 'maxPoolSize'. They both have to be numeric. The pool size has to be between '1' and '127'.

Example:

```
<loadBalancing>
        <minPoolSize>1</minPoolSize>
        <maxPoolSize>5</maxPoolSize>
</loadBalancing>
```

source The location of the configuration files. It is very useful for loading from any URL. Some additional tags are as follows: 'protocol' (file, http, ftp), 'url' any valid url without the protocol, 'path' the path from which to get the files, 'archive' the jar/zip file to load from, 'name' the name of the file, 'port' the port to connect to

All these settings are optional, but if you want to load from a URL the protocol and URL parameters are mandatory.

Example:

```
<source>
        <protocol>http</protocol>
        <url>ls500.living-systems.de</url>
        <archive>lars.jar</archive>
</source>
```

Responses: none

new_agent

Function: Creates a new agent for the specified agent *class* with the given *name*. Another agent usually sends this message to AgentManager and then the agent manager will send a reply message to indicate if the creation was successful or unsuccessful.

Parameters: *class, name and cfgFile* (Map, see start_agent)

Responses: ‘agent_created’ the agent was successfully created and inserted into the platform.

‘type_not_found’ the agent could not be built, probably due to a class not found error.

‘agent_present’ an agent with this name is present, choose another name.

reload_agent

Function: Creates a new agent for the specified agent *class* with the given *name*, and reloads the agent class. Another agent usually sends this message to AgentManager and then agent manager will send a reply message to indicate if the creation was successful or unsuccessful.

Parameters: *class, name and cfgFile* (Map, see above)

Responses: ‘agent_created’ the agent was successfully created and inserted into the platform

‘type_not_found’ the agent could not be built, probably due to a class not found

‘agent_present’ an agent with this name is already present, choose another name

restart_agent

Function: Restarts an agent that was originally created by the agent manager and is not currently running on the platform.

Parameters: *agentName* (String)

Responses: ‘agent_not_build’ the agent was not built because the generation parameters (class name, config file(s)) were unknown.

‘agent_created’ the agent was successfully created and inserted into the platform.

‘type_not_found’ the agent could not be built, probably due to a ClassNotFoundException.

'agent_present' an agent with this name is already present, choose another name

delete_agent

Function: Deletes the agent *agentName* from the platform by first running the *terminateAction()* method and then killing the agent (see below).

Parameters: *agentName* (String)

Responses: 'terminating_agent' the agent is being deleted.

'delete_failed' the agent could not be deleted.

agent_not_present an agent with this name does not exist on the platform.

kill_agent

Function: Deletes the agent *agentName* from the platform without running the *terminateAction()* method first.

Parameters: *agentName* (String)

Responses: 'agent_killed' the agent was successfully killed off.

'agent_not_killed' the agent could not be deleted.

restart_group

Function: Restarts all the agents within the group requested by the Sender that were originally created by the agent manager.

Parameters: *group name* (String)

Responses: 'restart_group_complete' all the agents within the group could be restarted.

'restart_group_partially' only some of the agents within the group could be restarted, the message informs the sender about the status of each agent.

‘restart_group_failed’ all the agents within the group could not be restarted. This message contains a special "description" field, which holds the reason for which the group couldn't be restarted.

delete_group

Function: Deletes all the agents within the group that were initially created by this agent manager by first running the *terminateAction()* method and then killing them.

Parameters: *group name* (String)

Responses: **‘**delete_group_complete’ all the agents within the group have been deleted

‘delete_group_partially’ only some of the agents within the group could be deleted, the message also informs the sender about the status of each agent.

‘delete_group_failed’ all the agents within the group could not be deleted. This message contains a special "description" field, which holds the reason for which the group couldn't be deleted.

kill_group

Function: Deletes all the agents within the group that were once created by the agent manager without firstly running the *terminateAction()* method.

Parameters: *group name* (String)

Responses: **‘**kill_group_complete’All the agents within the group have been killed

‘kill_group_partially’ Only some of the agents within the group could be killed, the message informs the sender about the status of each agent.

‘kill_group_failed’ all the agents within the group could not be killed. This message contains a special "description" field, which holds the reason for which the group couldn't be killed.

serialize_agent

Function: Serializes an agent, i.e. converts the entire agent object into a string object, which can then be sent over any data connection. To serialize an agent, the agent will be stopped and then converted into a string. The string is then sent back to the message's originator after which the agent is removed from the platform.

Parameters: *agentName* (String)

Responses: 'serializing_agent' the agent is being serialized

'serialized_agent' the agent was successfully serialized. This message (Map) contains two elements: the name (*name*) and the generated object string (*code*) for the serialized agent.

'serialize_failed' serialization could not be performed because the agent's *terminateAction()* did not return successfully.

'agent_not_present' an agent with this name does not exist on the platform

make_agent

Function: Takes a serialized agent, builds a new agent from the string object and injects it into the platform. The agent is started on the platform as usual.

Parameters: *name, code* (Map, see above)

Responses: 'agent_build' the agent was properly built and started

'class_not_found' the agent class could not be decoded or rebuilt, probably due to a missing class or a wrong class path.

'agent_not_build' the agent could not be built, try another name

start_migration

Function: Prepare the mobile agent for migration, serialize it, and send it to the destination platform

Parameters: *destination platform* (String)

Responses: 'migrating_agent' the preparation and serialization for the moving of the agent was successful.

'migration_failed' the agent couldn't be moved to the destination platform.

migrating_agent

Function: Takes the received serialized agent and builds another agent from the received information.

Parameters: *code of the agent* (String)

Responses: 'start_up' the de-serialized agent receives a message with which it tries to restart itself.

'migration_successful' inform the sender of the agent that the migration of the agent was successful.

'migration_failed' inform the sender of the agent that the migration of the agent has failed.

present_agent

Function: Checks whether the agent exists on the platform (this does not mean it is alive or working, only that it exists!)

Parameters: *agentName* (String)

Responses: 'agent_available' Agent exists

'agent_not_available'Agent does not exist

list_agents

Function: Creates a list of all agents and puts the list into a map. The map's keys are an enumeration of items in the style of "agent0, agent1, agent2, agent3…"

Parameters: *none*

Responses: 'list_of_agents' the content of the response are the listed agents.

get_version

Function: Get information about the build number and build date of the running platform

Parameters: *none*

Responses: 'version' String containing the current version of the platform.

set_lars_administrator (Config file only)

Function: Configures the user and password to be used to identify a LARS administrator (The LARS Administrator is someone who is allowed to shutdown the platform, see "shutdown_platform").

Parameters: *administrator, password* (Map)

Responses: none

Note: This message is interpreted only if it was obtained from a config file!

shutdown_platform

Function: Sends a request to AgentManager to shutdown the LARS platform. The values for the administrator and password have to be correct, see "set_lars_administrator".

Parameters: *administrator, password* (Map)

Responses: 'shutdown_platform_in_progress' if administrator and password were correct.

'shutdown_platform_declined' if administrator and password were not correct.

Config File

The config file for the AgentManager is usually called AgentManager.cfg and is located in the conf/lars directory of the platform installation. It contains the messages, to be sent to AgentManager during the startup phase of the platform, most importantly it contains the messages to start up other agents.

A very simple config file would look like this:

```
<CONFIG>
    <MESSAGE>
        <service>start_agent</service>
        <content>
            <class>com.ls.lars2.AgentMessageRouter</class>
            <name>amr</name>
            <cfgFile>lars/$c.log</cfgFile>
            <logFile>$a.log</logFile>
            <logLevel>INFO</logLevel>
            <logType>LOG4J</logType>
        </content>
    </MESSAGE>
</CONFIG>
```

The config file for AgentManager is quite easy to create.

The XML values within the message in the config file will form the message AgentManager receives. The requested service is **start_agent** and the content of the message consist of a Map with the following entries: *class, name, cfgFile* and the logging configuration. Agent config files can be referred to as a textual dump of normal LARS messages.

4.2.3 AgentMessageRouter

Child of: AgentTemplate

Related to: all agents that send messages

Functionality: This class is the wrapper agent for the MessageRouter class. Almost every agent sends AgentMessageRouter a register message in its initialization phase for it to become registered in the local platform's message router. If an agent terminates, it sends a sender_rip service message AgentMessageRouter can also be used for the following purposes:

Registering and deregistering (unregistering) service providers.

For maintaining message groups to which agents can be subscribed to and un-subscribed from.

AgentMessageRouter can be used by migrated agents to inform their home platform's message router of their new location.

Note: AgentMessageRouter logs ERROR and WARNING information in both the system's log file and it's own log file; all log messages with a less important log level are logged only in AgentMessageRouter's own log file.

Incoming Messages

AgentMessageRouter offers the following services. Required input parameters you have to specify are listed with the messages.

notify_agent_connected

Function: Instructs the AgentMessageRouter to notify the sender of this message as soon as the *awaitedAgent*(s) got connected to the local LARS platform.

Parameters: *awaitedAgent* (String or Map)

Responses: 'agent_connected' this response is sent as soon as the a*waitedAgent(s)* have been connected to the local platform. The content of this message is the name of the connected agent.

agent_connected (internal use only!)

Function: Notifies the AgentMessageRouter that the *connectedAgent* is available on the local platform. When AgentMessageRouter receives this message, it will notify all registered agents, who want to be informed about the availability of the *connected Agent*.

Parameters: *connectedAgent* (String)

Responses: none

Note: AgentMessageRouter interprets this message for internal use, only. Therefore, it checks the sender of this message in order to prevent any misuse of it.

notify_service_status_changed

Function: Registers the sender of this message at the AgentMessageRouter for being notified, when the status of a defined service has changed (i.e. a service provider registers or un-registers for that service).

Parameters: *awaitedService* (String)

Responses: 'service_status_changed' every time the status of that service has changed. Content of this message is: (i) the name of the service; (ii) the string "INCREASED" or "DECREASED", depending whether a service provider did register or unregister; (iii) the actual number of service providers for that service.

Note: the message router can only send the 'service_status_changed' response message. If you want to prevent the misuse of this message by other parties, you should ensure the message's sender to be *LARS_INTERNAL*.

initialize_inbox_check

Function: Registers this agent to the *notifierAgent* (mostly AgentCron) for being notified regularly, in order to check the inbox size of all connected messengers. This check implies a warning, which will be written to the System Logger when the size of any messenger's inbox exceeds the given *delimiterSize*.

The notification *period* is set up as in AgentCron.

Parameters: *notifierAgent* (String)

delimiterSize (String)

period (Map) (compare to AgentCron)

To configure the inbox_check correctly, you should place the following message into the config file of AgentMessageRouter (the period definition can be changed to your needs):

```
<MESSAGE>
```

```
    <service>startup_constraint</service>
    <content>
        <awaitedAgent>AgentCron</awaitedAgent>
        <MESSAGE>
            <service>check_inbox_size</service>
            <content>
                <delimiterSize>3</delimiterSize>
                <notifierAgent>AgentCron</notifierAgent>
                <period>
                    <type>INTERVAL</type>
                    <minute>10</minute>
                    <hour>0</hour>
                    <day>0</day>
                </period>
            </content>
        </MESSAGE>
    </content>
</MESSAGE>
```

Responses: none

check_inbox_size

Function: Tells AgentMessageRouter to check the inboxes of all connected messengers. Normally, this message is read from the *notifierAgent* configuration file.

Parameters: none

Responses: none

agent_not_notified

Function: Informs the AgentMessageRouter that its notification request per notify_agent (normally being sent due to an incoming 'initialize_inbox_check' message) has not been processed properly.

Parameters: none

Responses: none

register

Function: Tells the LARS platform's message router to forward any messages to the message router agent as specified in the message parameter. The *newMessageRouter* is

the global name of the agent message router, which will handle the sender messages.

Parameters: *newMessageRouter* (String)

Responses: none

unregister

Function: Removes a forwarding entry from the platform's message router.

Parameters: *none*

Responses: none

register_service

Function: sender wants to receive a copy of all service messages.

Parameters: service (String) or services (List); The service(s) this agent wants to become a service provider for.

Responses: none

Note: If multiple services are specified (message content is a List), the agent is registered as a service provider for all of the specified services. Naturally, the sender will not be registered as a provider of any of the specified services if the message cannot be processed successfully.

unregister_service

Function: The sender doesn't want to receive a copy of all service messages any more.

Parameters: service (String) or services (List); the service(s) that are no longer provided by the sender.

Responses: none

unregister_all_services

Function: sender doesn't want to receive a copy of any service messages.

Parameters: none

Responses: none

subscribe_to_message_group

Function: subscribes a single agent or a list of agents to the specified message group. If the group does not exist, a new group is created. All messages being sent to this group will be passed to all subscribed agents.

Parameters: *group* (String), the name of the group to subscribe to.

agent (String or Collection). The agent(s) that should be subscribed to that group.

transaction (String):

on (default): subscribe all agents or none of them

off: subscribe as many agents as possible

Responses: **'**subscribe_to_message_group_succeeded' message is dispatched as a response if all concerned agents are subscribed. The content of the response message is the same as the content of the request message.

'subscribe_to_message_group_failed' message is dispatched as a response if all concerned agents cannot be subscribed. The content of the response message is the same as the request message.

'subscribe_to_message_group_partially' response message is sent when the request message's transaction field is set to 'off' and partial subscription cannot be achieved. In addition to the request message's content, this response introduces two new keys/value pairs:

succeeded: value is a collection of names of agents that are now subscribed to the message group.

failed: value is a collection of names of agents that could not be subscribed to the message group.

Note: If multiple agents are specified (value of key 'agent' in message content is a Collection), either all of those agents are added to the specified message group or none of them.

unsubscribe_from_message_group

Function: Unsubscribes a single agent or a list of agents from the specified message group. All empty message groups will be removed from the message group's list.

Parameters: *group* (String), the name of the group to unsubscribe from.

agent (String or List). The agent(s) that should be unsubscribed.

Responses: 'unsubscribe_from_message_group_succeeded' if all agents can be successfully unsubscribed.

'unsubscribe_from_message_group_failed' if all agents cannot be successfully subscribed.

Content of the above response is that of the original request message.

unsubscribe_from_all_message_groups

Function: Unsubscribes a single agent from all message groups that the agent is subscribed to. All empty message groups will be removed from the message group's list.

Parameters: *agent* (String). The agent that should be unsubscribed.

Responses: 'unsubscribe_from_all_message_groups_succeeded' if the agent can be successfully unsubscribed from all groups.

'unsubscribe_from_all_message_groups_failed' if the agent cannot be successfully unsubscribed from all groups

Content of the above responses is that of the original request message.

remove_message_group

Function: removes a message group from the list of message groups.

Parameters: *group* (String). The name of the group that should be removed.

Responses: ‘remove_message_group_succeeded’
if the message group can be removed successfully or if the message group does not exist.

‘remove_message_group_failed’
if the message group name is malformed.

Content of the above responses is that of the original request message.

close_connection

Function: AgentMessageRouter will instruct the underlying message router to send a 'close_connection' message to all connected messengers.

Parameters: *none*

Responses: none

Note: This message can be sent either by AgentManager, or by the message router. All other originators of this message will be ignored.

sender_rip

Function: Before an agent dies, it should send this message to request that all references pointing to that agent should be removed (service registrations, message groups, forward, agent connect-notifications)

Parameters: *none*

Responses: none

log_system_information

Function: logs all known services, message groups and platforms.

Parameters: *none*

Responses: none

ask_group_members

Function: asks for members of one or multiple message groups.

Parameters: Map with one key *group*: value is a String or a List of Strings specifying the name of the group(s) the sender is interested in.

Responses: **'**response_group_members'

map content contains the key "*group*" and as value one map or a list of maps (one for each group of the request) with the following keys:

name specifying the group name

agent specify the subscribed agent(s) (String or List)

errorerror holding *ILarsConstants.NO_GROUP_AVAILABLE*, if the group does not exist or is empty.

ask_service_provider_members

Function: asks for members (service providers) of one or multiple message services.

Parameters: Map with one key "*service*": value is a String or a List of Strings specifying the name of the services(s) the sender is interested in.

Responses: **'**response_group_members message'

map content contains the key "*service_provider*" and as value one map or a list of maps (one for each service of the request) with the following keys:

name → specifying the service name

agent → specifying the service provider agent(s) (String or List)

error → holding *ILarsConstants.NO_SERVICE_PROVIDER_AVAILABLE*, if the service is unknown or no service providers for it are registered.

show_all_provided_services

Function: asks for a list of services, which are provided on the local LARS platform.

Parameters: *none*

Responses: **'**provided_services message'

map content, which includes the key "service" and a List with the names of all provided services as value.

Config File

AgentMessageRouter does not have a config file by default. However, it should register the log_system_information service (with help of a register_service message in its config file) if system information should be available in the logfile.

The following example shows how to register log_system_information:

```
<MESSAGE>
    <service>register_service</service>
    <content>log_system_information</content>
</MESSAGE>
```

When the message router inspects the inbox size of all connected messengers, the config file of AgentMessageRouter must contain the correct settings for sending the required 'initialize_inbox_check' message. As the initialization of the inbox checks sends a 'notify_agent' to the given notifier agent, the config file of AgentMessageRouter should contain an adequate 'startup_constraint' message (compare to AgentTemplate).

4.2.4 AgentPlatformSecurity

Child of: AgentSynchronization (see 4.3.10)

Related to: All components that are concerned with remote parties connection to the local platform.

Functionality: The LARS platform synchronization process allows the inter-connection of LARS platforms in order to enable cross-platform messaging. Lars platforms can be configured to 'know' remote platforms and to connect to them at runtime.

For instance, 141.168.1.1-lars1 can be configured to connect to 141.168.1.1-lars3, which enables the agents on both platforms to inter-communicate with each other.

Platform synchronization can also establish connections dynamically; a LARS platform, which connects to a remote platform, can be automatically informed of other platforms that are currently connected to that remote platform. This notification process depends on the kind of connection type that exists between the various platforms.

There are 3 kinds of connection types:

- public: all publicly known platforms and own platform connection parameters are send to the publicly connected remote platform.
- private: all publicly and privately known platforms and own platform connection parameters are send to the privately connected remote platform.
- hidden: only own platform connection parameters are sent to a remote platform with a hidden connection.

The following is an example to illustrate how platform connections are passed.

platform1 is connected to platform2a using a “hidden” connection, and to platform2b using a “public” connection. Now, platform3 opens a “private” connection to platform1. In this case, platform3 will be informed on the connection parameters of platform2b, which enables it to connect to the platform directly. The platform3 will not receive the connection parameters of platform2a, as this platform is 'hidden' by platform1.

AgentPlatformSecurity controls what information will be sent to a connecting platform. The information that is sent during platform connection will always contain the connection parameters of the local

platform and additional connection parameters depending on the connection type as explained above.

Furthermore, AgentPlatfromSecurity manages the security settings regarding the client-to-lars- and inter-lars-communication. The agent provides possibilities to define: trusted and untrusted parties - on the basis of the following settings (to be configured within the agent's config file):

- trusted and/or untrusted agents
- trusted and/or untrusted LARS platforms
- trusted and/or untrusted IP addresses

You can define the security of your local LARS platform to a very fine-grained level. For instance, you could specify that all agents coming from platform1 are not allowed to connect, but AgentX and AgentY of the very same platform are.

Note: By default your local LARS platform is absolutely locked. This means that connection to the local platform from the outside is prohibited by default. This default locking mechanism is deliberate in order to encourage living markets users to think about security from the onset and to only permit access to trusted agents, platforms or IP addresses.

Incoming Messages

You can send messages with the following services to AgentPlatformSecurity. Message parameters are listed with the messages and must be transmitted using a map.

set_trusted_agent (Config file only)

Function: Adds agent name(s) to the list of trusted agents.

Parameters: Map with key "*agent*" and with a String or Collection denoting the trusted agent name(s) as value(s).

Responses: none

set_untrusted_agent (Config file only)

Function: Adds agent name(s) to the list of untrusted agents.

Parameters: Map with key "*agent*" and with a String or Collection denoting the untrusted agent name(s) as value(s).

Responses: none

---

set_trusted_platform (Config file only)

Function: Adds platform name(s) to the list of trusted platforms.

Parameters: Map with key "*platform*" and with a String or Collection denoting the trusted platform name(s) as value(s).

Responses: none

---

set_untrusted_platform (Config file only)

Function: Adds platform name(s) to the list of untrusted platforms.

Parameters: Map with key "*platform*" and with a String or Collection denoting the untrusted platform name(s) as value(s).

Responses: none

---

set_trusted_ip (Config file only)

Function: Adds the IP address(s) to the list of trusted IP addresses.

Parameters: Map with key "*ipAddress*" and with a String or Collection denoting the trusted IP address name(s) as value(s).

Responses: none

---

set_untrusted_ip (Config file only)

Function: Adds the IP address(s) to the list of untrusted IP addresses.

Parameters: Map with key "*ipAddress*" and with a String or Collection denoting the untrusted IP address name(s) as value(s).

Responses: none

response_for_connection_parameters

Function: Local listener agents send this message to request for their connection parameters.

Parameters: Map with key 'port' and with a String denoting the port the sender of this message (a listener) is listening on.

Responses: none

register_platforms (Internal use only)

Function: Registers the platforms given by this message with the local LARS platform.

Parameters: List containing one or more *AgentSynchronization.LarsPlatformRepresentation* entries.

Responses: none

Note: This message is used for the platform synchronization process, thus it can only be sent by this platform's *AgentSynchronizeSupervisor* or *AgentSynchronizeConnectionHandler*. register_platforms messages that are sent by other originators will be ignored.

unregister_platforms (Internal use only)

Function: Unregisters the platforms given by this message from the local LARS platform.

Parameters: List containing one or more *AgentSynchronization.LarsPlatformRepresentation* entries.

Responses: none

Note: This message is used for the platform synchronization process, thus it can only be sent by this platform's *AgentSynchronizeSupervisor* or *AgentSynchronizeConnectionHandler*. unregister_platforms messages that are sent by other originators will be ignored.

inform_remote_platforms (internal use only!)

Function: This message is a request to inform all platforms as indicated in the message of the connection settings of all other connected platforms. The extend of the connection settings information that is passed depends on the connection type.

Parameters: List containing one or more *AgentSynchronization.LarsPlatformRepresentation* entries.

Responses: none

Note: This message is used for the platform synchronization process only, thus it can only be sent from this platform's *AgentSynchronizeSupervisor* or *AgentSynchronizeConnectionHandler*. inform_remote_platforms messages that are sent by other originators will be ignored.

Config File

The config file for this agent is usually called AgentPlatformSecurity.cfg and is located in the conf/lars directory of the platform installation. AgentPlatformSecurity.cfg describes the trusted/untrusted status of all agents, platforms and IPs.

Example:

```
<CONFIG>
   <MESSAGE>
      <service>set_trusted_ip</service>
      <content>
         <ipAddress> </ipAddress>
      </content>
   </MESSAGE>
   <MESSAGE>
      <service>set_trusted_platform</service>
      <content>
         <platform> </platform>
      </content>
   </MESSAGE>
</CONFIG>
```


Note: When specifying trusted and untrusted agents, platforms and/or IP addresses, a `connectionType` can be included.

There are two permitted `connectionType entities`:

- an asterisk "*": to indicate that the trusted/untrusted setting will be valid for all connection types.
- a connection type (e.g. rmi, socket or http): to indicate that the trusted/untrusted setting will only be applicable to connection type as specified.

It has to be noted that the default `connectionType` is '*'.

4.3 Service Agents

4.3.1 AgentTimer

Child of: AgentTemplate

Related to: any agent that requires a timing functionality.

Functionality: AgentTimer can send "wake up" calls to other agents after a specified time.

Note: AgentTimer logs ERROR and WARNING log information in both the system's log file and it's own log file. All log messages with a low priority log level are logged in AgentTimer's own log file.

Incoming Messages

You can send messages with the following service to AgentTimer. Message parameters you have to specify are listed with the messages.

notify_agent

Function: Inserts a notification request into the timer queue. The request identifies an *agent* and a set *time*. The *time* value is an absolute value and not relative to the current time. On the given time the specified agent will be notified with a wake_up message call.

Parameters: *time, agent* (Map).

Responses: 'agent_notified' the notification request is added to the timer queue.

'agent_not_notified' the request is malformed (i.e. invalid time format) and is not added to the time queue. To create AgentTimer's time stamp, a value in milliseconds can be added to the value returned by Java's System.currentTimeMillis().

The responses above use the reply ID of the incoming messages to send back the original parameter hashtable, which can be saved and used to cancel a notification at a later stage.

cancel_notify

Function: Removes a notification from the timer queue.

Parameters: *time, agent* (Map)

Responses: 'notify_canceled':
The notification request is removed from the timer queue.

'notification_not_canceled': The notification cannot be removed from the timer queue because of a malformed *time* argument, or because the notification request does not exist.

The above responses use the reply ID of the incoming message to send back the original parameter hashtable.

sender_rip

Function: Removes all notification requests of the sender from the timer queue. The terminateActions() method of all agents that cooperate with AgentTimer are required to send sender_rip message.

Parameters: *none* (content is not needed and will be ignored if included).

Responses: none.

Config File

AgentTimer does not need a config file by default.

4.3.2 AgentCron

Child of: AgentTemplate

Related to: any agent that requires a timing functionality.

Functionality: Agents that require executing a certain task periodically can request a periodical notification call from AgentCron. Details of the periodical calls are parsed and internally stored into a queue.

Furthermore, clients of AgentCron can request the removal or modification of their periodical notification calls. (To modify a periodical notification call, the original request must be removed and replaced with a new request.).

Note: AgentCron is depending on the services of AgentTimer!

Incoming messages

You can send messages with the following services to AgentCron. Message parameters are listed with the messages and must be transmitted using a Map.

notify_agent

Function: Inserts a notification request into the cron queue.

Parameters: The period, agent and task to be carried out (Map).

Responses: 'notified_agent' The notification request is added to cron queue.

'agent_not_notified' The request is malformed (i.e. invalid specification of period of time) and is not added to the queue.

The above responses use the reply ID of the incoming message to send back the original parameter hashtable, which can be saved and used to cancel a notification at later stage. .

cancel_notify

Function: Removes a notification from the cron queue.

Parameters: The period, agent and task to be carried out (Hashtable).

Responses: 'notify_canceled': The notification request is removed from the cron queue.

'notify_not_canceled': The notification request cannot be removed because of a malformed specification of the time parameter or because the notification request does not exist.

Both responses use the reply ID of the incoming message to send back the original parameter hashtable.

wake_up

Function: AgentTimer informs AgentCron to send one or multiple messages to its clients (agents that placed notification requests).

Parameters: none (content is not needed and will be ignored if included).

Responses: The message that the agent requested to be notified at the specified notification time. It should to be noted that a response to a wake up call is not generic.

sender_rip

Function: Removes all notification requests of the sender from the cron queue, because the sender has just died.

Parameters: none (content is not needed and ignored).

Responses: none.

Config File

AgentCron does not need a config file, however if supplied it can contain any of the above notification request(s).

The content of a notify_agent message must have the following format:

```
<content>
    <period>
        <type>          </type>
        <minute>        </minute>
        <hour>          </hour>
        <day>           </day>
        <day_of_month>  </day_of_month>
    </period>

    <MESSAGE>
        <service>       </service>
        <receiver>      </receiver>
        <content>       </content>
    </MESSAGE>
</content>
```

The possible values for the tag "type" are: *CRON*, *INTERVAL* and *TIME*.

CRON: specifies the exact period in time the notification has to be sent

"minute" can be an integer in the range of (0 to 59) or "*****".

"hour" can be an integer in the range of (0 to 23) or "*****".

"day" can be one of the days of the week in English or the"*****".

"day_of_month" can be one of the days of the month an integer between 1 and 31 or "*".

Note:

(7) The char "*" denotes "every" (e.g. if the minute field is set to '*', a notification call will be sent out every minute of the specified period).

(8) The day or the day_of_month fields must be set. Setting both fields in the same message is permitted. .

Example: To notify "agent_A" with the message "an_example_notify" on the third of every month when the 3rd is a Monday at 5 minutes pass every hour.

```
<content>
    <period>
        <type>CRON</type>
            <minute>5</minute>
            <hour>*</hour>
            <day>MONDAY</day>
            <day_of_month>3</day_of_month>
    </period>

    <MESSAGE>
        <service>an_example_notify</service>
        <receiver>agent_A</receiver>
        <content> </content>
    </MESSAGE>
</content>
```

INTERVAL: specifies interval of time between notifications

- "minute" - can be any positive integer => 0.
- "hour" - can be any positive integer => 0.
- "day" - can be any positive integer => 0.

Note: 1. The value 0 is the neutral value.

2. The "day_of_month" field is not permitted when an interval is specified.

Example: To notify "agent_A" with the message "an_example_notify" at an interval of 2 days and 5 minutes.

```
<content>
    <period>
        <type>INTERVAL</type>
        <minute>5</minute>
        <hour>0</hour>
        <day>2</day>
    </period>

    <MESSAGE>
```

```
        <service>an_example_notify</service>
        <receiver>agent_A</receiver>
        <content> </content>
    </MESSAGE>
</content>
```

TIME: specifies a point in time when the notification is required

- "minute" - can be an integer in the range of (0 to 59).
- "hour" - can be an integer in the range of (0 to 23).
- "day" - can be one of the days of the week in English.
- "day_of_month" - can be one of the days of the month as an integer between 1 and 31.

Note:

1. A **TIME** notification request is satisfied only once!
2. The day or the day_of_month filed must be set. Setting both fields in the same message is permitted.

Example: To notify "agent_A" with the message "an_example_notify" on Monday at 00:05 hours.

```
<content>
    <period>
        <type>TIME</type>
        <minute>5</minute>
        <hour>0</hour>
        <day>Monday</day>
    </period>

    <MESSAGE>
        <service>an_example_notify</service>
        <receiver>agent_A</receiver>
    <content> </content>
    </MESSAGE>
</content>
```

4.3.3 AgentLogin

Child of: AgentTemplate

Related to: any user, who wants to communicate with the LARS platform.

Functionality: This agent handles the login process pertaining JSecureSocket communication. User authentication

occurs without a database. The configuration file for this agent should contain a "set_user_list" message with users and their passwords in plain text. For security reasons, it is advised to pay a special attention to the location of AgentLog.cfg.

Incoming messages

You can send messages with the following services to AgentLogin. Message parameters are listed with the messages and must be transmitted using a map.

set_user_list (Config file only)

Function: Set a list of users, which are allowed to contact the platform via JSecureSocket communication.

Parameters: *name*, *password* (Map)

Name The name of the user.

Password The user's password.

Responses: none.

authenticate

Function: Checks whether a given user is allowed to contact the platform.

Parameters: userid, seed, public_key, hash (Map)

userid The name of the user.

seed Random number generated by the client.

public_key Client's public_key to enable an encrypted communication.

fingerprint A fingerprint including the user's password.

Responses: 'user_ok' The user is successfully authenticated.

'user_not_ok' The user is not successfully authenticated.

Config File

The config file for AgentLogin is usually called AgentLogin.cfg and is located in the conf/lars directory. Living markets customers must be aware of the fact that AgentLogin.cfg contains security critical information (user list) and an as such it must be protected adequately.

4.3.4 AgentListener

Child of: AgentTemplate

Related to: any agent, which communicates with agents on other platforms

Functionality: This agent facilitates remote parties communication with the local LARS platform.

Incoming Messages

Listed below are messages that are understood by the top abstract AgentListner class and its inherited listener subclasses. Message parameters are listed with the messages.

set_port (Config file only)

Function: Sets the port number where the agent will listen for incoming communication requests.

Parameters: *port* (String)

Responses: none

ask_for_connection_parameters

Function: Every listener provides an ask_for_connection_parameters service. A reply to

ask_for_connection_parameters service request is a 'response_for_connection_parameters' response.

Parameters: sender of the message

Responses: 'response_for_connection_parameters'
map content of the above response includes the following:

- *port* → specifies the port this listener is listening on.

- *ip-address* → specifies the ip address of the local platform.

---

set_outbox (Config file only)

Function: Sets the status of the outbox to

- true: to use outbox or

- false: for do not use the outbox

Parameters: status value for the outbox (String: true/false)

Responses: none

4.3.5 AgentSocketListener

Child of: AgentListener

Related to: any agent, which communicates with agents on other platforms.

Functionality: AgentSocketListener receives messages through a UNIX or Windows socket connection and passes them to the local platform. The messages are communicated using xml-formatted plain text.

Incoming Messages

You can send messages with the following services to AgentSocketListener. Message parameters are listed with the messages.

set_max_message_length (Config file only)

Function: Sets the maximum message length to be read from a socket.

Parameters: *message length* (String).

Responses: none.

set_compression (Config file only)

Function: Sets the compression settings to compress messages.

Parameters: *ConnectionCompressionType:*

no_compression

No compression will be used regardless of whether the auto compression is on or off.

zip:

Zip compression, if auto compression is set or a given message size is specified; a zip compression can be applied.

gzip:

GZip compression, if auto compression is set or a given message size is specified; a gzip compression can be applied.

ConnectionAutoCompression:

on:

Switches on the auto compression in order to apply the selected compression type on messages with a size greater than the given auto compression start size.

off:

Switches the auto compression off. .

ConnectionAutoCompressionStartSize: (default 16384)

Sets the auto compression message size.

connectionCompressionLevel: (0..9)

Sets the zip compression level. Zip compression allows for a fine tuned compression ranging from 0 (for no compression) to 9 (for maximum compression).

ConnectionZipEntryName

This parameter specifies the name of the ZIP file entry being created by compressing a message. This is an optional parameter and can be omitted.

Responses: none

Example:

```
<MESSAGE>
    <service>set_compression</service>
    <content>
        <connectionCompressionType>
            gzip
        </connectionCompressionType>
        <connectionAutoCompression>
            off
        </connectionAutoCompression>
        <connectionAutoCompressionStartSize>
            4096
        </connectionAutoCompressionStartSize>
        <connectionCompressionLevel>
            7
        </connectionCompressionLevel>
        <connectionZipEntryName>
            Message
        </connectionZipEntryName>
    </content>
</MESSAGE>
```

Config File

AgentSocketListener's config file usually has two entries:

set_port message:

To set the port a vacant port has to be selected to avoid conflict with other socket operations, namely the RMI listener or a web server.

set_max_message_length message:

The maximum message length is a string representation of an integer value and is used to set the maximum length of messages to be read from the socket.

Compression settings are optional and can be excluded from the AgentSocketListener's config file. However, we can distinguish two options should we choose to include compression settings:

Auto compression:

With auto compression, compression is applied if the received message size reaches a maximum size. The maximum size has a default value, which can be overridden.

A fine-tuned compression (applicable to ZIP compression only):

A fine tuned compression is possible with ZIP compression only. ZIP compression permits various levels of compression ranging from 0 for no compression to 9 for maximum compression. It has to be noted that a zip entry name is optional and can be omitted if not desired.

Notes: Messages understood by AgentSocketListener have the same XML format as the individual messages in the config files. AgentSocketListener can pass only one message at a time.

TIP: The default communication mode with AgentSocketListner is through plain XML text messages, which can be encrypted with PKI encryption if necessary PKI encryption is set in the agent's configuration file.

4.3.6 AgentJSocketListener

Child of: AgentSocketListener

Related to: any agent, which communicates with agents on other platforms.

Functionality: AgentJSocketListener receives messages through a UNIX or Windows socket connection and pass them into the platform. Messages are communicated using object serialization.

Incoming Messages

This agent offers the same services as AgentSocketListener.

Config File

This agent's config file can contain the same configuration settings as that of AgentSocketListener. A vacant port is to be selected to avoid conflict with other socket operations, namely the RMI listener or a web server.

Notes: Messages understood by AgentSocketListener have the same XML format as the individual messages in the config files. AgentSocketListener can pass only one message at a time.

TIP: The default communication mode with AgentSocketListner is through plain XML text messages, which can be encrypted with PKI encryption if necessary. PKI encryption is set in the agent's configuration file.

4.3.7 AgentJSecureSocketListener

Child of: AgentSocketListener

Related to: any agent, which needs to communicate with agents on other platforms over a secure (encrypted) communication line.

Functionality: AgentJSecureSocketListener receives messages through a UNIX or Windows socket connection, decrypt them and pass them to the local platform. Communicated messages are exchanged using object serialization.

Incoming Messages

You can send messages with the following services to AgentSocketListener. Message parameters are listed with the messages and must be transmitted using a Map.

set_login_agent (Config file only)

Function: Sets the name of the login agent, which is responsible for authenticating remote parties.

Parameters: *agent name* (String)

Responses: none

Config File

AgentJSecureSocketListener‘s config file usually has three entries:

set_port

To select a port that is not occupied by any other socket operation, namely the RMI listener or a web server.

set_max_message_length message:

To set the maximum length of messages to be read from the socket. The maximum message length is a string representation of an integer value.

set_login_agent

To specify the login agent. Care must be taken to ensure that the specified login agent really exist, otherwise the remote parties attempts to be authenticated and to connect to the local will fail.

Notes: Messages understood by AgentSecureSocketListener have the same XML format as the individual messages in the config files. AgentSocketListener can pass only one message at a time.

Attention: Encrypted object stream communication mode using synchronous encoding is the default communication mode of AgentSecureSocketListener. For increased security, the agent can be configured to apply PKI encryption.

4.3.8 AgentRMIListener

Child of: AgentListener.

Related to: any agent, which communicates with agents on other platforms.

Functionality: This class is the wrapper agent needed for the communication of a RMI listener. It waits for incoming RMI requests from remote agents.

Note: RMI (Remote Method Invocation) is a standard Java technique to remotely call methods on a server system. Further details on RMI can be found in the Sun's Java documentation.

For AgentRMIListener to work, the RMI registry (located in the JDK's bin directory) has to be started and running on the same port as the agent is configured.

Incoming Messages

This agent offers the same services as AgentSocketListener.

Config File

AgentRMIListener 's config file usually has only one entry (set_port message to set the required port as inherited from AgentListener). A vacant port must be selected to avoid conflicts with other socket operations, namely the socket listener or a web server operation.

4.3.9 AgentJMSListener

Child of: AgentListener.

Related to: any agent, which communicates with agents on other platforms.

Functionality: AgentJMSListener is the wrapper agent that enables the JMSListener communication. AgentJMSListener main purpose is to start up the JMSListener.

Note: JMS (Java Messaging Service) is a standard API for communicating with different platforms. Further details on JMS can be found in the Sun's Java documentation. For AgentJMSListener to work, a JMS server has to be started as well as the LARS platform.

Incoming Messages

You can send messages with the following services to AgentJMSListener. Message parameters are listed with the messages and must be transmitted using a Map.

set_jms_parameters (Config file only)

Function: Configures the JMS communication protocol.

Parameters The message should contain the following:

java.naming.provider.url
The URL address of the JMS server.

java.naming.factory.initial
The *JNDI* initial context factory, usually defined by the JMS provider.

java.naming.security.principal
The security principal, which can be an empty String but must not be null.

java.naming.security.credentials
The *security credential*, which can be an empty String but must not be null.

Example:

```
<MESSAGE>
    <service>set_jms_parameters</service>
    <content>
        <java.naming.provider.url>
            localhost
        </java.naming.provider.url>
        <java.naming.factory.initial>
            org.jnp.interfaces.NamingContextFactory
        </java.naming.factory.initial>
        <java.naming.security.principal/>
        <java.naming.security.credentials/>
    </content>
</MESSAGE>
```

register_queue_pair

Function: Connects a specified receiver and a sender queue with the JMS server.

Parameters The message should contain the following:

sendingQueueName
Defining the queue name for sending JMS messages.

receivingQueueName
Defining the queue name for receiving JMS messages.

Example:

```
<MESSAGE>
  <service>register_queue_pair</service>
  <content>
    <sendingQueueName>queue/A</sendingQueueName>
    <receivingQueueName>queue/B</receivingQueueName>
  </content>
</MESSAGE>
```

unregister_queue_pair

Function: Disconnects a specified receiver and a sender queue from the JMS server. This releases the queue pair for further usage by someone else.

Parameters The message should contain the following:

sendingQueueName
Defining the queue name for sending JMS messages.

receivingQueueName
Defining the queue name for receiving JMS messages.

ask_for_connected_queue_pairs

Function: Allows others to retrieve the queue pairs, which are in use by JMS consumers.

Parameters no parameters

Response The message the sender receives contains the queue pairs. Service name: *response_for_connected_queue_pairs*

Config File

JMS communication parameters and JMS queue pairs are usually set up in AgentJMSListener's config file.

4.3.10 AgentSynchronization

Child of: AgentTemplate

Related to: AgentSynchornization is an abstract class. AgentPlatformSecurity, AgentSynchronizeSupervisor and AgentSynchronizeConnectionHandler extend AgentSynchornization.

Functionality: The required functionalities of the extending classes are defined in AgentSynchornization.

Incoming Messages

You can send messages with the following services to AgentSynchornization. Message parameters are listed with the messages and must be transmitted using a Map.

set_constants

Function: This message, which is found in the agents configuration file, contains local platform specific information that is required by all AgentSynchornization inherited classes (agents).

Parameters: own platform settings (Map)

set_constants message contains the following:

- the names of the 3 synchronization agents:

```
<AgentSynchronizeSupervisor>...
<AgentSynchronizeConnectionHandler>...
```

```
<AgentPlatformSecurity>...
```

- the names and types of the listeners running on the local platform:

```
<ownPlatformListener>
    <listenerName> ...
    <listenerType> ...
</ownPlatformListener>
```

- order of protocol, if more then one protocol to connect to other platforms is defined:

```
<preferredProtocolOrder>
    <protocolType>...
</preferredProtocolOrder>
```

- synchronization intervals. Under this tag the following two values can be specified:

attemptConnections:

The time that must lapse prior to a reconnection attempt with a lost remote platform.

CheckConnections:

The time interval for remote connections inspection:

```
<synchronizatonIntervals>
    <attemptConnections>...
    <checkConnections>...
</synchronizatonIntervals>
```

Responses: none

synchronize_platforms

Function: This message, which is to be found in the agents configuration file, contains remote platform specific information that are required by all AgentSynchornization inherited classes (agents):

Parameters: remote platform settings (Map) as described below:

- *platformId* → specifies the remote platform's name

- *ipAddress* → specifies the remote platform's ip address

- *connectionType* → specifies the used communication protocol

- *port* → specifies the port, where a protocol-specific listener listens on

- *connectionParameters* → some protocol-specific connection parametersThe xml-tags are:

```
<platform>
    <platformId>...
    <ipAddress>...
    <port>...
    <access>...
    <connectionType>...
    <connectonParameters>...
</platform>
```

Responses: *none*

Config File

AgentSynchnonization does not need a config file.

4.3.11 AgentSynchronizeSupervisor

Child of: AgentSynchronization.

Related to: agents that participate in the synchronization mechanism.

Functionality: This agent is the platform synchronization supervisor.

AgentSynchronizeSupervisor issues connection orders to establishing connections with new remote platforms. It closes properly lost connections to remote platforms and checks from time to time whether the remote platforms are still connected.

Incoming Messages

You can send messages with the following services to AgentSynchronization. Message parameters are listed with the messages and must be transmitted using a Map.

register_remote_platforms (registered as a service)

Function: A new remote platform is registered. After registration AgentSynchronizeSupervisor instructs AgentSynchronizeConnectionHandler to establish the new registered platform's connection.

Parameters: remote platform parameters (Hashtable).

Example:

```
<platform>
    <access>public</access>
    <port>7002</port>
    <ipAddress>192.168.140.104</ipAddress>
    <connectionType>socket</connectionType>
    <platformId>lars2</platformId>
</platform>
```

Responses: none

unregister_remote_platforms (registered as a service)

Function: A new remote platform is unregistered. After unregistration AgentSynchronizeSupervisor instructs AgentSynchronizeConnectionHandler to close the remote platform's connection.

Parameters: remote platform parameters (Hashtable).

Example:

```
<platform>
    <ipAddress>192.168.140.104</ipAddress>
    <platformId>lars2</platformId>
</platform>
```

Responses: none

list_platforms (registered as a service)

Function: If AgentSynchronizeSupervisor receives a list_platforms message, it responds with a list_of_platforms reply. The platform list contains all remotely connected platforms.

Parameters: none

Responses: 'list_of_platforms' (Map of all remotely connected platforms).

Config File

The config file for AgentSynchronizeSupervisor, which contains important messages to connect other LARS, is usually called *PlatformSynchronization.cfg* and can be found in the conf/lars.

4.3.12 AgentSynchronizeConnectionHandler

Child of: AgentSynchronization.

Related to: Agents that participate in the synchronization mechanism.

Functionality: AgentSynchronizeConnectionHandler manages remote platforms' connections. It creates new messengers or removes existing messengers from a LARS platform by closing open connections. The following connection types are possible and can be specified in PlatformSynchronization.cfg:

socket

jsocket

jsecuresocket

rmi

All synchronization agents' configuration files have the structure as defined in AgentSynchronization.cfg.

Incoming Messages

You can send messages with the following services to AgentSynchronization. Message parameters are listed with the messages and must be transmitted using a Map.

open_connections (internal use only!)

Function: Receives a list of remote platforms and tries to connect to them.

Parameters: remote LARS platforms (List).

Responses: none.

close_connections (internal use only!)

Function: Receives a list of remote platforms and tries to close their connections.

Parameters: remote LARS platforms (List).

Responses: none.

Config File

AgentSynchronizeConnectionHandler does not need a config file.

4.3.13 AgentSystemInformation

Child of: AgentTemplate.

Related to: AgentCron.

Functionality: AgentSystemInformation is able to write system information in the system's log file.

Incoming Messages

You can send messages with the following services to AgentSystemInformation. Message parameters are listed with the messages and must be transmitted using a Map

log_system_information

Function: This message causes AgentSystemInformation to call Lars.logSystemInformation().

Parameters: *none.*

Responses: none.

In order to be able to write to system's log file constantly you need to do two things:

1. The following message (notification) must be included in AgentCron.cfg:

```
<MESSAGE>
    <service>notify_agent</service>
    <content>
        <period>
            <type>INTERVAL</type>
            <minute>15</minute>
            <hour>0</hour>
            <day>0</day>
        </period>
        <MESSAGE>
            <service>log_system_information</service>
                <receiver>SERVICE_PROVIDER</receiver>
                <content></content>
        </MESSAGE>
    </content>
</MESSAGE>
```

2. The log_system_information service must be registered in AgentSystemInformation's config file as show below:

```
<MESSAGE>
    <service>register_service</service>
    <content>log_system_information</content>
</MESSAGE>
```

Config File

AgentSynchronizeConnectionHandler does not need a config file.

5 Agent-like clients

5.1 Introduction

This chapter gives a short overview of these components and how to use them in an application. Agent-like clients can be considered to be counterparts of the LARS agents that are not running on the LARS platform although they behave as if they are (from the LARS point of view).

Agent-like clients are remote applications (e.g. servlets, applets) that can connect to a running LARS platform and communicate with any of the agents living on that platform. Afterwards the remote application disconnects from the LARS platform.

There is a client communication framework in the LARS package that allows agent-like clients to communicate with a running LARS platform. Remote applications can use this framework to send and retrieve messages from the platform as if they are agents.

This framework currently consists of the following components:

- The interface IToLars, which defines methods for connecting to and disconnecting from a LARS platform.

- The interface ICommunication that defines methods for communicating with a LARS platform using synchronous or asynchronous messaging. This interface also defines some communication-dependent constants.
- The interface IFromLars that defines a single method for receiving messages from the LARS platform.
- The class ClientCommunication that implements the IToLars interface and provides the infrastructure for communicating with the LARS.
- A set of communication protocols, which are configured using different connection parameters.

5.2 Design

The overall client communication framework design is shown below.

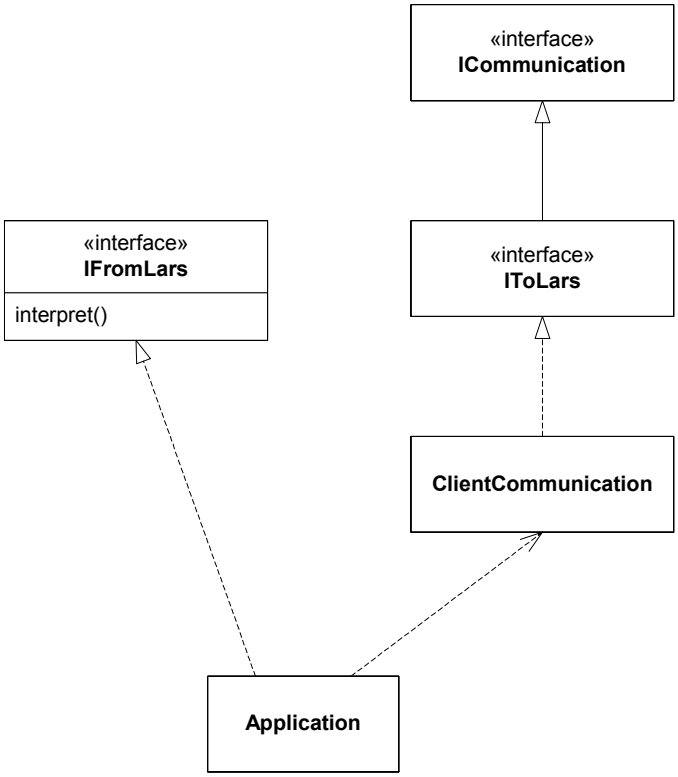

Figure 134: The design of the client communication framework

Figure 14 above shows that there are two points from which a remote client-side application can connect to the framework through an interface.

1. It has to implement the `IFromLars` interface, which is then used by the framework to deliver messages from the connected LARS platform to the application.

2. It has to instantiate a `ClientCommunication` object that will be used for sending messages from the application to the connected LARS platform.

5.3 Communication Protocols

5.3.1 Available communication Protocols

The following protocols are currently used to establish a connection with the LARS platform:

- RMI™ (Remote Method Invocation) RMI communicates by invoking remote objects methods and communicating the relevant message to them.

- Socket
  Socket communication is achieved by sending a data-stream from the remote application to the LARS platform and vice versa. The Socket protocol transmits messages as XML formatted string.

- Jsocket is used for communication by transmitting a datastream. Unlike Socket communication, Jsocket messages are sent as serialized objects instead of XML formatted strings.

- JSecureSocket
  This communication protocol works in a way similar to the Jsocket communication. In addition, the transmitted messages will be secured by encrypting them using a synchronous encoding algorithm.

- JMS™ (Java Message Service)
  This protocol communicates by interposing a JMS server, which is used as a communication middleware.

- HTTP (hypertext transfer protocol)
  The messages will be transmitted via POST/GET request from the client application to a special Java Servlet™ and vice versa.

5.3.2 Configuration

As mentioned above the interface `ICommunication` defines a couple of constants or connection parameters needed for establishing a

connection with a remote LARS platform. These parameters can be separated into two groups i.e. *general* parameters and *protocol-specific* parameters.[9]

General communication parameters

The parameters for this group are vital for setting up a connection. These parameters include information on:

- The **HOST_IP** of the machine where the remote LARS platform is running on. This parameter is mandatory.
- The logical **HOST_ID** for the contacted LARS platform. This is necessary because multiple LARS platforms could run on the same machine. This parameter is mandatory.
- The **PROTOCOL** to be used for establishing the connection. This can be one of the following: *rmi*, *socket*, *jsocket*, *jsecuresocket*, *jms* or *http*. This parameter is mandatory.
- The **PORT** where the LARS platform's communication listener is listening on. The port does mainly depend on the used protocol: in general the RMI listener is listening on port 1099, while HTTP listener uses port 80. This parameter is mandatory.
- The **TIMEOUT** to be used for the communication. This parameter is used when sending a message to the LARS platform and awaiting a response from it using synchronous communication. This parameter is optional – default setting is *3000 ms*.
- A specification whether or not to **USE_OUTBOX**. When using the outbox, the outgoing communication (from the application to the LARS) will be handled in a single thread, making the communication non-blocking. This parameter is optional – default setting is *false*.

Protocol-specific parameters

Depending on the used protocol there are additional parameters, which allow the remote application to configure the connection in a more detailed way. The following list shows the existing parameters. For more information please consult the LARS API documentation.

- Parameters for Socket
  - the **MAXIMUM_MESSAGE_LENGTH**, which specifies how many characters a receiving message can contain. This parameter is optional – default setting is *32768* characters.

[9] The separation does not apply to the JMS messenger, as this communication protocol does expect a complete different set of connection parameters.

  - the **COMPRESSION_TYPE**, which configure how the communicated messages shall be compressed using either no, zip or gzip compression. This parameter is optional – default setting is *NO_COMPRESSION*. Additional **compression parameters** are described in the API documentation of the AgentSocketListener.

- Parameters for JSocket
  - the *compression settings* as described for the Socket protocol.
- Parameters for JSecureSocket
  - the *compression settings* as described for the Socket protocol.
  - the **AUTHENTICATION_USER** and **AUTHENTICATION_PASSWORD**, which are needed for getting authenticated by the LARS platform.
    These both parameters are mandatory.

Communication parameters for the JMS protocol

When an application communicates with the LARS platform using the JMS technology, another set of connection parameters that are quite different from the above-mentioned ones[10] need to be specified:

- **JMS_PROVIDER_URL** specifies where to find the JMS server.
  This parameter is mandatory. The parameter awaits the string representation of an ip address or a url.
- **JMS_INITIAL_CONTEXT_FACTORY** specifies where to find the initial context for JMS communication. This parameter is mandatory. The parameter awaits a fully qualified classname of an implementation for the interface `InitialContextFactory`.
- **JMS_SECURITY_PRINCIPAL** specifies the identity of the principal for authenticating the remote party. Although this parameter is mandatory, it can contain an empty string.
- **JMS_SECURITY_CREDENTIALS** specifies the credentials of the principal for authenticating the remote party. Although this parameter is mandatory, it can contain an empty string.
- **JMS_RECEIVING_QUEUE_NAME** specifies the name of the JMS queue to be used for incoming messages. This parameter is mandatory.
- **JMS_SENDING_QUEUE_NAME** specifies the name of the JMS queue to be used for outgoing messages. This parameter is mandatory.

[10] Please consult http://www.java.sun.com/products/jms/index.html for detailed information

Note: When configuring the names for `JMS_RECEIVING_QUEUE_NAME` for a remote application, you have to specify the `JMS_SENDING_QUEUE_NAME` of the contacted LARS platform should also be specified. This is because the platform's outgoing messages are going to be the application's incoming messages. Likewise the LARS platform's `JMS_RECEIVING_QUEUE_NAME` has to be specified when setting up your application's `JMS_SENDING_QUEUE_NAME`.

5.4 Framework for a Client Application

A remote application that wants to communicate with the LARS platform needs to perform the following steps:

(9) The class needs to implement the `IFromLars` interface by implementing the method `interpret (Message)`.

```
import com.ls.lars.communication.*;

public class SampleApplication implements IFromLars
{
    public boolean interpret(Message message)
    {
        ...
    }
...
}
```

(10) The application needs to hold an instance of `IToLars`. As `com.ls.lars.communication.ClientCommunication` implements that interface the application just creates an object of that class.

```
private IToLars larsConnection = null;

larsConnection = new ClientCommunication( ... );
```

(11) The constructor for the ClientCommunication class takes three parameters which are listed below:

- a logical name which uniquely identifies the client to the LARS platform

- a Map containing connection parameters that configure the communication protocol to be used

- an instance of `IFromLars`, which receives the messages that have been sent from the LARS to the remote application

In this example we create a `ClientCommunication` that communicates with the remote LARS platform over RMI.

```
String clientName = "SampleApplication";
```

```
Map parameters = new HashMap();
parameters.put(ICommunication.HOST_ID,      "lars");
parameters.put(ICommunication.HOST_IP,      "141.28.228.12");
parameters.put(ICommunication.HOST_PORT,    "1099");
parameters.put(ICommunication.PROTOCOL,     "rmi");

//for ensuring uniqueness we append the current time
//in milli-seconds to the clientName
clientName += System.currentTimeMillis();

try {
    larsConnection = new ClientCommunication(clientName,
                                             parameters,
                                             this);
    larsConnection.start();
} catch (ConnectionException cex) {
    ...
}
```

(1) As the ClientCommunication class is based on thread, we have to call start() after creating an instance of ClientCommunication with the new object.

(2) From now on, the client application is free to perform any other useful stuff (for example initializing a GUI).

(3) Before it can send the first message to the LARS platform, it needs to be sure that the connection has been established. Therefore the following line is mandatory before sending the first message:

```
try {
    larsConnection.waitForConnection();
} catch (ConnectionException cex) {
    ..
}
```

(1) Messages are sent and received as described in previous chapters.

```
message = new SingleMessage(ILarsConstants.SERVICE_PING,
                    "amr@" + larsConnection.getLarsHost(),
                    "ping");
```

(2) The 'pong' response for this message would be received by the underlying ClientCommunication object, which would call the interpret () method in the new application and pass the received message to it.

6 Platform Synchronization

6.1 General

Platform synchronization allows connection of LARS platforms in order to enable cross-platform messaging. One of the simplest tasks of platform synchronization is to connect two LARS platforms, as shown in Figure 14.

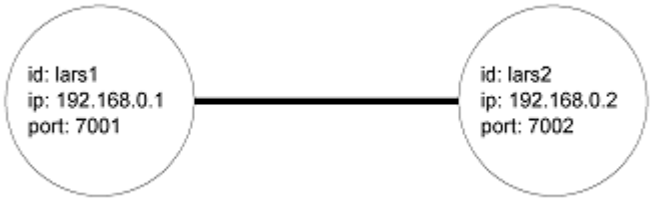

Figure 14: Two connected LARS platforms

Platform **lars1** is configured to connect to platform **lars2** at port **7002**. This enables the agents residing on platform lars1 to communicate with agents residing on platform lars2 and vice versa.

Platform synchronization can also establish connections with other platforms dynamically. This means that a running LARS platform only needs to know one other LARS platform to connect to. After connecting with that platform, it receives the connection information for other connected platforms. Therefore a network of connected platforms can be established automatically.

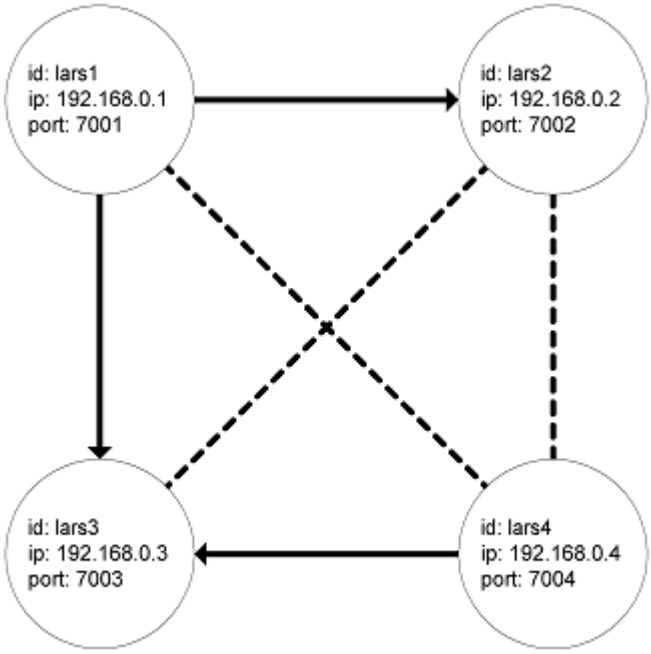

Figure 15: A network of inter-connected LARS platforms

Figure 15 shows a network of four connected LARS platforms. The solid arrows indicate that actual connections were established from one platform to another platform as stated in the configuration file. The dashed lines show the connections established dynamically by the platform synchronization algorithm. Only the neighbors listed below are included in the platform synchronization files for the different LARS platforms.

For platform:

- lars1: Connect to platform lars2 and platform lars3.
- lars4: Connect to platform lars3.

The result of platform synchronization is that all platforms know each other and are able to send messages to their neighbors without routing. Generally it should be enough to make one connection to a number of synchronized platforms and the connections to the other platforms are established automatically.

Dynamic building up of the platform synchronization depends on the platform synchronization security policy. The security policy is related to the type of connection between two platforms.

There are three different kinds of connections:

a. Public: If two platforms share a public connection, they inform each other about their connection parameters (defined as public) and all other public connected platforms they know.

b. Private: If two platforms share a private connection, they inform each other about their connection parameters (defined as private) and all other public connected platforms they know. This seems to be the same as the public security policy. The difference is that other public connected platforms do not get the connection information from private ones. Thus private connected platforms establish connections with public connected platforms dynamically and not the other way around.

c. Hidden: If two platforms share a hidden connection, they inform each other about their connection parameters (defined as hidden) and nothing else. This prevents establishing connections with other platforms dynamically.

These different kinds of security policies allow partially synchronized platform networks. Figure 16 and Figure 17 are examples for combining different platform security policies.

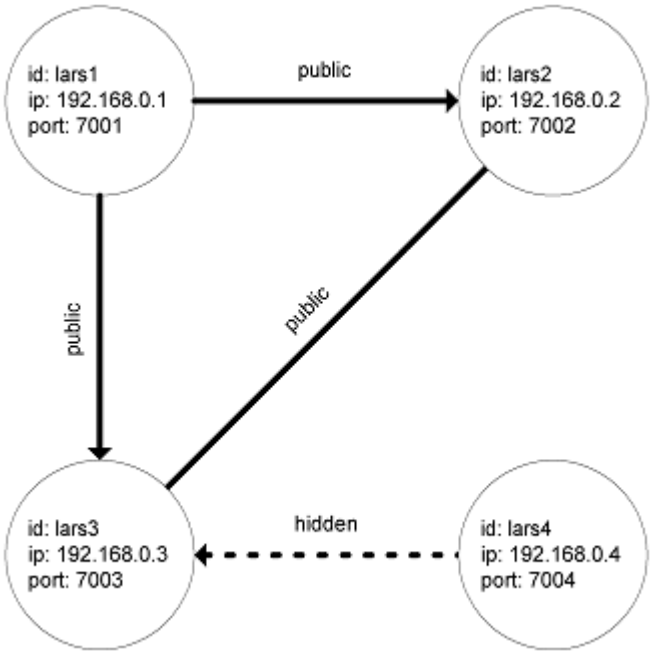

Figure 16: Four LARS platforms with public and hidden connections

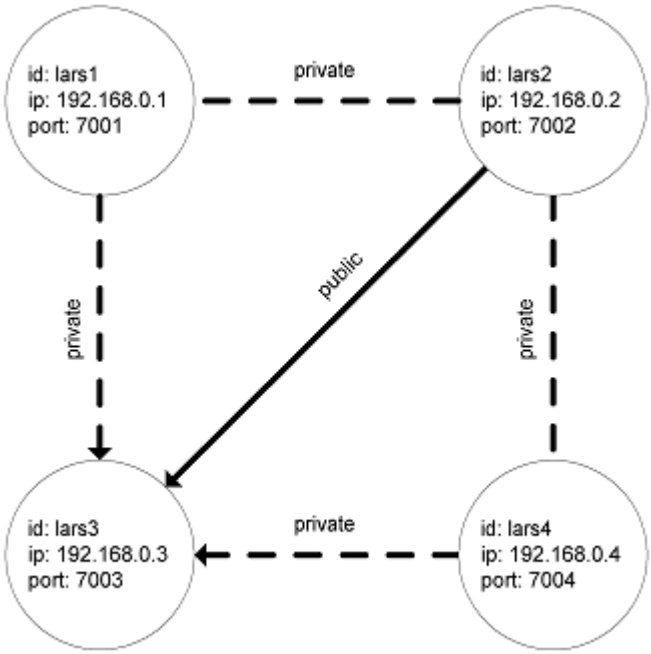

Figure 17: Four LARS platforms with public and private connections

6.2 Related Agents

There are three new agents for platform synchronization i.e. AgentSynchronizeSupervisor, AgentSynchronizeConnectionHandler, AgentPlatformSecurity and at least one listener agent and the AgentTimer.

6.2.1 AgentSynchronizeSupervisor

AgentSynchronizeSupervisor gives connection orders establishing connections to new remote platforms. It closes lost connections to remote platforms properly and checks from time to time whether the remote platforms are still connected. The messages the AgentSynchronizeSupervisor understands see 5.3.10 and 5.3.12.

6.2.2 AgentSynchronizeConnectionHandler

AgentSynchronizeConnectionHandler manages the connections to the remote platforms. It creates new messengers or removes messengers from a LARS platform. The messages the AgentSynchronizeConnectionHandler understands see 5.3.10 and 5.3.12.

6.2.3 AgentPlatformSecurity

AgentPlatformSecurity is responsible for the security policy of the LARS platform synchronization. The chosen policy (public, private, hidden) is obtained from the information sent about the connected remote platforms.

It is also important that the security policy of trusted and untrusted platforms/IP addresses allows the listener to accept connections from the particular remote platform(s). For example, make sure that the security configuration file (often called Security.cfg) contains the IP addresses of the remote platforms. For messages understood by AgentSynchronizeConnectionHandler, please see 5.3.10

6.2.4 AgentTimer

Since the platform synchronization mechanism checks connected remote platforms from time to time, the AgentTimer must run on the LARS platform.

6.2.5 AgentListener

To establish connection it is essential that the configured port of the agent listeners is the same as the port defined in the platform synchronization configuration file.

6.3 Platform Synchronization Configuration File

All platform synchronization related agents read the same configuration file (often called *PlatformSynchronization.cfg*). In the configuration file there are two services specifiable:

- set_constants: The service `set_constants` defines the general configurations of the platform synchronization.
- `platform_synchronization`: It defines the configurations of the remote platform connections.

6.3.1 General Configurations

The following default settings can be overwritten using the set_constants message:

- Name of the AgentSynchronizeSupervisor:
  - Default: ass
  - Configurable in the configuration file with an XML-tag:

```
<AgentSychronizeSupervisor>ass</AgentSychronizeSupervisor>
```

- Name of the AgentSynchronizeConnectionHandler:
  - Default: asch
  - Configurable in the configuration file by XML-tag:

```
<AgentSychronizeConnectionHandler>asch</AgentSychronizeConnectionHandler>
```

- Name of the AgentPlatformSecurity:
  - Default: aps
  - Configurable in the configuration file with an XML-tag:

```
<AgentPlatformSecurity>aps</AgentPlatformSecurity>
```

- Name of the listeners for the platform:
  - Default: asl (AgentSocketListener) Only the AgentSocketListener must run. Therefore platform synchronization is done by socket connections only.
  - Configurable in the configuration file with an XML-tag: other forms of communication with other platform(s) can also be defined by specifying other listeners in the configuration file. For each listener, the name and type of connection to be used have to be specified. Constants for the supported connection types are defined in the com.ls.lars.communication.ICommunication interface.

```
<ownPlatformListener>
  <listenerName>asl</listenerName>
  <listenerType>socket</listenerType>
</ownPlatformListener>
<ownPlatformListener>
  <listenerName>arl</listenerName>
  <listenerType>rmi</listenerType>
</ownPlatformListener>
```

- Preferred protocol order:
  - Default: socket
  - Configurable in the configuration file with an XML-tag: If connection with a remote platform can be established using

several different connection types (e.g. jsecure socket, jsocket, socket, rmi), the preferred order of connection establishment can be predefined.

- Constants for the supported connections are defined in *com.ls.lars.communication.ICommunication* interface.

```
<preferredProtocolOrder>
    <protocolType>jsecuresocket</protocolType>
    <protocolType>socket</protocolType>
    <protocolType>rmi</protocolType>
</preferredProtocolOrder>
```

Note: It is important to specify the listener supporting the protocol type defined in the preferredProtocolOrder tag and to make sure that the AgentManager starts the specified listener.

- Synchronization Intervals: There are two different time intervals which can be specified:
  - attemptConnections: If there are orders to establish or to close the connections to a remote platform(s), the retry time interval (default: 6s) can be set. This time interval is used if the platform synchronization is not completely finished.
  - checkConnections: If the platform has no new information or the platform connections are not established, the retry time interval (default: 60s) can be set. This time interval is used if the platform synchronization is stable.

The attemptConnection interval is shorter than the checkConnection interval.

```
            <synchronizationIntervals>
                <attemptConnections>6</attemptConnections>
                <checkConnections>60</checkConnections>
            </synchronizationIntervals>
```

If the default values are good enough you do not have to specify anything.

An example of `set_constants` configuration:

```
    <MESSAGE>
        <service>set_constants</service>
        <content>
            <AgentSychronizeSupervisor>
               ass
            </AgentSychronizeSupervisor>
            <AgentSychronizeConnectionHandler>
               asch
            </AgentSychronizeConnectionHandler>
            <AgentPlatformSecurity>aps</AgentPlatformSecurity>
            <ownPlatformListener>
                <listenerName>asl</listenerName>
                <listenerType>socket</listenerType>
            </ownPlatformListener>
```

```
            <ownPlatformListener>
                <listenerName>ajsl</listenerName>
                <listenerType>jsocket</listenerType>
            </ownPlatformListener>
            <preferredProtocolOrder>
                <protocolType>socket</protocolType>
                <protocolType>jsocket</protocolType>
            </preferredProtocolOrder>
            <synchronizationIntervals>
                <attemptConnections>5</attemptConnections>
                <checkConnections>60</checkConnections>
            </synchronizationIntervals>
        </content>
    </MESSAGE>
```

6.3.2 Synchronous Communication Configurations

The remote platform(s) with which a platform should establish a connection is defined here. A remote LARS platform is configured with the five parameters listed below:

Platform Id

```
                <platformId>lars2</platformId>
```

Platform IP address

```
                <ipAddress>192.168.0.2</ipAddress>
```

Port number set by the AgentListener

```
                <port>7002</port>
```

Access type of the platform synchronization

```
                <access>public</access>
```

Type of listener to be connected (e.g. AgentSocketListener, AgentRMIListener)

```
                <connectionType>socket</connectionType>
```

Parameters of the connection (e.g. AgentSocketListener). An example of the connection parameters for AgentSocketListener is shown below for more details:

```
        <connectionParameters>
            <compressionType>zip</compressionType>
            <autoCompression>off</autoCompression>
            <autoCompressionStartSize>
                4096
            </autoCompressionStartSize>
            <compressionLevel>7</compressionLevel>
            <zipEntryName>Message</zipEntryName>
        </connectionParameters>
```

An example of `synchronize_platform` configuration excluding connection parameters:

```
<CONFIG>
    <MESSAGE>
        <service>synchronize_platforms</service>
        <content>
            <platform>
                <platformId>lars2</platformId>
                <ipAddress>192.168.0.2</ipAddress>
                <port>7002</port>
                <access>public</access>
                <connectionType>socket</connectionType>
            </platform>
        </content>
    </MESSAGE>
</CONFIG>
```

Note: It is important that the particular LARS platform(s) have connection permission on the remote platform(s).

7 HowTos

7.1 LARS Config-Files

This section shall provide information about config files used in the **LARS** in form of a FAQ. The FAQ starts with common questions one should ask before starting to configure a LARS platform. It ends with questions that could come up during the configuration of specific agents.

7.1.1 Requirements

- *lars.cfg*: general config file for the LARS platform (see 7.1.4)
- *AgentManager.cfg*: config file to specify which agents AgentManager has to create (see 7.1.10)
- *LarsAdministrator.cfg*: config file to specify the LARS administrators allowed to shutdown the LARS platform (see 7.1.14)
- *AgentRMIListener.cfg*: to specify the RMI port (otherwise the default port 1099 is used) and to define how to use the message out buffer
- If some project specific configuration is needed, a project specific config file, e.g. *Project.cfg* or *<Projectname>.cfg* is created to configure things that are needed by nearly all agents (see **Error! Reference source not found.**)

- *Security.cfg*: to specify trusted/distrusted agents, LARS platforms or IP addresses used by remote agents or remote platforms. The *Security.cfg* file is used by the AgentPlatformSynchronization agent (see 7.1.26 and 7.1.27)

- Config file for each agent, that lives on the LARS platform and needs different configuration parameters than those in the config files mentioned above.

7.1.2 Location

Config files are usually located in the following directory:

"/www/<customerName>/<projectName>/conf" with the subdirectories as shown in Figure 18. Apart from the specific project config files and other project related installations that need config files, configurations can be found in the following sub directories:

- conf/lars contains only LARS config files (e.g. *lars.cfg*, *LarsAdministrator.cfg*)

- *conf/agents/<agent-purpose>* contains agent specific config-files

- *conf/common* contains config-files that are common to all agents of the project

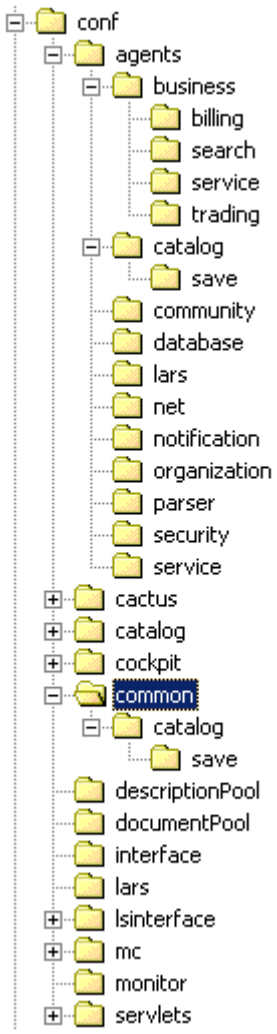

Figure 18: configuration directory structure

7.1.3 Appearance

Config files are written in XML. Each file starts with a "<CONFIG>" tag and ends with a closing "</CONFIG>" tag. The tags in between depend on the config file.

```
<CONFIG>
    <!-- depend on the agent you want to configure -->
</CONFIG>
```

Note 1: Tag names are case sensitive.

Note 2: Config files are much more easy to read if the XML structure is made visible by indenting inner tags with tabulators!

7.1.4 Specifying Configuration Files

The config file used to configure the LARS platform is given as a command line argument when invoking the LARS.

```
Example:
 java com.ls.lars.server.Lars –config /www/living-
systems/auction/conf/lars/lars.cfg
```

Figure 19: An Example for specifying the LARS platform's configuration file

If no config file is specified, the file *lars.cfg* in the current working directory is used as default.

7.1.5 Platform Configuration Files

```
<CONFIG>

    <LOGGING>
        <globalLogPath>
            /www/customerName/projectName/log
        </globalLogPath>
        <logFile>lars.log</logFile>
        <logLevel>INFO</logLevel>
    </LOGGING>

    <PLATFORM>
        <platformId>lars</platformId>
        <ipAddress>195.226.125.242</ipAddress>
        <globalConfigPath>
            /www/customerName/projectName/conf/
        </globalConfigPath>
        <agentManagerConfigFile>
            agents/lars/AgentManager.cfg
        </agentManagerConfigFile>
        <agentManagerConfigFile>
            agents/lars/Administrator.cfg
        </agentManagerConfigFile>
        <agentManagerLogFile>
            $a.log
        </agentManagerLogFile>
        <agentManagerLogLevel>INFO</agentManagerLogLevel>
        <locale>de_DE</locale>
        <encoding>UnicodeBig</encoding>
    </PLATFORM>

</CONFIG>
```

Figure 20: An Example of lars.cfg

The platform config file is usually named lars.cfg and contains the following two tags enclosed within its *CONFIG* tag:

In the *LOGGING* block i.e. within the LOGGING tag, configuration issues regarding logfiles are specified as follows:

- *globalLogPath* specifies an absolute path used as the starting path for relative logfile access (compare question 7.1.8). There is no need to specify this tag, but if it is not given, all logfiles have to be addressed with the absolute path! The path specified in this tag does not need to end with a slash (/), but it may.

- *logFile* determines the file to which the LARS platform writes log messages. It may be given as a relative path, if the *globalLogPath* is specified with a valid directory.

- *logLevel* sets the log level for the platform[11]. The recommended log level for the platform is INFO.

In the *PLATFORM* block other platform specific parameters are configured:

- *platformId* sets the name of the LARS platform, which is used for communication and logging purposes[12].

- *ipAddress* can be used to set the IP address of the LARS platform. This tag is only needed if the address cannot be determined automatically, e.g. if the machine the LARS is running on has multiple network addresses

- *globalConfigPath* specifies an absolute path used as the starting path for relative config file access (compare question 7.1.8). There is no need to specify this tag, but if it is not given, all config files have to be addressed with the absolute path!
  Since LARS v3.0 you are also able to load config files from a web server. Therefore you can set the globalConfigPath to a URL (see example in Figure 21).

```
<globalConfigPath>
    <protocol>http</protocol>
    <ipAddress>www.server.com</ipAddress>
    <path>customerName/projectName/conf</path>
</globalConfigPath>
```

Figure 21: An example of how to set the global config path to a URL

- *agentManagerConfigFile* specifies where the config file of the agent manager will be found. The addressing may be absolute or – if globalConfigPath is set – relative. Since LARS v3.0 it is possible to specify multiple config files by having multiple AgentManager config files, e.g.:

```
            <agentManagerConfigFile>
                    agents/lars/AgentManager.cfg
```

[11] Note, that since LARS v2.1 it is possible to use the name of the log level instead of a number

[12] This name really needs to be the same as configured in the servlet – otherwise communication between servlet and agents is impossible.

```
        </agentManagerConfigFile>
        <agentManagerConfigFile>
                agents/lars/CatalogAgentManager.cfg
        </agentManagerConfigFile>
        <agentManagerConfigFile>
                agents/lars/ServiceAgentManager.cfg
        </agentManagerConfigFile>
        <agentManagerConfigFile>
                agents/lars/BusinessAgentManager.cfg
        </agentManagerConfigFile>
```

- *agentManagerLogFile* specifies where the log file of the agent manager will be located. The addressing may be absolute or relative to the configured *globalLogPath*.

- *agentManagerLogLevel* specifies the level up to which the agent manager should write its logging details into its log file.

- *locale* is used to set the default Locale of the Java Virtual Machine and consists of a valid ISO Language Code[13], a delimiting underscore ('_') and a valid ISO Country Code[14]. This tag is not mandatory, if it is not specified; the locale settings of the environment where LARS started are taken. But note that this it is the only way to influence the locale settings, because changing the locale at LARS runtime is disallowed!

- *encoding* specifies how files and streams (for example written to a socket or read from a socket) are encoded (the java property "sun.io.unicode.encoding" is set to the configured value). This tag is not mandatory.

Note: The order of the tags in each block of the platform config file is not evaluated – except of the tag order in the AgentManger config file!

7.1.6 Agent Configuration Files

Agent config files consist of a *CONFIG* block containing *MESSAGE* tags. These message tags are read and interpreted by the agent (see 7.1.22). The config file may contain a message that the agent understands and that does not expect an answer (see 7.1.23).

---

[13] A valid ISO Language Code is a lower-case two-letter code as defined by ISO-639. You can find a full list of these codes at a number of sites, such as:

http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt

[14]A valid ISO Country Code is an upper-case two-letter code as defined by ISO-3166. You can find a full list of these codes at a number of sites, such as:

http://www.chemie.fu-berlin.de/adressen/isocodes.html

Note: The order of several messages given is derived from their order in the config file.

The *MESSAGE* block contains the following tags:

- *service* specifies the service of the message (that's the string, which is checked in the interpret method to determine the action to perform).
- *receiver* remains unset and is not interpreted if given.
- *sender* remains unset and is not interpreted if given.
  The sender of a config file message is automatically set to *Message.ADDRESS_CONFIG_FILE*.
- *content* depends on the message service and contains the body of the message – this may be only one value or another complex XML structure, which then reaches the agent as a map.

7.1.7 Internationalization Parameters Configuration

Locale and stream encoding see 7.1.5.

7.1.8 Addressing of Configuration and Logfiles

It is possible to specify relative paths to log- and config files in the lars.cfg (see 7.1.5).

Even if the relative paths are specified in lars.cfg, it is possible to configure any path to a different absolute directory just by specifying an absolute path to the file instead of a relative one. (A path being absolute or relative, is determined via java.io.File.isAbsolute (). Note, that for a Windows configuration, the path "\log" is not absolute, because there is no drive specification, the following example shows an absolute path "d:\log" or "\\networkdrive\log".)

If the given logfile or config file path is recognized as a relative path, the configured globalLogPath respectively globalConfigPath is appended to the name used to access the file.

7.1.9 Variable Substitution

There are currently three variables that can be used to replace logfile and config file names:

- $A is replaced with the agent's name
- $P is replaced with the platform's name
- $C is replaced with the agent's class name without the package (since LARS v2.3)

These variables may be used, when specifying an agent's logfile or config file, but not for specifying the log or config file for the LARS platform itself.

This makes it quite flexible to use config files: If the same agent is running on more than one LARS platform (e.g. for load scalability reasons) or if some agents need the same configuration parameters. All these agents may use the same config file (and yet have their own logfiles).

Examples:

- The String from the config file `'/www/customerName/projectName/log/$A.log' could become '/www/customerName/projectName/log/AgentInfo@192.168.100.114-lars.log'.
- The String from the config file `'/www/customerName/projectName/log/AgentInfo_$P.log' could become '/www/customerName/projectName/log/AgentInfo_192.168.100.114-lars.log'.
- The String from the config file `'/www/customerName/projectName/log/$C.log' could become '/www/customerName/projectName/log/AgentInfo.log'.

7.1.10 AgentManager Configuration Files

Since LARS v3.0 it is recommended to specify at least two config files for AgentManager in the *agentManagerConfigFile* tag of the LARS platform's config file: *AgentManager.cfg* (see 7.1.11) and LarsAdministrator.cfg (see 7.1.14).

Further on the following files may be used to organize *start_agent* messages:

- AgentManager.cfg
  (containing the LARS agents startup messages)
- BusinessAgentManager.cfg
  (containing the business agents like auction agents startup messages)

- ServiceAgentManager.cfg
  (containing the service agents like AgentSendMail, AgentStaticMaker ...startup messages)

- CommunityAgentManager.cfg
  (containing the community agents like AgentMessageBoard ... startup messages)

- NetAgentManager.cfg
  (containing the net agents like AgentFTP startup messages)

Of course additional AgentManager config files are possible.

All of these config files are located in the ".../conf/agents/lars" directory and are listed in the "*lars.cfg*" file which is located in the ".../conf/lars" directory.

7.1.11 AgentManager.cfg

AgentManager.cfg starts with a "start_agent" message that asks AgentManager to start AgentMessageRouter.

Note: *AgentMessageRouter needs to be the first agent started by AgentManager!*

Note: In former versions of LARS, a "set_log" message had to be the first message of the AgentManager.cfg. With the new LARS v3.0, this message has become obsolete, as the appropriate settings are done in the platform's config file (see 7.1.5).

This is followed by the "start_agent" message, which tells AgentManager to start platform services like *AgentTimer, AgentSystemInformation* and *Listeners*.

```
<CONFIG>

    <MESSAGE>
        <!--this needs to be the first message -->
        <service>start_agent</service>
        <content>
            <class>
                com.ls.lars.server.AgentMessageRouter
            </class>
            <name>amr</name>
            <cfgFile />
            <logFile>$a.log</logFile>
            <logLevel>INFO</logLevel>
        </content>
    </MESSAGE>

    <MESSAGE>
```

```
        <service>start_agent</service>
        <content>
            <class>com.ls.lars.server.AgentRMIListener</class>
            <name>arl</name>
            <cfgFile>agents/lars/$c.cfg</cfgFile>
            <logFile>$a.log</logFile>
            <logLevel>INFO</logLevel>
        </content>
    </MESSAGE>

    <MESSAGE>
        <service>start_agent</service>
        <content>
            <class>com.ls.lars.server.AgentTimer</class>
            <name>at</name>
            <cfgFile />
            <logFile>$a.log</logFile>
            <logLevel>INFO</logLevel>
        </content>
    </MESSAGE>

    <MESSAGE>
        <service>start_agent</service>
        <content>
            <class>
                com.ls.lars.server.AgentSystemInformation
            </class>
            <name>asi</name>
            <cfgFile>
                agents/lars/AgentSystemInformation.cfg
            </cfgFile>
            <logFile>$a.log</logFile>
            <logLevel>WARNING</logLevel>
        </content>
    </MESSAGE>

    <!--... further start_agent messages ....  -->

</CONFIG>
```

Figure 22: An Example of AgentManager.cfg (LARS v2.3)

The content tags of such a "start_agent" message consist of the following elements:

- *class* tag: the name of the class to be started (complete class name with package, without. class)
- *name* tag: how the agent will be named (the platform extends this name with an "@" character and the platform's name)
- *logFile* tag: which log file has to be used (since LARS v2.3, see 7.1.13)
- *logLevel* tag: at what log level the new agent should be started (since LARS v2.3, see 7.1.13)

- *cfgFile* tag(s): which config files are to be read and interpreted (see 7.1.16).
  Since LARS v2.3: If no config file is needed, an empty cfgFile (<cfgFile /> or <cfgFile></cfgFile>) tag should be written to avoid error messages in the log files.

- Since LARS v3.0: if you are using a *globalConfigPath* within a URL you can explicitly set where to get each config file by just overwriting the TAGs (protocol, ipAddress, path), see the example in Figure 23.

```
<cfgFile>
    <protocol>ftp</protocol>
    <ipAddress>www.server2.com</ipAddress>
    <path>lars/conf/config.jar</path>
    <name>AgentLogin.cfg</name>
</cfgFile>
```

Figure 23: An Example of a cfgFile tag with a protocol

It is also possible to load a config file from a jar file by adding the jar/zip file to the path. Otherwise the path is set without any file, because the file name from *name* tag will be used.

7.1.12 Startup-Order of Agents

Yes, the startup-order is very important (See 7.1.10).

With the new LARS v3.0, it is also possible to define startup constraints (see 7.1.17).

7.1.13 Log file and Log Level Configuration

Since we started using LARS v2.3, the log file in the "start_agent" message in AgentManager's config file(s) should be configured. Only then can this log file be used for logging errors that may occur when the agent is being started (e.g. when parsing the XML syntax of the agents' config files). This makes finding configuration errors (like a forgotten slash on the closing tag) much easier.

There are lots of agents that start logging before their log file is set up. If the log file in the start_agent message has not been configured, this

results in writing log messages to standard output, the output in this instance would be found in a log file like *"/www/log/lars.log"*[15].

7.1.14 LarsAdministrator.cfg

One or more LARS Administrators can be specified in order to shutdown the LARS platform. An admin name and password should be defined for each administrator.

This file is usually located in the …/conf/lars/ directory.

Note: It is important to have a separate config file for the definition of the LARS administrators. This is because the LARS Administrator(s) is obtained by the larsctl script before shutting down the platform.

```
<CONFIG>

    <MESSAGE>
        <service>set_lars_administrator</service>
        <content>
            <administrator>mattin</administrator>
            <password>questVK6G</password>
        </content>
    </MESSAGE>

    <MESSAGE>
        <service>set_lars_administrator</service>
        <content>
            <administrator>michael</administrator>
            <password>7gj8op</password>
        </content>
    </MESSAGE>

</CONFIG>
```

Figure 24: An Example of LarsAdministrator.cfg

Note: The permissions for this file should not allow others to read this file!

7.1.15 Multiple Configuration File

Is it possible to use more than one config file for one agent.

[15] The directory *"/www/log/"* is used for logging everything, that is written to standard out or standard error. Ideally LARS-logfiles residing there should be empty.

This occurs when the information needed by the Agent is obtained from different config files set up for project specific functionalities.

It is generally a good idea to set up project specific things that are interesting for most agents in a separate config file instead of copying this information into every single config file[16].

A good example of where it makes sense to have multiple config files are some config files containing project specific configuration messages or a config file containing the parser to be used.

Prior to LARS v2.3 config files for the different log levels had to be defined e.g. a config file named LoglevelWarning.cfg containing only one message as shown in Figure 25.

```
<CONFIG>

    <MESSAGE>
        <service>set_log</service>
        <content>
            <logFile>$A.log</logFile>
            <logLevel>WARNING</logLevel>
        </content>
    </MESSAGE>

</CONFIG>
```

Figure 25: An Example of LoglevelWarning.cfg

By using variable substitution (see question 7.1.9) and relative file addressing (see question 7.1.6) in the *logFile* tag, this config file can be used by each agent that needs to run with log level *WARNING* (see also question 7.1.17). It is no longer necessary to create separate config files for different log levels because the log level and the log file's name are now specified in the start_agent message (see question 7.1.10).

7.1.16 More than One Configuration File

To use more than one config file for one agent, multiple *cfgFile* tags can be defined in the content of the *start_agent* message in the project specific AgentManager config file, e.g. BusinesAgentManager.cfg:

```
<MESSAGE>
  <service>start_agent</service>
    <content>
      <class>
```

[16] Often problems arise when deploying your project from the stage server to the production server, because someone forgot to update a config file. This difficulty increases, if the same information is maintained in different config files.

```
 com.ls.business.trading.auction.english.server.AgentAuctionBid
      </class>
      <name>AgentAuctionBid</name>
      <logFile>$a.log</logFile>
      <logLevel>TRACE5</logLevel>
      <cfgFile>agents/service/ErrorHandler.cfg</cfgFile>
      <cfgFile>agents/parser/ParserDefault.cfg</cfgFile>
      <cfgFile>
        agents/business/trading/AgentAuctionBid.cfg
      </cfgFile>
    </content>
</MESSAGE>
```

Figure 26: Specifying more than one config file for an agent

The files are read and interpreted in the order they are defined.

7.1.17 Startup Dependencies

Startup dependencies are an elegant way of specifying requirements that must be satisfied before the agent's starts. Reasons could be:

- A particular agent must run before a message can be sent to it.
- The sequence of agents is of importance.
- Etc.

Note: The agent start sequence in the AgentManager.cfg is also important. Therefore, it is usually sufficient enough to put agents on which other agents depend at the beginning of the AgentManager.cfg file.

7.1.18 Defining Agent Startup Dependencies

It often happens that an agent is dependent on another agent’s service (see 7.1.17). An example of this is when an agent needs to perform some tasks regularly and therefore depends on the service of AgentCron. With the new LARS 3.0, to ensure AgentCron is not asked for regular notification before it has started up completely, startup constraints can be defined within any config file message or list of config file messages.

This is done with help of a *Startup dependencies* can be defined by using *a startup_constraint* message that contains one or more <awaitedAgent> tags in the content. (The names of the awaited agents are automatically extended with an “@” and the platform ID’s.) If all of the so awaited agents are available on the platform, AgentMessageRouter notifies the agent and the agent then interprets

the messages from the content of the startup_constraint message in the given order. The message is interpreted is interpretation as any other message read from a config file, see question 7.1.22.

```
<MESSAGE>
    <service>startup_constraint</service>
    <content>
        <awaitedAgent>AgentA</awaitedAgent>
        <awaitedAgent>AgentB</awaitedAgent>
        <MESSAGE>
            <service>do_this_or_that_first</service>
            <content>
                [message specific content goes here]
            </content>
        </MESSAGE>
        <MESSAGE>
            <service>do_another_task_afterwards</service>
            <content>
                [message specific content goes here]
            </content>
        </MESSAGE>
    </content>
</MESSAGE>
```

Figure 27: An example of how to specify startup constraints

7.1.19 Sending a Message to an Agent

The agent being waited for is defined in the startup_constraint (see 7.1.17 and 7.1.18). You can send a message to an agent you waited for (<awaitedAgent>) by specifying a message: *send_as_configured*

For example to register an agent with AgentCron for regular check_inbox_size (every 5 minutes). Then the following startup_constraint could be defined in the agent's config file. (Figure 28):

```
<CONFIG>
  <MESSAGE>
    <service>startup_constraint</service>
    <content>
      <awaitedAgent>aCron</awaitedAgent>
      <MESSAGE>
        <service>send_as_configured</service>
        <content>
          <MESSAGE>
            <service>notify_agent</service>
            <receiver>aCron</receiver>
            <content>
              <period>
                <type>INTERVAL</type>
                <minute>5</minute>
                <hour>0</hour>
                <day>0</day>
```

```
              </period>
              <MESSAGE>
                <service>check_inbox_size</service>
                <receiver>$a</receiver>
                <content/>
              </MESSAGE>
            </content>
          </MESSAGE>
        </content>
      </MESSAGE>
    </content>
  </MESSAGE>
</CONFIG>
```

Figure 28: startup constraints with send_as_configured

7.1.20 Types of Messages to be Sent

If no message type is specified, the SingleMessage message type is used. In most cases, this is sufficient for the configuration files. If it is necessary to send another type, e.g. a SingleServiceMessage, it has to be defined with the *type* xml-tag:

```
    <type>service_single</type>
```

For a detailed description see section 3.2.

7.1.21 Prioritizing Messages

There is no general rule in deciding which message should be the first one in an agent's config file.

7.1.22 Reading Messages

When a message is read from a config file, the agent's interpret method is invoked and the agent processes the config file message like any other agent communication message.

7.1.23 Restrictions

Because the sender of a message that was read from a config file is always *Message.ADDRESS_CONFIG_FILE*, a reply to such a message will fail. This failure will be logged to the agent's log file.

7.1.24 Overcoming Restrictions

To use such a message in a config file, the agent's behavior can be changed to interpret the message in the following way:

If the sender equals *Message.ADDRESS_CONFIG_FILE*, the agent should not send a reply, but write INFO message to the log file containing the information that otherwise would have been sent as a reply.

7.1.25 Messaging

The messages an agent understands are explained in the agent's javadoc pages in detail in the classes' header. It is also noted if a message is expected to be read from a config file.

7.1.26 Security.cfg

The *Security.cfg* file is used in the *AgentPlatformSecurity* and defines the trusted/untrusted agents , LARS platforms and IP addresses. This is relevant for all remote messengers and actions initiated from outside the local LARS platform. IP addresses may contain an asterisk ("*") at the end (as in the Apache Web Server configuration), which denotes a set of IP addresses including every possibility behind the asterisk, e.g. "192.168.100*" or "192.168.100. *" is the set of all IP addresses starting with "192.168.100."

Untrusted agents can also be specified (by their name). The untrusted agents are not allowed to request a connection. Similar to the trusted IP addresses the names may include asterisks, which is useful in denying e.g. client requests, because client names are often extended by adding the current time (in milliseconds since 1970-01-01) in order to create unique names.

```
<CONFIG>

    <MESSAGE>
        <!--configures the list of platforms, that are -->
        <!--allowed to contact the platform            -->
        <service>set_trusted_platform</service>
        <content>
            <platform>141.168.1.1-lars2</platform>
            <!--  or more -->
        </content>
    </MESSAGE>

    <MESSAGE>
        <!--configures the list of platforms, that  -->
```

```
        <!--are not allowed to contact the platform -->
        <service>set_untrusted_platform</service>
        <content>
            <platform>141.168.1.1-lars4</platform>
            <!--  or more -->
        </content>
    </MESSAGE>

    <MESSAGE>
        <!--configures the list of agents, that are -->
        <!--allowed to contact the platform         -->
        <service>set_trusted_agents</service>
        <content>
            <agent>Servlet*</agent>
            <!--  or more -->
        </content>
    </MESSAGE>

    <!--configures the list of agents, that are not -->
    <!--allowed to contact the platform             -->

    <MESSAGE>
        <!--configures the list of IP addresses, that are -->
        <!--allowed to contact the platform               -->
        <service>set_trusted_ip</service>
        <content>
            <ipAddress>192.168.100.*</ipAddress>
            <ipAddress>195.172.8.81</ipAddress>
        </content>
    </MESSAGE>

    <!--configures the list of IP addresses, that are -->
    <!--not allowed to contact the platform           -->

</CONFIG>
```

Figure 29: Example of a Security.cfg

7.1.27 Security.cfg

The Security.cfg file is used in the AgentPlatformSecurity and is **mandatory** if connections are to be established. The default is to **trust no one**.

7.1.28 Configure Listeners

There are different protocols that can be used (rmi, socket, jsocket, jsecuresocket and jms) to connect to a LARS platform. In the config files of the specific AgentXXXListeners, the connection specific parameters are specified for the XXXListeners. Only security issues are configured in

AgentPlatformSecurity's config file (see 7.1.26 and 7.1.27) i.e. who is allowed or not allowed. An example of the configuration for the AgentRMIListener is shown below:

```
<CONFIG>

    <MESSAGE>
        <service>set_port</service>
            <content>8006</content>
    </MESSAGE>

</CONFIG>
```

Figure 30: An Example of the AgentRMIListener.cfg file

7.1.29 Configure Platform Synchronization

The following example illustrates how to set up four LARS platforms in such a way that there will be a permanent[17] connection between the four platforms.

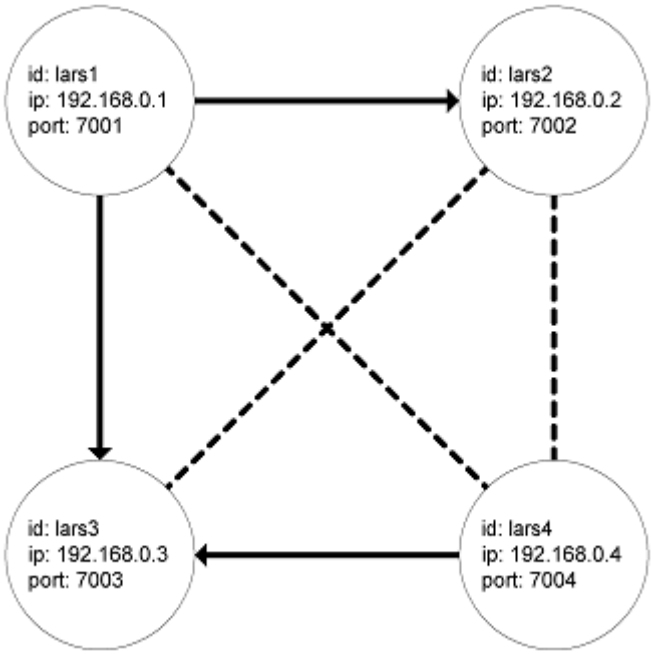

Figure 31 shows the synchronization of four platforms using public connections. The solid lines show what has to be configured in the configuration files and the dashed lines indicate the connections established automatically by the platform synchronization algorithm.

[17] In fact the connection is not permanent at all – but if the connection gets broken, the platform synchronization mechanism will re-establish it automatically

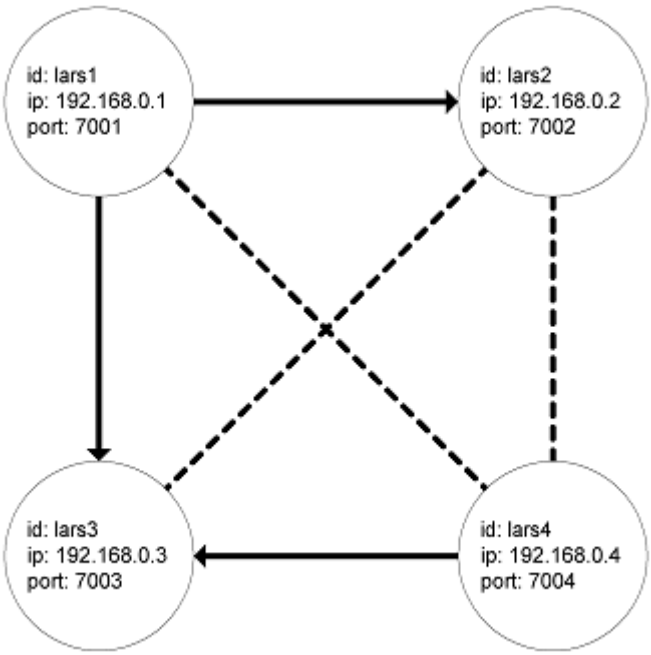

Figure 31: An Example of Four Platforms With public connections

The following sections show the configuration files for this setup.

Configuration of the platform lars1

`PlatformSynchronization.cfg` read by the AgentSynchronization agents:

```
<CONFIG>
    <MESSAGE>
        <service>synchronize_platforms</service>
        <content>
            <platform>
                <platformId>lars2</platformId>
                <ipAddress>192.168.0.2</ipAddress>
                <port>7002</port>
                <access>public</access>
                <connectionType>socket</connectionType>
            </platform>
            <platform>
                <platformId>lars3</platformId>
                <ipAddress>192.168.0.3</ipAddress>
                <port>7003</port>
                <access>public</access>
                <connectionType>socket</connectionType>
            </platform>
        </content>
    </MESSAGE>
</CONFIG>
```

Security.cfg read by AgentPlatformSecurity:

```
<CONFIG>
    <MESSAGE>
      <service>set_trusted_ip</service>
      <content>
         <ipAddress>192.168.0.1</ipAddress>
         <ipAddress>192.168.0.2</ipAddress>
         <ipAddress>192.168.0.3</ipAddress>
         <ipAddress>192.168.0.4</ipAddress>
      </content>
    </MESSAGE>
</CONFIG>
```

Configuration of the platform lars2

PlatformSynchronization.cfg not needed.

Security.cfg read by AgentPlatformSecurity:

```
<CONFIG>
    <MESSAGE>
      <service>set_trusted_ip</service>
      <content>
         <ipAddress>192.168.0.1</ipAddress>
         <ipAddress>192.168.0.2</ipAddress>
         <ipAddress>192.168.0.3</ipAddress>
         <ipAddress>192.168.0.4</ipAddress>
      </content>
    </MESSAGE>
</CONFIG>
```

Configuration of the platform lars3

PlatformSynchronization.cfg not needed.

Security.cfg read by AgentPlatformSecurity:

```
<CONFIG>
    <MESSAGE>
      <service>set_trusted_ip</service>
      <content>
         <ipAddress>192.168.0.1</ipAddress>
         <ipAddress>192.168.0.2</ipAddress>
         <ipAddress>192.168.0.3</ipAddress>
         <ipAddress>192.168.0.4</ipAddress>
      </content>
    </MESSAGE>
</CONFIG>
```

Configuration of the platform lars4

PlatformSynchronization.cfg read by AgentSynchronization agents:

```
<CONFIG>
    <MESSAGE>
        <service>synchronize_platforms</service>
        <content>
            <platform>
                <platformId>lars3</platformId>
                <ipAddress>192.168.0.3</ipAddress>
                <port>7003</port>
                <access>public</access>
                <connectionType>socket</connectionType>
            </platform>
        </content>
    </MESSAGE>
</CONFIG>
```

Security.cfg read by AgentPlatformSecurity:

```
<CONFIG>
    <MESSAGE>
      <service>set_trusted_ip</service>
      <content>
         <ipAddress>192.168.0.1</ipAddress>
         <ipAddress>192.168.0.2</ipAddress>
         <ipAddress>192.168.0.3</ipAddress>
         <ipAddress>192.168.0.4</ipAddress>
      </content>
    </MESSAGE>
</CONFIG>
```

7.1.30 Agent Pooling (agent load balancing)

Firstly, it is required that the agent you want to pool must implement the `IPoolable` interface. The Developer should make sure that this agent is safe to be run threaded and takes care of the shared resources.

For example, consider a poolable agent with the start_agent configuration within `AgentManager` set to request for several threads when starting the Agent by defining the xml-tag loadbalancing as shown below:

```
    <loadBalancing>
            <minPoolSize>1</minPoolSize>
            <maxPoolSize>5</maxPoolSize>
    </loadBalancing>
```

7.2 Employing JMS for connecting agents

This section provides an outline of how to use Java Message Services (JMS) as the communication protocol between agent-like clients and agents or agents running on different platforms.

7.2.1 Selecting a JMS server

Since JMS is an API, an implementation of the API should be selected in order to use JMS.

EJB 2.0 compliant servers provide a JMS implementation. Even some servers that support only an earlier version of the EJB standard provide JMS implementations. Examples are BEA Weblogic Server (commercial) or JBoss (open source).

Besides EJB servers, there are also standalone implementations of JMS, like MQSeries (IBM).

For testing purposes, we recommend to use the JBoss EJB server (see http://www.jboss.org).

With respect to production purposes, this section does not give guidelines for evaluation and selection. Issues like license costs, reliability and available support have to be taken into account.

7.2.2 Configuring a JMS server

The JMS implementation of your choice must be configured to provide two named JMS queues for every pair of platforms that shall be connected via JMS:

- a queue for sending messages
- a queue for receiving messages

Please consult the JMS server manuals for vendor-specific instructions on how to do this. In case you are using JBoss, its default configuration is already sufficient.

Whatever implementation of JMS is used, please make sure that the chosen server does not use the standard RMI port 1099 for its own purposes. In case you encounter any strange behavior after starting the JMS implementation, please check for this error first.

7.2.3 Configuring LARS to enable the use of JMS

Please adapt the configuration settings for AgentJMSListener (see the corresponding section in the *Agent Handbook* for details). The default configuration settings apply to the default installation of JBoss; therefore you only need to set the name or IP address of the computer that hosts the JMS implementation. AgentJMSListener should be configured on each platform.

AgentManager should also be configured so that it starts AgentJMSListener on startup. In case you are not yet familiar with the purpose or the configuration of the AgentManager, the chapter *HowTos* in the *Agent Handbook* provides several sections that deal with all aspects you need to be familiar with.

As a final step, the preferred communication protocol between the platforms has to be set to JMS. For the details, please refer to the chapter on Platform *Synchronization* in the *Agent Handbook*.

8 LARS Cockpit

8.1 Introduction

Cockpit is the successor of the LARS Manager. The big difference between Cockpit and LarsManager is that the Cockpit builds a main Interface for all user interface implementations. The cockpit package recognizes the need to split the functionality of the LARS into two i.e. 'View' and 'Control'.

This means that all functionality like connecting, disconnecting, sending messages, message boxes, starting command classes, … is implemented in the cockpit package and is independent of the user interface implementation.

The only important connection between the cockpit and the user interface implementation is an interface ('ICommand') that implements the main method to do anything. This method is called 'executeCommand'. But more details about this are given later on in this document.

The cockpit will obviously have more functionality than the LarsManager, but in the first release there will only be a shell implementation of the user interface. Therefore most of the examples will be for the shell implementation.

8.2 Installing the Cockpit

8.2.1 Packages needed

There are a lot of packages needed. The first one is the cockpit package. Probably all the User interface implementations will also be in this package.

The other packages are:

- ls-packages: base, lars, util, service, security
- external-packages: logi.crypto.jar, xml.jar

The names of the internal (living-systems) jar-Files will change, but this does not matter at the moment.

8.2.2 Writing a start script for Windows

Example:

```
@ECHO OFF

SET javaSun=C:\JavaSoft\jdk13\lib
SET javaLib=C:\lars3\lib
SET javaLS=C:\lars3\build\

SET CP=.;..
SET CP=%CP%;%javaSun%
SET CP=%CP%;%javaLS%

SET CP=%CP%;%javaLib%\external\logi.crypto.jar
SET CP=%CP%;%javaLib%\external\xml.jar

SET CP=%CP%;%javaLib%\internal\lsbase_3_1.jar
SET CP=%CP%;%javaLib%\internal\lslars_3_1.jar
SET CP=%CP%;%javaLib%\internal\lsutil_3_1.jar
SET CP=%CP%;%javaLib%\internal\lsservice_3_1.jar
SET CP=%CP%;%javaLib%\internal\lssecurity_3_1.jar
SET CP=%CP%;%javaLib%\internal\lscockpit_3_1.jar

java -cp %CP% com.ls.cockpit.Cockpit -cfg cockpit.cfg
```

The start script for unix or linux is as shown in the example above. The names of the packages can change. Using the same versions of the cockpit and the LARS platform is recommended. As long as nothing

changes in the Messengers, or ClientCommunication, the cockpit should run with each LARS version since the new package structure is available.

The last line specifies to start the cockpit. There are two possible parameters to specify, the config file and the user interface type. There is no need to specify the user interface type, if you configured it in the configuration file. Otherwise the default will be used. Default is 'shell'.

The order in which the User Interface is used is as follows: firstly the type given by command line is used, secondly the type obtained from the configuration file is used and the thirdly the default is used. To specify the user interface type just type '-type [uiType]', where 'uiType' is the type like 'shell', 'applet', 'servlet',

The same order given above is valid for the configuration file. To specify the configuration file just type '-cfg c:\cockpit\cockpit.cfg' or something similar. Using the full path for the configuration file is recommended.

8.3 Configuring the Cockpit

8.3.1 Writing a cockpit configuration file

Example of a cockpit configuration file:

```
<CONFIG>
    <LOGGING>
        <logFile>c:/cockpit/log/cockpit.log</logFile>
        <logLevel>TRACE5</logLevel>
    </LOGGING>

    <CONNECTION>
        <name>default</name>
        <platformId>lars3</platformId>
        <platformIp>192.168.0.2</platformIp>
        <platformPort>1099</platformPort>
        <connectionType>RMI</connectionType>
        <keyFile></keyFile>
        <protocol></protocol>
        <provider></provider>
        <user></user>
        <password></password>
        <larsDirectory></larsDirectory>
    </CONNECTION>

    <CONNECTION>
        <name>360T</name>
        <platformId>lars3</platformId>
        <platformIp>192.168.0.2</platformIp>
        <platformPort>8006</platformPort>
```

```
    <connectionType>RMI</connectionType>
</CONNECTION>

<COMPRESSION>
    <compressionType>no_compression</compressionType>
    <autoCompression>off</autoCompression>
    <autoCompressionStartSize>
        16384
    </autoCompressionStartSize>
    <compressionLevel>9</compressionLevel>
    <zipEntryName>Message</zipEntryName>
</COMPRESSION>

<MONITORING>
    <agentList>c:/cockpit/agentListStd.lst</agentList>
    <agentList>c:/cockpit/agentListExt.lst</agentList>
</MONITORING>

<DIRECTORIES>
    <historyDir>c:/cockpit/history/</historyDir>
    <monitorDir>c:/cockpit/monitor</monitorDir>
    <commandListDir>c:/cockpit/cmdList</commandListDir>
</DIRECTORIES>

<UI>
    <uiType>shell</uiType>
    <maxColumns>80</maxColumns>
    <maxRows>25</maxRows>
    <tabSize>4</tabSize>
    <helpFile>c:/cockpit/cockpitHelp.help</helpFile>
</UI>

<COMMANDLIST>
    <class>
        <name>com.ls.cockpit.Test1</name>
        <parameters>
            <parameter>test1</parameter>
        </parameters>
        <command>
            <name>test1</name>
            <alias>t1</alias>
            <description>
                <short>executes the command test1</short>
                <long>
                    <usage>test1</usage>
                    <parameter>-test1</parameter>
                    <example>
                        test1 -test1 test1
                    </example>
                </long>
            </description>
            <dependencies>
                <command>Disconnect</command>
            </dependencies>
        </command>
        <command>
```

```
                <name>test2</name>
                <description>
                    <short>executes the command test2</short>
                    <long>
                        <usage>test2</usage>
                        <parameter>-test2</parameter>
                        <example>
                            test2 -test2 test2
                        </example>
                    </long>
                </description>
            </command>
        </class>
    </COMMANDLIST>

    <ALIASES>
        <class>
            <name>com.ls.cockpit.StandardCommandLib</name>
            <command>
                <name>ping</name>
                <alias>p</alias>
            </command>
            <command>
                <name>logSystem</name>
                <alias>sys</alias>
            </command>
        </class>
        <class>
            <name>com.ls.cockpit.Connect</name>
            <command>
                <name>connect</name>
                <alias>con</alias>
            </command>
        </class>
    </ALIASES>
</CONFIG>
```

This is just an example. The details will be explained later.

Main TAGs of the configuration file

TAG	Child TAGs	Description
LOGGING	logFile	specifies, where the log file shall be stored. Default is ‘cockpit.log’.

	logLevel	specifies the level to log with (see the logging documentation, what kind of log levels exists). Default is 'TRACE5'.
CONNECTION	name	specifies the connection session name. If no name is given it will be set to default. Default is 'default'.
	platformId	specifies the platform ID you want to connect to. Default is 'lars'.
	platformIp	specifies the IP address you want to connect to. Default is '127.0.0.1'.
	platformPort	specifies the port on which the platform is listening to. Default is '1099'.
	connectionType	specifies the connection type you want to connect with, like RMI, Socket, JSocket, JSecureSocket,. Default is 'RMI'.
	keyFile	specifies the security key file for encrypting messages. (This is no longer used because you have to pay for the file and a secure connection is also obtained by using a combination of asymmetric and symmetric encryption with jSecureSocket). Default is an empty String.

	protocol	specifies the protocol to be used for the key file (no longer used, see above). Default is an empty String.
	provider	specifies the provider for the key file where the implementation of using the key file is located (no longer used, see above). Default is an empty String.
	user	specifies the user name for connecting to any LARS platform using the jSecureSocket. The user has to be entered in the corresponding configuration file of the running LARS platform. Default is an empty String.
	password	specifies the password for the user above. Default is an empty String.
	larsDirectory	specifies the directory where the main class of the LARS platform (in the package) is located. This is used if at any time it will be implemented to start a LARS platform from the cockpit. Default is './com/ls/lars2'.
	messageLength	specifies the maximum length of messages that can be received. Default is '32768'.

COMPRESSION	compressionType	specifies the type of the compression. There are three possibilities: 'no_compression', 'zip' and 'gzip'. If you're using compression I recommend you use gzip compression. Default is 'no_compression'.
	autoCompression	specifies if the auto compression is to be turned 'on' or 'off'. If the auto compression is turned on, the messages will be compressed with the specified compression type. If they are bigger than the specified compression type, the auto compression start size is used. Default is 'off'.
	autoCompression StartSize	specifies the start size with which messages should be compressed if auto compression is activated. Default is '16384'.
	compressionLevel	specifies the compression level to compress the messages with. This feature is only used on zip compression. Therefore the compression level '0' means no compression and compression level '9' means maximal compression. Default is '9'.

	zipEntryName	specifies the zip entry name. It doesn't really matter, what name is used, because this feature isn't used. The zip entry name is normally used in zip compression to name the entries (in most cases the file names). Default is 'Message'.
MONITORING	agentList	specifies an agent list file. The agent list file contains the agents listed in the agent manager's configuration file. An agentList file will be described later on in this document. With this TAG you just specify the file name or a list of file names.
DIRECTORIES	globalDir	specifies the global directory valid for all possible entries where a path or file is specified in this configuration file.
	helpDir	specifies the default search path for loading any help file during runtime.
	historyDir	specifies the default search path for stored history messages.
	commandListDir	specifies the default search path for stored command list. A command list file will be described later on in this document.
	monitorDir	specifies the default search path for stored monitor list files. This means it is the search path for monitoring list files.
UI	uiType	specifies the type of user interface. Currently there is only 'shell' implemented. The ui type 'swing' will soon follow.

	helpFile	Specifies where the help file can be found. You can set the help files as a normal string, or within the configuration files using a URL. An example will be shown below.
User interface type specific settings. Here for ui type shell.	maxRows	specifies the maximum view of rows. Default is 25.
	maxColumns	specifies the maximum view of columns. Default is 80.
	tabSize	specifies the tabulator size. Default is 4.
	minRestSpace	specifies the minimum rest space allowed. If there is a hierarchical view, with tabulators and the minimum rest space is reached before line breaking, the tabs will be reduced until the minimum rest space is smaller than the possibility of line breaking.

The help files can be in the jar file. To load files from a jar file you have to use a URL string. The URL string will be generated if you specify the helpfile as shown in the following example.

```
<helpFile>
    <protocol>file</protocol>
    <archive>c:/cockpit/lib/lscockpit_3_1.jar</archive>
    <name>com/cockpit/HelpFiles/cockpitHelp.help</name>
</helpFile>
```

TAGs for setting 'aliases' of commands

TAG	Child TAGs	Description

ALIASES	class	'Aliases' is the root TAG after which other TAGs can be defined i.e. a single class TAG for only one class, or a list of class TAGs for specifying more than one class to set aliases for.
class	name	specifies the class name. E.g.: 'com.ls.cockpit.Connect'
	command	specifies one single command or a list of commands.
Command	name	specifies the original name of a command. E.g.: 'connect'
	alias	specifies the alias to which the command will be mapped. E.g.: 'con'

TAGs for starting new command classes

This can be in a separate file and can be loaded via command. But if you want to start external command classes on startup of the cockpit, it has to be set it in the configuration file.

TAG	Child TAGs	Description
COMMANDLIST	class	'COMMANDLIST' is the root TAG. After that can be a single class TAG for only one class, or a list of classes TAGs for specifying more than one class.
class	name	specifies the classes name. E.g.: 'com.ls.cockpit.Connect'
	parameters	specifies the parameters of a command class. In between this TAG you can do what you want, because you have to care about it what happens with this values by your own. The parameters a classes specific, so they wont be used in parent class.

	command	specifies one single command or a list of commands.
command	name	specifies the original name of a command. E.g.: 'connect'
	alias	specifies the alias to which the command should be mapped. E.g.: 'con'
	dependencies	specifies the commands , which the current command depends on. E.g.: You can only send messages if you are connected. To be able to specify more than one command, you have to specify the commands between the 'command' TAGs. E.g.: '<dependencies><command> connect</command></dependencies>'
	description	specifies the description of the current command. There are two more TAGs. One for the short description and one for the long description. See below for these TAGs.
description	short	specifies the short description. In this case only a single String.
	long	specifies the long description. There are three more TAGs that can be used. The first TAG is the 'usage' TAG, the second TAG is the 'parameter' TAG, which can be used than once for a list and the last TAG is the 'example' TAG, where you can specify en example. See the example above for more details.

The monitoring list file

```
<LISTINDEX>
    <LIST>
        <service>start_agent</service>
        <content>
```



```
            <class>com.ls.lars.server.AgentPingPong</class>
            <name>app</name>
            <cfgFile>lars3/AgentPingPong.cfg</cfgFile>
        </content>
    </LIST>
    <LIST>
        <service>start_agent</service>
        <content>
            <class>
                com.ls.internet.generic.AgentExchange
            </class>
            <name>AgentExchange</name>
            <cfgFile>
                LivingAuctions/AgentExchange.cfg
            </cfgFile>
        </content>
    </LIST>
</LISTINDEX>
```

This file is exactly the same as the AgentManager configuration file except for a few changes. It differs from the AgentManager config file with the rootTAG and the TAG for each agent (LIST instead of MESSAGE).

The class and config files are not really needed for monitoring. However, to be able to start agents that contain the monitoring list file, the class and config files should be set.

8.3.2 Writing a help file for cockpit

The example of the help file shown below can be used for the shell implementation. It can also be used for other implementations. It will be used for other implementations as a default help file, but there will be some additional things. For example the long description of the shell implementation can't be used for the swing implementation, but the short description can be used.

```
<HELP>
    <class>
        <name>com.ls.cockpit.CommandManager</name>
        <command>
            <name>commandPath</name>
            <description>
                <short>
    shows the default command path to load command files from
                </short>
                <long>
                    <usage>commandPath</usage>
                </long>
            </description>
        </command>
        <command>
            <name>setCommandPath</name>
```

```
            <description>
                <short>
    sets the default command path to load command files from
                </short>
                <long>
                    <usage>setCommandPath</usage>
                    <parameter>-name [pathName]</parameter>
                    <example>
                        setCommandPath c:/command
                    </example>
                </long>
            </description>
        </command>
    </class>

    <class>
        <name>com.ls.cockpit.Quit</name>
        <command>
            <name>quit</name>
            <description>
                <short>
    disconnects from platform (if connected) and ends cockpit
                </short>
                <long>
                    <usage>quit</usage>
                </long>
            </description>
        </command>
    </class>
</HELP>
```

The structure of the TAGs used is similar to the structure of the command list file used to instantiate new command classes. The only thing to look out for as shown above is the 'parameter' TAG which looks like this: '<parameter>-name [pathname]</parameter>'. The squared brackets are important because of the XML structure. You can't take spike brackets ('<', '>'), but if you want to take it just use '<' for '<' and '>' for '>'.

8.4 Using Commands

8.4.1 Introduction

The parameters for commands can be set in two ways. The first way is if the parameter is just a string. This causes the shell implementation. The string will be parsed and then converted to a Map. It really makes no sense to have only strings as parameters, for example, within a graphical user interface; the parameters can be set directly, as they are needed. The second way to set the parameters would be as a Map. The

two forms of setting the parameters will be described in the standard implementation of the cockpit.

8.4.2 CommandManager

Command 'commandPath'

The command 'commandPath' shows the current valid path to load command files from. It returns a String containing the current valid command path. Parameters are not needed.

Command 'setCommandPath'

The command 'setCommandPath' sets the current valid path to load command files from. It returns a String containing the result of setting the path. The parameter is just a String containing the path you want to set to.

Command 'loadCommandPath'

The command 'loadCommandPath' loads a command file containing the command classes to be started. The command file has to be a valid command file as described below. If the command file is a valid command file the classes will be instantiated automatically. It returns a String containing the result of loading the command file and instantiating the classes. The parameter is just a String containing the command file with the full path, or only the command file on its own. If only the command file is set, the command path will be added in front of it.

8.4.3 Connect

There are two ways of specifying the parameters. Firstly a single string can be used which generates shell implementation and the second is to use a Map containing the keys and values to be changed.

The following keys are valid when a string is used:

- -name
- -id

- -ip
- -port
- -type
- -key
- -protocol
- -provider
- -user
- -password
- -larsDirectory
- -messageLength

The following keys are valid when a Map is used:

- name (NAME_TAG)
- platformId (PLATFORM_ID_TAG)
- platformIp (PLATFORM_IP_TAG)
- platformPort (PLATFORM_PORT_TAG)
- connectionType (CONNECTION_TYPE_TAG)
- keyFile (KEY_FILE_TAG)
- protocol (PROTOCOL_TAG)
- provider (PROVIDER_TAG)
- user (USER_TAG)
- password (PASSWORD_TAG)
- larsDirectory (PLATFORM_DIRECTORY_TAG)
- messageLength (MESSAGE_LENGTH_TAG)

To be sure the right entries are always used; the String constants defined in 'com.ls.cockpit.ICockpitConstants' should be used. Otherwise some changes might be lost.

When using the string parameters, constants are not defined for all entries, because the user enters the string in a shell like implementation. The string(s) are stored in String-Arrays. In other implementations where you use TextFields, CheckBoxes and similar

things, the Map implementation should be used, otherwise the string has to be parsed and this takes a lot of time.

Command 'connect'

The command 'connect' connects to any LARS platform. Generally the following parameters should be specified:

1. the LARS ID for the required LARS
2. the IP address for the machine the required LARS is running on
3. the connection type required (please note that there has to be a listener for the connection type required)
4. the port , which the listener of the connection type is listening on.

If a connection is specified in the configuration file without a name, the name will be set to 'default'. This default name will then be used for session handling.

Connection can be made without any parameters, in which case the default settings specified in the configuration file will be used for the connection.

If there are no default settings given the defaults from the cockpit will be used.

If another session is specified (defined in the config file), the settings from the specified session will be used. For example, if a connection setting called 'hello' is defined in the config file and the User types (in shell version) 'connect hello', the user will be connected to the platform using connection settings specified in the 'hello' connection settings.

You can also connect by specifying the connection parameters directly by typing (in shell version) 'connect –type socket –port 2005 –ip 192.168.0.1 –id lars' for example. The order doesn't matter. It might look a bit different in the other user interface implementation.

Command 'disconnect'

The command 'disconnect' disconnects the cockpit from the platform it is connected to. It returns the result of disconnecting from the platform. Parameters aren't needed.

Command 'connection'

The command 'connection' shows the different connections available. Just typing 'connection' shows the default connection settings. If you want to get a list of the available connections just type 'connection all'. To show a special connection you can type 'connection' and the desired connection like 'hello'.

Command 'setConnection'

The command 'setConnection' sets the different connections. By using 'setConnection' without specifying a name, the default connection will be changed. To change another connection setting other than the default connection, the name of the connection to be changed has to be specified. See above for all settings that can be changed.

It is generally used like the 'connect' command.

Command 'currentConnection'

The command 'currentConnection' shows the connection settings the user is currently connected with. If the settings have been changed, it returns the result of the command with the parameters that are not needed.

Command 'removeConnection'

The command 'removeConnection' removes the specified connection from your connection entries. It returns the result of the command. The parameter needed is the name of the connection to be removed. If a connection name is not specified, the default connection will be removed.

8.4.4 Compression

There are two ways of specifying the parameters. Firstly a single string can be used which generates the shell implementation and the second way is to use a Map containing the keys and values to be changed.

The following keys are valid when using the string:

- -type

- -auto
- -size
- -level
- -name

The following keys are valid when using the Map:

- compressionType
  (COMPRESSION_TYPE_TAG)
- autoCompression
  (AUTO_COMPRESSION_TAG)
- autoCompressionStartSize
  (AUTO_COMPRESSION_START_SIZE_TAG)
- compressionLevel
  (COMPRESSION_LEVEL_TAG)
- zipEntryName
  (ZIP_ENTRY_NAME_TAG)

To ensure that the right entries are used, please use the String constants defined in 'com.ls.cockpit.ICockpitConstants'. Otherwise some changes might be lost.

When using the string parameters, constants don't have to be defined for all entries, because the user enters the string in a shell like implementation. The string(s) are stored in String-Arrays. In other implementations where you use TextFields, CheckBoxes and similar things, the Map implementation should be used, otherwise the string has to be parsed and this takes a lot of time.

Command 'compression'

The command 'compression' shows the current compression settings. It returns the result (the settings in a Map) of the connection settings. Parameters aren't needed.

Command 'setCompression'

The command 'setCompression' sets the compression settings. It returns the result of changing the compression settings. The parameter needed is a String containing the values to be changed or a Map.

8.4.5 History

The only way to handle the parameters is by using a single String, because only one parameter is used for each command. (E.g.: name of a file or path, number of a message,)

The history stores all outgoing messages to a list with the exception of normal 'ping' messages. The reason is to make it possible to store such messages and send it again at any other time.

The following keys valid when using the string:

- -name
- -num

Command 'history'

The command 'history' shows the current history entries containing messages in a list. It returns the result (the Messages in a list, a single Message or a String). Parameters aren't needed.

Command 'historyEntry'

The command 'historyEntry' searches through the history entries and shows the specified message entries like 'service', 'receiver', etc. For example, this can be used to get all messages (message numbers), which get the agent 'hello'. The parameter has to be a valid message 'TAG' like service, sender and receiver. See the new Message documentation for valid message TAGs.

Command 'loadHistory'

The command 'loadHistory' loads the specified history file. This can be used to send messages sent before without typing in all the content. It can also be used to send messages like a script, by sending all messages. It returns the result of sending the message. The parameter needed is the file name of the history file. You can give the name with the full path, or just use the name without specifying the path. The specified commandListPath will then be used.

Command 'saveHistory'

The command 'saveHistory' saves the current messages containing the history list to the specified history file. It returns the result of saving the history messages. The parameter needed is the file name of the history file. You can give the name with the full path, or just use the name without specifying the path. The specified commandListPath will then be used.

Command 'clearHistory'

The command 'clearHistory' clears the complete history entries if nothing is specified. If any message number is specified which contains the history, only the specified message will be deleted from history.

Command 'sendHistory'

The command 'sendHistory' sends all messages containing the history if no message is specified. Otherwise it will only send the specified message. Normally the messages sent from the history file will be added to the outbox and the history file itself once again. But if all messages will be sent it makes no sense to add all the messages to the history file once again. In this case the messages won't be added to the history file.

Command 'historyPath'

The command 'historyPath' shows the current valid default path for saving and loading history files. If no path is specified, the user has to add the full path or the history files will be stored in the path where cockpit was started.

Command 'setHistoryPath'

The command 'setHistoryPath' sets the default path to load and save files. Only valid paths will be accepted, this means the path has to be available.

8.4.6 Inbox

The only way to handle the parameters is by using a String, because only one parameter is used for each command. (E.g.: name of a message field, number of a message,)

All messages coming in are visible in the Inbox. Messages to be sent to the cockpit by any agent running on the LARS platform can also be seen. In most cases, these will be messages that are responses to previously sent messages. For example, when a ping message is sent, a pong message is sent back as the reply. This might not be a very good example, because a ping message won't be stored in the inbox and the outbox if the ping command is used directly. But if the send command is used to send a ping message, it will be stored in the outbox and the answer 'pong' will be stored in the inbox.

The following keys are valid when using the string:

- -name
- -num

Command 'inbox'

The command 'inbox' shows the current inbox entries containing messages in a list. It returns the result (the Messages in a list, a single Message or a String). Parameters aren't needed.

Command 'inboxEntry'

The command 'inboxEntry' searches through the inbox entries and shows the specified message entries like 'service', 'receiver', This can be used to get all messages (message numbers), which get e.g. the agent 'hello'. The parameter has to be a valid message 'TAG' like service, sender and receiver. See the new Message documentation for valid message TAGs.

Command 'clearInbox'

The command 'clearInbox' clears the complete inbox entries if nothing is specified. If any message number is specified containing the inbox entry, only the specified message will be deleted from inbox.

8.4.7 Outbox

The only way to handle the parameters is to use a String, because only one parameter to be used for each entry. (E.g.: name of a message field, number of a message).

All out going messages are stored in the outbox, with the exception of the 'ping' command. The messages associated with the 'ping' command won't be stored, because it makes no sense to store messages just to see if an agent sends a pong in response to the ping command.

The following keys are valid when using the string:

- -name
- -num

Command 'outbox'

The command 'outbox' shows the current outbox entries containing messages in a list. It returns the result (the Messages in a list, a single Message or a String). Parameters aren't needed.

Command 'outboxEntry'

The command 'outboxEntry' parses through the outbox entries and shows the specified message entries like 'service', 'receiver', etc. For example, this can be used to get all messages (message numbers), which get the agent 'hello'. The parameter has to be a valid message 'TAG' like service, sender and receiver. See the new Message documentation for valid message TAGs.

Command 'clearOutbox'

The command 'clearOutbox' clears the complete outbox entries if nothing is specified. If any message number is specified containing the outbox entry, only the specified message will be deleted from the outbox.

8.4.8 Monitor

The monitor can be used to monitor any agent. This means that sending pings and receiving pongs, or getting a delivery message from the MessageRouter can monitor agents.

If a ping is sent to an agent containing the monitoring list and the agent responds with a pong, this proves that the agent is present and can react to messages. If the MessageRouter sends a delivery message, this means that it has no entry of a messenger for this agent. This means that the agent isn't present on this platform. If the agent doesn't respond within a period of time and the MessageRouter doesn't send a delivery message, this means that a messenger is present for this agent, but the agent is not responding. This could mean that the agent is present, but has so much to do that it can't or the agent is dead.

The only way to handle the parameters is using a single String, because where is for each command only one parameter to be used. (E.g.: name of an agent, class containing the full package structure, cfg the configuration files for this agent,)

The following values are mainly used for adding agents to monitor.

The following keys are valid when using the String:

- -name
- -class
- -cfg
- -xml

The following keys are valid when using the Map:

- name (NAME_TAG)
- class (CLASS_TAG)
- cfgFile (CFG_FILE_TAG)
- xml (XML_TAG)

Command 'monitor'

The 'monitor' command shows the state of each agent containing the monitor. There are three states. The first is that the agent responds and is active. The second is that the agent isn't present on this platform and the third that the agent doesn't respond. If the third case is valid, the agent is probably dead. It returns the result (the agents in a list with the state of each agent). Parameters aren't needed.

Command 'addMonitor'

The command 'addMonitor' adds the specified agent to the monitoring list. But it is recommended that the corresponding configuration file be also specified. The class name has to be specified; otherwise the agent won't be added to the monitoring list. This is to ensure that the agent can be restarted if necessary. There are two ways to specify the agent, either by specifying the name, class and configuration files, or by specifying it within an xml structure as a combination of the three files.

Command 'clearMonitor'

The command 'clearMonitor' clears the specified agent from the monitoring list. If no agent is specified the complete monitoring list will be cleared.

Command 'monitorPath'

The command 'monitorPath' just shows the current valid default path for loading monitoring files. If no path is specified, the full path has to be added by the user or the monitor files will be loaded from the path where the cockpit was started.

Command 'setMonitorPath'

The command 'setMonitorPath' sets the default path to load monitoring files. Only valid paths will be accepted, this means that the path has to be available.

Command 'loadMonitor'

The 'loadMonitor' command loads any specified monitoring file, parses it and gets the state of the agents within the monitoring list. The file can be accessed by loading it with the file name.

Command 'restartMonitor'

The 'restartMonitor' command restarts the specified agent. There are two possibilities. First is that the agent has started already and will be restarted by using this command and by specifying the agent with the specific class and configuration files. The second possibility is that the agent has not been started at all and will be started by using the specified agent name with the specific class and configuration files. To restart any agent just call this command and specify the agent to be restarted.

Command 'updateMonitor'

The 'updateMonitor' command updates the state of the monitoring agents. The state is usually updated automatically, but if there is ever any problem with updating the state, it can be done manually with this command.

8.4.9 Help

Help is only used to display a description of the available commands. A description is only available if a valid help file was loaded before. By default it will use the help file of the 'shell' implementation if available. The help file for the shell implementation should implement a description of all commands that can be used by all other user interface implementations.

Command 'help'

There is no parameter available for the 'help' command. Just using this command returns a list of all available commands containing a short description if available. To get more information, the command for which a better description is required should be specified. If a better description is available it will be returned. The return value can be a string, a list, a map, etc. Any object is allowed and can be returned, it depends on the help file.

To specify a command just use the argument wit which to specify it. If you want to get the help for the command ‘connect’ you have to execute the command ‘help’ containing the argument ‘connect’. If a help description is available it will be returned. Otherwise it will return a String with ‘No description available!’

Command ‘helpPath’

The ‘helpPath’ command shows the current valid default path from which help files are loaded. If no path is specified, the full path has to be added or the help file will be loaded from the path where the cockpit was started.

Command ‘setHelpPath’

The command ‘setHelpPath’ sets the default path to load help files from. Only valid paths will be accepted, this means that the path has to be available.

Command ‘loadHelp’

The command ‘loadHelp’ loads any specified help file, parses it and sets the description for all available commands. You can access the file by loading the command with the specific file name.

8.4.10 Quit

Command ‘quit’

This ‘quit’ command ends the cockpit. If the user is already connected, it will try to disconnect from any platform the user is connected to and then the cockpit program will be terminated.

8.4.11 StandardCommandLib

The StandardCommandLib defines all standard commands with which to send standard messages to the LARS platform. Most of these messages are interpreted by the AgentManager e.g. ‘new agent’, ‘delete agent’,

A receiver isn't needed for these commands. There are some special TAGs like 'class', 'cfg', … which are used to start e.g. a new agent. This information can also be set by using the xml TAG.

The following keys are valid when using the string:

- -service (used in messages for setting the service)
- -receiver (used in messages for setting the receiver, it can be a string or a list of receivers)
- -replyWith (used in messages for setting the replyWith)
- -inReplyTo (used in messages for setting the inReplyTo)
- -type (the message type: single, serviceSingle, serviceBroadcast, group, multicast)
- -platform (used by service messages for the platform to send messages to)
- -xml (used for content, specified as an xml content, an xml valid structure or a Map)
- -string (used for content, specified as a string content)
- -priority (the priority of a message)
- -expireHops (the expireHops of a message)
- -admin (the administrators name if needed)
- -pwd (the administrators password if needed)
- -name (a name for an agent to be started, deleted, …)
- -class (the class to start a new agent, e.g. 'com.ls…')
- -cfg (the configuration file(s) used for starting an agent)
- -inboxNumber (the message number containing the inbox entry to be forwarded, or to create a reply for)

The following keys are valid when using a Map:

- service (SERVICE_TAG)
- receiver (RECEIVER_TAG)
- replyWith (REPLY_WITH_TAG)
- inReplyTo (IN_REPLY_TO_TAG)
- type (TYPE_TAG)

- platform (PLATFORM_TAG)
- xml (XML_CONTENT_TAG)
- string (STRING_CONTENT_TAG)
- priority (PRIORITY_TAG)
- expireHops (EXPIRE_HOPS_TAG)
- admin (ADMINISTRATOR_TAG)
- pwd (PASSWORD_TAG)
- name (NAME_TAG)
- class (CLASS_TAG)
- cfgFile (CFG_FILE_TAG)
- inboxNumber (INBOX_NUMBER_TAG)

Command 'send'

The command 'send' sends any message to any agent. The command send is absolutely flexible. You can send different possible types of messages.

The TAGs always needed for this command are the 'service', and receiver TAGs. Additional information is not necessary in each message. It depends on the service. E.g. if you just want to wake up an agent, the agent just has to get a wakeup message with no content. But if the agent needs more information the content has to be set. There are two types of content. The string content is used if you just have a single string, e.g. '<content>hello</content>' (the content TAGs aren't needed). The xml content will be parsed by an xml parser and added to the message. Examples of how it is used in shell implementation are shown below in the shell description.

The replyWith TAG specifies the id with which to answer the message. This means that if a ping is sent and the replyWith was '1' the pong will have a '1' in the 'inReplyTo' TAG.

The type TAG specifies the kind of message to be sent like 'single message', 'service single message'. On service messages, no receiver is needed, but it is possible to specify the platform to send a service message to.

On all messages, a priority for the message and the expired hops for which the message is valid can be set. This means that if a message has the value '2' for expireHops and the message is sent through three

platforms, the message will be deleted. It's like the hop limit flag in the IP Header.

Command 'sendSync'

The command 'sendSync' is used in exactly the same as the command 'send'. The only difference is that the message is sent synchronously. This means that the message is sent and the agent will wait for an answer. The agent is blocked until the answer comes or a timeout is thrown.

The command needs also a valid replyWith TAG. If the TAG isn't set, it will be generated automatically with a replyWith id. It will return the answer message instead of a string if the message was successfully sent.

Command 'createReply'

The command 'createReply' creates a reply message for a message in the inbox. There are three TAGs to be specified. The first is the service. The second is to specify the content. The content can also be null. The third TAG to specify is the message number of the message in the inbox to create a reply for.

The result of creating a reply for a message is that the message is successfully sent, or it fails. The result is just a string like most of the results.

To specify the service, inbox number and content just use the TAGs as described above.

Command 'forwardMessage'

The command 'forwardMessage' forwards any message to one or more receivers. You can forward a message by just specifying the receiver(s) as described above and specifying the message number for the message within the inbox to be forwarded.

Command 'new'

The command 'new' is for starting a new agent for any agent's class. There are two or more things to be specified in order to start an agent. The first is the agent's name with which the agent will be registered on the platform. The second is the class name e.g.

'com.ls.lars.server.TestAgent'. If the agent needs any configuration file to start, one or more configuration files can be specified. If there is more than one configuration file to be specified the file names has to be in a list.

There is a default name for the AgentManager. If the AgentManager's name differs from the default, the receiver containing the AgentManager's name has to be specified. The result is usually a message returned from the LARS that the new agent has been started.

Command 'restart'

The command 'restart' first deletes an agent and then starts it again. It won't take the agent's source from cache; it will be started newly irrespective of the agent being present before or not. This is very useful if an agent is being developed and something new is to be tested. The LARS platform does not have to be restarted, it is possible just to restart the agent and the changed code will be used.

For restarting any agent just specify the required agent's name. A message will be sent back to indicate if the start was successful or not.

There is a default name for the AgentManager. If the AgentManager's name differs from the default, the receiver containing the AgentManager's name has to be specified. The result is usually a message returned from the LARS that the agent has been restarted.

Command 'reload'

The command 'reload' does exactly the same as the command restart as described above. The difference is that you have to specify the agent name, the class name and the configuration files as it is used for the command 'new'.

There is a default name for the AgentManager. If the AgentManager's name differs from the default, the receiver containing the AgentManager's name has to be specified. The result is usually a message returned from the LARS that the agent has been reloaded.

Command 'delete'

The command 'delete' deletes the specified agent. To delete any agent you have to specify the agent's name. A message is usually returned stating that command has been executed. There is a default name for the AgentManager. If the AgentManager's name differs from the default, the receiver containing the AgentManager's name has to be specified.

Command 'kill'

The command 'kill' kills the specified agent. To kill any agent you have to specify the agent's name. A message is usually returned stating that command has been executed.

The difference between delete and kill is that the delete command tries to delete the agent. If the agent is busy and has to finish its job, it won't be deleted. The kill command kills an agent irrespective of the agent being busy or not. The agent will be killed even if it is just doing something on a database or any other task.

There is a default name for the AgentManager. If the AgentManager's name differs from the default, the receiver containing the AgentManager's name has to be specified.

Command 'list'

The command 'list' lists all agents running on the platform. Parameters aren't needed. The response to the message is a string. The response message contains the inbox and then in the content of the message all agents available on the platform will be shown.

There is a default name for the AgentManager. If the AgentManager's name differs from the default, the receiver containing the AgentManager's name has to be specified.

Command 'listPlatforms'

The command 'listPlatforms' equals the command list in usage, but not the running agents will be shown, it will be shown all available platforms if some platforms are synchronized together.

There is a default name for the AgentSynchronizeSupervisor. If the AgentSynchronizeSupervisor's name differs from the default, the receiver containing the AgentSynchronizeSupervisor's name has to be specified.

Command 'logSystem'

The command 'logSystem' sends a message to log the system information to the log file. A receiver isn't needed. Normally this service is implemented as a service message. The result is the string indicating if the message was successfully sent or not.

Command 'shutdownLars'

The command 'shutdownLars' ends all agents by deleting them after they finish their jobs and then it stops the running LARS session. To do this just call the command by specifying the administrator's name and password as described above.

There is a default name for the AgentManager. If the AgentManager's name differs from the default, the receiver containing the AgentManager's name has to be specified.

Command 'restartGroup'

The 'restartGroup' command works exactly like the restart command, but it won't restart just one agent, it will restart all agents within the specified group.

Instead of the agent name, the group name has to be specified.

Command 'deleteGroup'

The 'deleteGroup' command works exactly like the delete command, but not just one agent will be deleted, it will delete all agents within the specified group.

Instead of the agent name, the group name has to be specified.

Command 'killGroup'

The 'killGroup' command works exactly like the kill command, but not just one agent will be killed, it will kill all agents within the specified group.

Instead of the agent name you have to specify the group name.

Command 'ping'

The command 'ping' sends a ping to the specified agent and expects a pong from this agent. It will be sent a synchronous ping and receive a pong or a timeout exception will be thrown. The return value is a message containing the pong if the pinged agent answered successfully.

The command ping can be sent, by just specifying the receiver.

8.5 Programming your own command classes

8.5.1 Writing a command class

There are several methods you have to implement. You can inherit from three classes. The first is the 'CommandTemplate', the second is the 'InternalCommand' and the third is the 'ExternalCommand' class.

It is recommended that the user should inherit from the 'InternalCommand' or 'ExternalCommand' class.

The three methods you have to implement are: 'init ()', 'Map getHelp ()' and 'Map getAlias ()'.

The init () method is for doing initial things, i.e. whatever needs to be initialized. The getHelp () method is used to get the help information if some help is hardcoded in the class. Otherwise null or a new Map can be returned. The getAlias () method is used to set aliases for any command. If aliases are not needed null or a new Map can be returned.

The structure of these Maps's has to be the same as that of the configuration file for the cockpit, or as it is specified in the help file or command list file, (see above - 'Configuring the Cockpit').

These methods will be called automatically when running the command class. The command class can be started by specifying it in the configuration file for the cockpit, or by using the command 'loadCommand [filename]' as specified above.

The best way to understand how it works is to see the example below.

An example of a class

```
package com.ls.cockpit;


import java.util.Map;
import java.util.HashMap;


import com.ls.InvalidArgumentException;


public class Test1 extends InternalCommand implements
    ICockpitConstants
{
```



```
public Test1(Map parameters, Map dependencies,
             Map description)
{
    super(parameters, dependencies, description);
}


protected void init()
{
}


protected Map getHelp()
{
    Map help    = new HashMap();
    Map help1   = new HashMap();

    help1.put("test1", "executes test1");
    help1.put("test1Execute", "executes test1Execute");
    help.put("short", help1);

    help1 = new HashMap();
    help1.put("test1", "Usage: test1 -hello you, ...");
    help1.put("test1Execute",
              "Usage: test1Execute -hello you, ...");
    help.put("long", help1);

    return help;
}


protected Map getAlias()
{
    Map alias   = new HashMap();
    Map alias1  = new HashMap();

    alias1.put("test1", "t1");
    alias1.put("test1Execute", "t1e");

    alias.put("alias", alias1);

    return alias;
}


protected Object commandTest1(Object arguments)
    throws CommandNotFoundException,
           CommandNotAvailableException,
           InvalidArgumentException
{
    return "command Test1 executed.";
}


protected Object commandTest1Execute(Object arguments)
    throws CommandNotFoundException,
           CommandNotAvailableException,
```

```
                InvalidArgumentException
    {
        return "command Test1Execute executed.";
    }
}
```

The constructor has to call the super method, otherwise the settings for the dependencies; parameters and description won't be set.

The Map parameter can be used for any specific command class settings. It can be used to create settings needed only by this class. If the parameters are stored in a class variable, this class variable can be used to do some initial settings by using the init method since the init method is called automatically.

The help method needs two settings a short description and a long description. The short description contains a Map with all commands relevant to this class. The value of these commands (i.e. the value of the 'keys' in the Map) does not need to be strings. They can be any hierarchical structure containing maps and lists. The same thing applies for the long description.

As shown in the example above, the aliases are set in a map containing all commands with their aliases. This map of aliases has to be put in a map with a key value 'alias'.

The Command methods as shown above have to be protected, returning an object and expecting an object. They have to throw the 'CommandNotFoundException', 'CommandNotAvailableException' and the 'InvalidArgumentException'. Otherwise the method won't be registered.

To register a method, the method name has to start with the prefix 'command'. The rest of the method name after the prefix will be used as the command name. For example if the method name is called 'commandHello' the command will be available by calling 'hello'.

8.5.2 Starting a command class

There are two ways of starting a command class. It has already been described above. Firstly it can be set it in the cockpit configuration file as described in the example of a configuration file, or it can be started by using the command 'loadCommand [filename]'.

It has been described above, but this is a short example:

```
<COMMANDLIST>
    <class>
        <name>com.ls.cockpit.Test1</name>
```



```
        <parameters>
            <parameter1>test1</parameter1>
            <parameter2>test2</parameter2>
        </parameters>
        <command>
            <name>test1</name>
        </command>
    </class>
</COMMANDLIST>
```

8.6 Using the user interface 'shell'

8.6.1 Introduction

The user interface and all its settings can be specified in the configuration file for the cockpit.

A description of the file has been given in the chapter on 'Writing a configuration file'. There are five possible settings for the shell implementation. Not all of them are needed because there are default settings for them. The possible settings are as follows:

- maxColumns (specifies the available columns)
- maxRows (specifies the available rows)
- tabSize (specifies the width of tabs)
- minRestSpace (specifies the minimum number of rows after any key containing a Map)
- helpFile (specifies the location and name of a help file)

The values for a help file are special . To make it possible to load a help file from a jar file, it is necessary to use a URL form. If the help file is anywhere on the harddisk, it can be specified as follows:

```
<helpFile>c:/cockpit/HelpFiles/cockpit.help</helpFile>
```

If the help file is located in a jar file, it can be specified as follows:

```
<helpFile>
    <protocol>file</protocol>
    <archive>lscockpitshell.jar</archive>
    <name>com/ls/cockpit/shell/HelpFiles/cockpit.help</name>
</helpFile>
```

8.6.2 Using the command line

After starting the shell implementation a string will be shown to indicate that the program has started e.g. FTP programs. The string is called 'Command>'. The cockpit now waits for user entries to execute any command. The command line shall be instinctive as all command lines are in any operating system. As in most operating systems, a help description can be obtained if the command was specified. A help description can also be obtained from the cockpit if a valid help file was specified in the configuration file of the cockpit. All available commands will then be shown with a short description. If a command is specified, a detailed description will be shown for that command.

There are many different ways to enter a command. Firstly the command has to be valid. Secondly commands and arguments should be separated by a white space. Therefore the command always has to be a string without any white spaces. The rest of the string entry is the arguments.

Generally there are three ways to call a command. The kind of user entry needed depends on the command.

(3) The first possibility is to execute a command without having arguments. An example of such a command is the command 'help'.

(4) The second way is a command with only one argument. If only one argument is needed, the argument can be specified by just typing the argument, or by using the valid TAG. An example of this is the help command specifying a command. The first possibility is to enter 'help connect' and the second is to enter 'help –name connect'.

(5) The third way to enter a command is if more than one argument is needed. Then the TAGs always have to be used. An example of this is the connect send command. For the send command, at least the service and the receiver parameters are needed. The command can be specified as follows: 'send –service ping –receiver am'.

If there is more than one receiver (e.g. for Multicast messages) or a list of configuration files, the receivers or config files can be separated with a comma.

It is quite difficult to parse the arguments because of so many different possibilities.

A complex example is as follows: 'send –service set_log –receiver aps –xml <logFile>test.log</logFile><logLevel>ERROR</logLevel>'

This might seem a bit easy, but it could become difficult in xml settings where anything is allowed. This means there can be white spaces, -anything and any other strings.

These arguments could be parsed as follows:

For each command class are different key values possible. For example in the command class 'Connect' are the following keys valid: '-name', '-id', '-ip', '-port', '-type', '-user', '-pwd', '-directory', '-messagelength', '-key', '-value'. This means, if you want to set the lars id for connection you can use the command 'setConnection -id lars'. With this command the id of lars for connection settings will be changed to 'lars'. Cause you have different connection sessions, and the session name wasn't specified it was set for the default connection. Otherwise you had to specify e.g.: 'setConnection -id lars -name test'.

It makes no difference in which order the key value pairs will be specified and how many of these key value pairs are used. These key value pairs are the arguments. In shell version it is only a String so it has to be parsed. The arguments '-id lars -name test' has to parsed to mappings id=lars, name=test then.

To parse this arguments it will be searched first for '-' characters. Afterwards it will be checked if the key '-xxx' (e.g. '-id') is a valid key for this command class. Now it will be checked if the next '-yyy' key (if present) is a valid key and isn't contained in a xml-structure. Then you can be nearly sure that the key and the following string until the next key is a valid argument setting and you can take this for a key value pair.

But one problem exists anymore. If you specify e.g. a name and the name contains a string that would be a valid key it won't work correctly (e.g. '-name xy-type', where '-type' would be a valid key). For such things you can quote this like it's known in most shells like in Windows, and most linux and unix shells (e.g. '-name "xy-type"').

8.6.3 Command line examples

Here are some command examples:

```
Command> connect –type jSocket –port 2003 –id lars –ip
192.168.0.2

Command> list

Command> new –name test –class com.ls.lars.server.TestAgent  -
cfg Test1.cfg, Test2.cfg

Command> new –xml <name>test</name>
<class>com.ls.lars.server.TestAgent</class><cfgFile>Test1.cfg</
cfgFile><cfgFile>Test2.cfg</cfgFile>
```

```
Command> send –service sql –receiver AgentSql –xml <sql>select
* from table where agentName=’Hello’</sql> <sql>select * from
hello where agentName=’Hello’</sql>

Command> send –service ping –receiver AgentSql, TestAgent,
HelloAgent –type multicast

Command> send –service ping –receiver AgentSql, TestAgent,
HelloAgent –type multicast –replyWith 1 –priority 10

Command> connect socket

Command> connect –name socket

Command> setConnection –name socket –type jSocket –port 2001 –
id lars3 –ip 192.168.0.100

Command> history

Command> history 5

Command> clearHistory 5

Command> loadHistory test1.msg

Command> saveHistory test1.msg

Command> createReply –inboxNumber 3 –service hello

Command> forwardMessage –inboxNumber 7 –receiver asl, ajsl

Command> logSystem asi
```

9 Appendix

9.1 Contact

9.1.1 living systems Web Site

http://www.living-systems.com

9.1.2 Technical Support

product.support@living-systems.com

9.1.3 Feedback

Please send all you feedback concerning

- Wish list
- Enhancement Requests

- Bug Reports

to:

product.feedback@living-systems.com

9.1.4 Subsidiaries

All living systems subsidiaries are autonomous with marketing and sales responsibilities, including software expertise in customizing our technologies locally and providing local support. All subsidiaries are wholly owned by **living systems AG**.

Global Headquarters:

living systems AG
Humboldtstraße 11
D-78166 Donaueschingen
Germany

Tel.: +49 (771) 8987-0
Fax: +49 (771) 8987-100

Boston:

living systems, inc.
150 Baker Avenue
Suite 108
Concord, MA 01742
Boston – USA

Tel.: +1 (978) 371-5500
Fax: +1 (978) 371-7203

London:

living systems UK Ltd.
78 Cannon Street
London EC4N 6NQ
United Kingdom

Tel.: +44 (207) 618-8580
Fax: +44 (207) 618-8583

São Paulo:

living systems Americas Ltda.
Rua Dr. Cardoso de Melo 1460, 10.
andar Vila Olímpia - SP - São Paulo
CEP 04548-005 – Brazil

Tel.: +55 (11) 3047-4635
Fax: +55 (11) 3047-4735

Singapore:

living systems Asia Pte. Ltd.
8 Temasek Boulevard
#25-02 Suntec Tower Three
Singapore 038988

Tel.: +65 (887) 5995
Fax: +65 (887) 4994

Timisoara:

s.c. living systems Romania s.r.l.
Str. Paris no. 2A
1900 Timisoara
Romania

Tel.: +40 (56) 29 37-26
Fax: +40 (56) 29 27-25

10 Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W