
AIMMS Multi Agent and Web Services User's Guide - Multi-Agent Reference

This file contains only one chapter of the book. For a free download of the complete book in pdf format, please visit www.aimms.com

Copyright © 1993–2011 by Paragon Decision Technology B.V. All rights reserved.

Paragon Decision Technology B.V.	Paragon Decision Technology Inc.	Paragon Decision Technology Pte.
Schipholweg 1	500 108th Avenue NE	Ltd.
2034 LS Haarlem	Ste. # 1085	80 Raffles Place
The Netherlands	Bellevue, WA 98004	UOB Plaza 1, Level 36-01
Tel.: +31 23 5511512	USA	Singapore 048624
Fax: +31 23 5511517	Tel.: +1 425 458 4024	Tel.: +65 9640 4182
	Fax: +1 425 458 4025	

Email: info@aimms.com
WWW: www.aimms.com

AIMMS is a registered trademark of Paragon Decision Technology B.V. IBM ILOG CPLEX and sc CPLEX is a registered trademark of IBM Corporation. GUROBI is a registered trademark of Gurobi Optimization, Inc. KNITRO is a registered trademark of Ziena Optimization, Inc. XPRESS-MP is a registered trademark of FICO Fair Isaac Corporation. MOSEK is a registered trademark of Mosek ApS. WINDOWS and EXCEL are registered trademarks of Microsoft Corporation. $\text{T}_{\text{E}}\text{X}$, $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$, and $\text{A}_{\text{M}}\text{S}_{\text{L}}\text{A}_{\text{T}}\text{E}_{\text{X}}$ are trademarks of the American Mathematical Society. LUCIDA is a registered trademark of Bigelow & Holmes Inc. ACROBAT is a registered trademark of Adobe Systems Inc. Other brands and their products are trademarks of their respective holders.

Information in this document is subject to change without notice and does not represent a commitment on the part of Paragon Decision Technology B.V. The software described in this document is furnished under a license agreement and may only be used and copied in accordance with the terms of the agreement. The documentation may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Paragon Decision Technology B.V.

Paragon Decision Technology B.V. makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Paragon Decision Technology B.V., its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claims for lost profits, fees or expenses of any nature or kind.

In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors shall not be responsible under any circumstances for providing information or corrections to errors and omissions discovered at any time in this book or the software it describes, whether or not they are aware of the errors or omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors do not recommend the use of the software described in this book for applications in which errors or omissions could threaten life, injury or significant loss.

This documentation was typeset by Paragon Decision Technology B.V. using $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ and the LUCIDA font family.

Chapter 3

Multi-Agent Reference

This chapter presents, in detail, the facilities to construct a multi-agent application in AIMMS. It is assumed that you have read the introduction presented in Chapter 1. Prior to reading this chapter, you may like to build a first example. In this case you are advised to follow the hands-on tutorial provided in Chapter 2.

This chapter

3.1 Requirements and limitations

The multi-agent technology of AIMMS is built on top of the commonly used Transmission Control Protocol (TCP) which is integrated in all Windows operating systems that are supported by AIMMS. Unless you deploy an agent in a community as a webservice, its use is currently restricted to a Local Area Network (LAN). If you do deploy an agent as a webservice, however, the agent community can be accessed over the Internet.

TCP

The current implementation of the AIMMS' multi-agent technology does not support sender authentication and encryption of messages. You are, therefore, advised not to use the current AIMMS agents in an environment where agent authenticity is important, or where agents exchange confidential information.

No authentication or encryption

To start projects on remote computers using the function StartRemoteProject described later in this chapter, AIMMS uses DCOM (Distributed COM) to remotely create AIMMS COM objects. The IT policy of your company, or the configuration of DCOM within your network, may limit your capabilities to use DCOM to create remote AIMMS COM objects. If you are experiencing problems starting remote AIMMS sessions, you should consult your IT department for further assistance.

Starting remote projects

Given the limitations outlined above, the current version of AIMMS' multi-agent technology can, for instance, be successfully deployed for

Usability

- the implementation of a distributed problem solving approach through coarse-grain parallelization, or
- a simulation of a market with optimizing agents.

To overcome some of the limitations listed above, Paragon will continue the development of its multi-agent technology. While the functionality at the AIMMS level will remain the same or be extended, at the transport level the following extensions are anticipated in the next generation of the AIMMS multi-agent technology:

Expected extensions

- security and authentication enhancements to allow use in business-critical environments.

With a message queuing service in place, messages can be sent back and forth between message queues running on separate computers in a local area network. Programs on these computers can access the message queue and read incoming messages. In the context of the multi-agent technology in AIMMS, each program has its own associated message queue to receive its own incoming messages. In AIMMS, messages can be sent to and from AIMMS projects and, thus, there must be a message queue for each participating project. As the developer of one or more agents inside a participating project, you have access to a dynamic link library to support you in sending and receiving messages. This dynamic link library contains all the code required to create and access message queues, and the interaction with this code is predefined through a special module written in the AIMMS modeling language.

Message queuing in AIMMS

3.2 Installing the agent module

By including the special AIMMS module for multi-agent applications, you have all the necessary machinery in terms of identifiers and procedures to construct agent-based programs using the AIMMS language. The module itself is read-only, which implies that you can reference its identifiers, but cannot modify them. By selecting the menu item **Settings-Multi Agent-Install Agent Module**, the module will be automatically added to your AIMMS project.

Multi-agent module

Unless otherwise indicated, all identifiers inside the multi-agent module are public. This means that you *may, but need not*, reference each identifier through the prefix `MultiAgent::`. Throughout the tutorial provided in Chapter 2 we consistently prefixed all identifiers declared in the multi-agent module. One advantage of such prefixing is that you can easily distinguish between predefined names and application-specific names. A disadvantage, however, is that identifier names can become quite lengthy. Multiple occurrences of the same prefix in one statement may also become irritating. If there are name clashes with identifier names in your model, you can always make the identifiers in the multi-agent module protected through the `Protected` attribute of the multi-agent module.

Identifiers are public

3.3 Community setup

A multi-agent community in AIMMS consists of a network of interacting agents in which each individual agent plays a specific *agent role*. Agents interact by sending messages back and forth. Their vocabulary is defined through predefined *message types*. There is strict mapping between message types and agent roles, and this determines the possible communications between individual members of the community.

*AIMMS agent
community ...*

An agent community is essentially set up through the specification of all agent roles and all message types, combined with a mapping between these roles and types that expresses permitted interaction within the community. Specifying or changing this community setup requires the use of the **Multi Agent Community Setup** dialog box already discussed in the previous chapter.

... and its setup

The information entered through the **Multi Agent Community Setup** dialog box is stored in a separate file, referred to as the *community setup file*. All agent projects in the community must have access to this file or to a copy of it. Multiple developers of a multi-agent application must ensure that changes to the community setup file are not made at the same time. Ideally, this file should be under the control of a single person who is in charge of the overall design of the multi-agent application.

*The community
setup file*

Through the menu command **Settings-Multi Agent-Select Community Setup File** you can select the configuration file name that you want to use within your multi-agent project. If you are starting a new multi-agent application, the command allows you to enter a new file name. Once a community setup file has been selected, the menu commands **Multi Agent-Community Setup** and **Multi Agent-Agent Setup** become available for further configuration of the agent community and individual agents. The community setup file is read anew each time either the **Multi Agent Community Setup** dialog box or the **Multi Agent Implementation** dialog box is opened.

Selecting the file

3.3.1 Agent roles

On the **Agent Roles** tab of the **Multi Agent Community Setup** dialog box (see Figure 3.1), the designer of the community maintains a list of all agent roles that appear in the community. You can add new agent roles, or delete existing agent roles, using the **Add** and **Delete** buttons. To add a new role, just enter a simple description such as Controller, or Customer. The community should have at least one agent role.

*Creating agent
roles ...*

All specified agent roles will automatically appear as data in the set `Multi-Agent::AllAgentRoles`. The contents of this set is fully defined through the **Multi Agent Community Setup** dialog box. Even though you cannot change the contents of this set within the AIMMS program, you can declare and define your own subsets of this set.

... adds data to a set

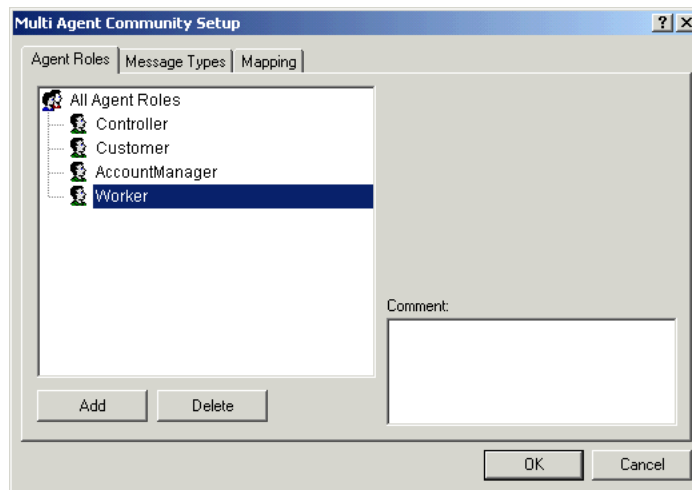


Figure 3.1: Declaring agent roles

3.3.2 Message types

On the **Message Types** tab of the **Multi Agent Community Setup** dialog box (see Figure 3.2), the designer of the community maintains a list of all the message types that can be used in the community. This list defines the entire language between the agents. You can add new message types, or delete existing message types, using the **Add** and **Delete** buttons. To add a message type, just enter a clear description such as `RequestHelp` or `RequestAgentStatistics`.

Creating message types ...

Each message type can have associated data entries. These entries are the arguments of the message type, and can refer to scalar or multi-dimensional data that are to be communicated when a message of the given type is sent. You can view the arguments of a message type by opening the message type entry in the dialog box. This expands the underlying branch to reveal all the arguments. Through the **Add** and **Delete** buttons, you can then add new arguments or delete existing ones.

... with arguments

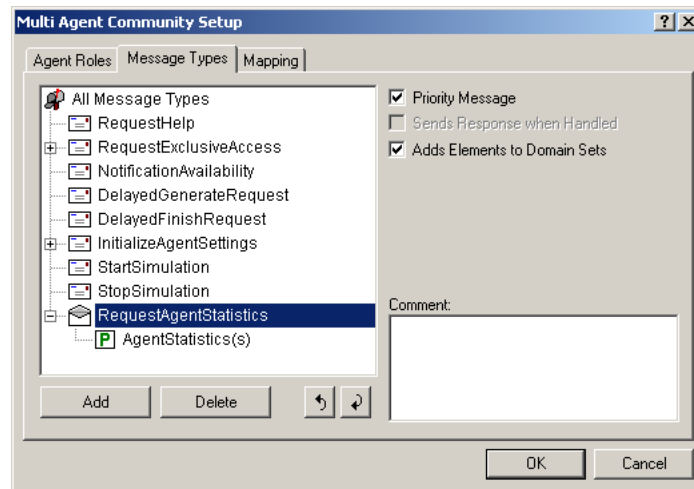


Figure 3.2: Declaring message types

You can create multi-dimensional arguments simply by typing the indices within parentheses. The index names that you provide are only used to determine the dimension and do not have to exist within your model. However, it is often convenient for the index names to coincide with common declarations of sets and indices if these exist inside the various agent projects. The same naming convention is used when AIMMS creates a skeleton for a message handler, and/or a send procedure, from within the **Multi Agent Implementation** dialog box as discussed in the next section.

... multi-dimensional

Each argument in a message type refers to either a parameter (numeric, string, element) or a set. A numeric argument is either a floating point number (double) or an integer. For an element-valued argument you also have to specify the range set. As with the indices, you may enter a non-existing set name.

... their type and range

The person in charge of the overall design of a multi-agent application must know whether or not units of measurement will be used by the developers of the individual agents in the community. If units are to play a role within the multi-agent community, then a unit must be specified for each numeric parameter. It is recommended that all developers use the same quantities and units.

... their unit

Each argument in a message type has a mandatory input/output property designation. An argument with the property `Input` indicates that data are to be passed from the sending agent to the receiving agent. An argument with the property `Output` indicates that data are to be received by the sending agent, from the receiving agent, at the time of sending. Therefore, the use of one or more output arguments implies message synchronization and causes the

... their property

sending agent to wait for the message to be handled completely. Output arguments should not be used when a sender needs to send a single message to several individual agents. In this case, unnecessary waiting can be avoided if the receiving agents send a separate asynchronous response message with the output data. If necessary, the sending agent can wait for these response messages before continuing its execution, by using one of the special synchronization functions offered by AIMMS (see also Section 3.5.6).

The designer of a multi-agent application may specify whether or not a particular message type has a priority status. Message types with a priority status require special attention by the designer, because messages of such type are handled by the receiving agent in between execution statements, and could have negative side effects if global data are overwritten. Typical message types with a priority status are those with output arguments only, and for which data must be obtained in a timely manner.

... and their priority

All specified message types will automatically appear as data in the set `Multi-Agent::AllMessageTypes`. The contents of this set are entirely defined through the **Multi Agent Community Setup** dialog box. Even though you cannot change the contents of this set inside the AIMMS program, you can declare and define your own subsets of this set.

... adds data to a set

3.3.3 Mapping

On the **Mapping** tab of the **Multi Agent Community Setup** dialog box (see Figure 3.3), the designer of the community can define a mapping between message types and agent roles. This mapping indicates the permitted interactions between individual agents within the agent community, and is used to guide the specification of individual agents through the **Multi Agent Implementation** dialog box discussed in Section 3.4.

Agent roles and message types

The mapping between agent roles and message types can be entered in one of two ways, and you can freely mix them.

Two entry modes ...

- For each particular message type you can indicate **From Agents** and **To Agents** as being the originators or recipients of messages of this type.
- For each particular agent role, you can indicate **Sends Messages** and **Receives Messages** as being the messages that can be sent or received by agents with this role.

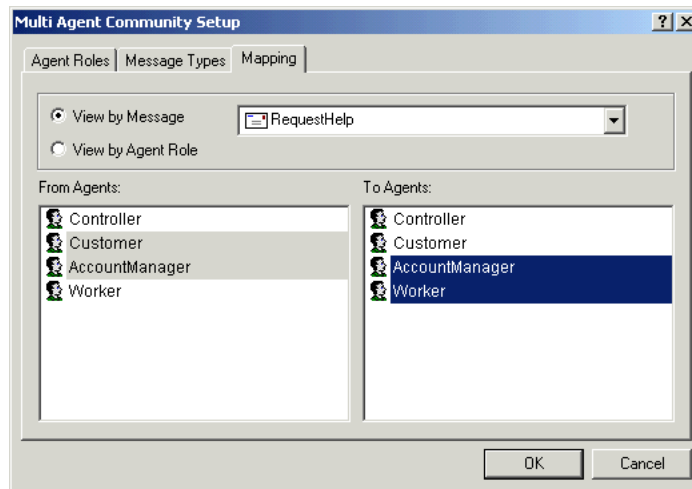


Figure 3.3: The **Mapping** tab of the **Multi Agent Community Setup** dialog box

The specified mapping between agent roles and message types will appear as data in the sets `MultiAgent::SendMapping` and `MultiAgent::ReceiveMapping`. These sets contain two-tuples with agent roles in the first component and message types in the second component. The set `MultiAgent::SendReceiveMapping` is derived from the above two sets, and contains three-tuples with the sending agent role, the receiving agent role, and the message type, as the three components. You can use these three sets to check whether an agent is allowed to send a message to a specific agent. The contents of these sets can only be determined through the **Multi Agent Community Setup** dialog box.

... adds data to sets

3.4 Agent setup

If you are going to build an AIMMS project for one or more particular agents in the community, you ensure that you include the special multi-agent module in your project, and select the proper community setup file, as described in the previous section.

Getting started

An important part of the implementation of a particular agent in the community is the specification of the names of the procedures through which the agent can send messages, and the names of the procedures through which messages arrive and are handled by this agent. In the **Multi Agent Implementation** dialog box you can define the link between such procedures and the corresponding message types for each particular agent role.

Multi Agent Implementation
dialog box

The community setup file is used by all agent projects in the community, and can be modified by the developers of these projects. As a result, a new version of this file could cause mismatches with the specification in your own agent project. If such mismatches occur, they will be indicated when you open the **Multi Agent Implementation** dialog box, or press the **OK** button.

Changes in community setup

3.4.1 Agent roles in project

On the **Agent Roles** tab of the **Multi Agent Implementation** dialog box (see Figure 3.4), you can select one or more agent roles to be implemented within the current project. In most multi-agent projects only one agent role is implemented. Selecting only one agent role does not imply that only one agent will be created through the project. The same agent project can be opened multiple times allowing for similar agents to be active in the community.

Selecting agent roles ...

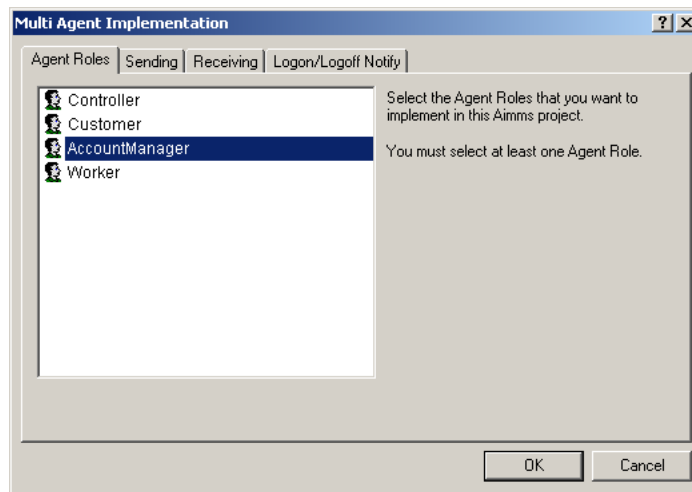


Figure 3.4: Specifying the agent role for an agent project

The agent roles that you select on the **Agent Roles** tab of the **Multi Agent Implementation** dialog box will automatically appear as data in the set `Multi-Agent::AllImplementedAgentRoles`, a subset of `Multi-Agent::AllAgentRoles`.

... adds data to a set

The selected agent roles determine the contents of the other tabs in the **Multi Agent Implementation** dialog box. These tabs will only show information that is relevant to the agent roles that you have selected for implementation. Accordingly, you must select at least one agent role before you can access any of these other tabs.

Other tabs

3.4.2 Send procedures

In AIMMS, to be able to send a message of a specific message type, you have to create a send procedure for it. The **Sending** tab of the **Multi Agent Implementation** dialog presents a list of the message types that are allowed to be sent by the agent role(s) you are implementing. For each message type in this list, you can specify whether you want to create a send procedure for it, and what the name of this send procedure will be.

Sending tab

When you close the dialog box, AIMMS will create this external procedure in the Multi Agent Send Procedures section of your model tree. If this section does not yet exist, it will be appended to the bottom of the tree. The contents of this external procedure describe the link to an external function in the multi-agent DLL. The procedure is generated by AIMMS and you should not try to make modifications to it manually.

Section Multi Agent Send Procedures

The argument list of each generated send procedure depends on the arguments of the message type. If the message type only contains input arguments, the send procedure will resemble:

Only input

- `Send_MyInputMessage(from_agent,to_agent<,arg1,...,argN>[,SendDelay])`

where `SendDelay` is an optional argument to allow AIMMS to hold the message for the given number of seconds. The use of this argument is further explained in Section 3.5.6. This send procedure will return 1 if the message is successfully delivered at the queue of `to_agent`.

If the message type has one or more output arguments, the send procedure will be of the form:

With output

- `Send_MyOutputMessage(from_agent,to_agent<,arg1,...,argN>[,TimeOutPeriod])`

where the optional argument `TimeOutPeriod` specifies the number of seconds to wait for the requested output values to be returned. The default value of this argument is 10 seconds. This send procedure will return 1 if the message is successfully delivered to the queue of `to_agent`, and the output values are received in time.

A send procedure returns 1 if successful, and 0 if an error has occurred. In the case of an error, you can call the function `MultiAgent::GetLastErrorCode` to obtain the error code of the specific error that occurred. To obtain a string representation of this error you can pass the error code `nr` to the function `MultiAgent::ErrorString(nr)`.

Error return

3.4.3 Handler and filter procedures

In AIMMS, messages that are sent to an agent in your project are passed on to a filter procedure and/or a handler procedure. The **Receiving** tab of the **Multi Agent Implementation** dialog presents a list of the message types that may be sent to an agent of the agent role(s) that you are implementing. For each message type in this list, you can specify whether you want to create a filter and/or handler procedure for it, and what the name of such a procedure will be.

Receiving tab

When you close the dialog box, AIMMS creates a skeleton for the selected handler procedure in the **Multi Agent Handlers** section of your model tree. If this section does not yet exist, it will be appended to the bottom of the tree. AIMMS only generates the procedure declaration together with the arguments that are imposed by the message type that it handles. The body of the handler is initially empty. The argument list of a message handler procedure always has the form:

Section Multi Agent Handlers

- `Handler_MyMessage(from_agent,to_agent<,arg1,...,argN>)`

As with the handler procedures, AIMMS creates a skeleton for the selected filter procedure in the **Multi Agent Filters** section of your model tree. Again, if this section does not yet exist, it will be appended to the bottom of the tree. The body of a filter procedure is initially empty. The arguments of a filter procedure are the same for each message type, namely:

Section Multi Agent Filters

- `Filter_MyMessage(from_agent,to_agent)`

3.4.4 Logon/logoff notification

If you want to trace which agents are entering or leaving the community, you can identify two procedures in your model which will be called automatically each time an agent logs on to, or logs off from the community.

When to use

On the **Logon/Logoff Notify** tab of the **Multi Agent Implementation** dialog box you can specify if you want to receive a notification of a log on or log off, and which procedure names you want to use for this.

Logon/Logoff Notify tab

When you close the dialog box, AIMMS will create a skeleton for the selected notification procedures in the **Multi Agent Logon and Logoff** section of your model tree. If this section does not yet exist, it will be appended to the bottom of the tree. A notification procedure has only one argument, namely an element in the set `MultiAgent::AllAgents` that refers to the agent that has just logged on or logged off. The body of a notification procedure is initially empty.

The Multi Agent Logon and Logoff section

3.5 Sending and receiving messages

This section describes how the send, filter, and handler procedures fit in the process of sending messages from one agent to another. *How it works* Figure 3.5 gives an overview of this process.

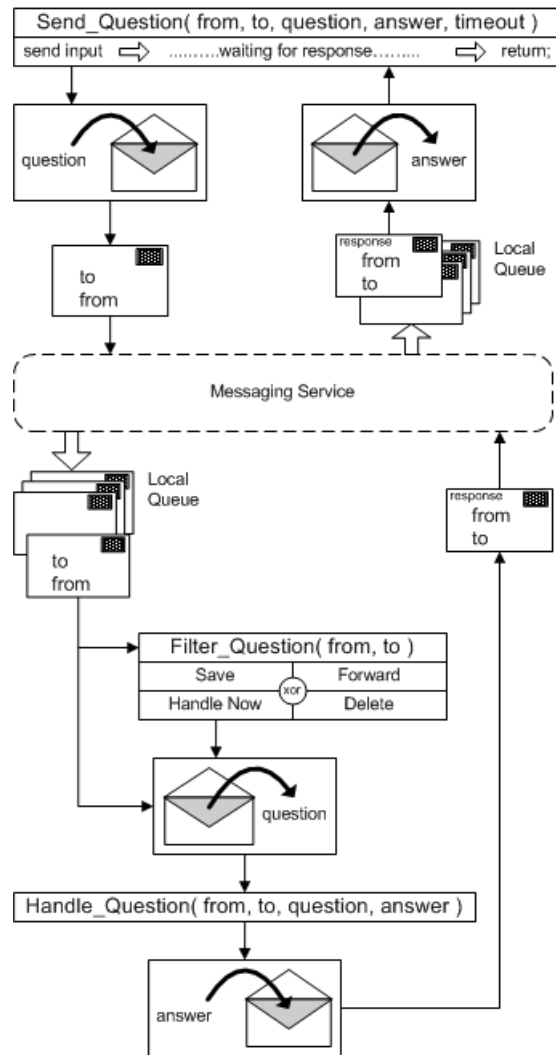


Figure 3.5: Sending and receiving messages

3.5.1 The send procedure

When you send a message from one agent to another, AIMMS places all the relevant information of that message in a special binary data package, and sends this data package to the queue of the destination agent.

Message package

You do not need to know the exact format of this package, but to understand how the messages are handled by the receiving application you can compare it to an ordinary envelope. On the outside of the envelope AIMMS writes the sending agent, the destination agent, and the message type. Inside the envelope, AIMMS puts all the data that relate to the arguments of the message.

... as an envelop

The last step of the send procedure is to close the envelope and send it to the queue (or the mailbox) of the destination agent. If the send procedure fails to locate this queue, it will return a value 0, and you can call the function `MultiAgent::GetLastErrorCode` to obtain the specific error code (see also Section 3.4.2).

Sending it

If the message only includes input arguments, and the send procedure returns without an error (i.e. a return value of 1), then you are sure that the message has been delivered to the queue of the destination agent. You have no guarantees if, or when, the message will be actually processed by that agent.

Only delivered

If the message includes output arguments, then the send procedures will **not** return immediately after the message has been delivered to the queue of the destination agent. Instead, the procedure will wait, for a specified amount of time, for the destination agent to send back a response message containing the values of the output arguments. Note that the waiting agent will only respond to priority messages when waiting. Thus, if the destination agent sends back a synchronous callback message to the waiting agent as part of handling the original request, such a callback message will timeout unless it is a priority message.

Wait for response

As soon as this response message arrives, the output arguments of the procedure are assigned the correct values and the procedure returns successfully. In the case of any failure, or if the maximum amount of wait time has elapsed, the procedure will return a value of 0 and you can call the function `MultiAgent::GetLastErrorCode` to obtain the specific error code (see also Section 3.4.2). If the procedure does not return successfully, the values of the output arguments are undefined.

Handling responses

3.5.2 The message queue

AIMMS automatically processes all messages that arrive in the queue of a multi-agent project as soon as possible. This means that you do not have to write any statements yourself to check whether new messages have arrived, or if there are still messages pending.

Automatic processing

If messages arrive at a faster rate than AIMMS and your model can process them, the messages are placed in the queue. As soon as AIMMS becomes idle, a pending message is taken from the queue using a first-in, first-out (FIFO) rule.

FIFO

Certain message types can be assigned a higher priority. If a priority message arrives at the queue, it is placed above any pending non-priority message. Thus, a priority message does not obey the FIFO rule, if non-priority message are pending.

... except priority messages

Besides the special order in which priority messages are processed, AIMMS also tries to handle a priority message even when it is not idle. During the handling of a previous message or during some other task, AIMMS checks, in between the execution of any two AIMMS statements, whether a priority message is pending. If so, it is processed immediately.

Priority processing

How a message is processed once it is taken from a queue, is discussed in the following two subsections.

Processing

3.5.3 Filtering messages

Using the analogy with the envelop, when AIMMS processes a message, it first looks at the outside of the envelope and reads the names of the sending agent, the destination agent, and the message type. If you have defined a filter procedure for the combination of message type and destination agent role, AIMMS calls this procedure with the two arguments that represent the sending agent and the destination agent completed.

How it works

Since the envelop remains closed, and AIMMS does not read the data inside, calling a filter procedure is faster then calling a handler procedure.

Filtering is fast

Furthermore, during the call to the filter procedure, the envelope remains in its original state, and this allows you to easily readdress and forward it to other agents. Only when the message is eventually sent to a handler procedure, will the envelope be opened, its contents read, and finally destroyed.

Message still exists

In a filter procedure you can write your own statements to determine what you want to do with the message that is currently being processed. Essentially, you can take three explicit actions:

Three actions

- readdress and forward the message to another agent,
- save the message so that you can handle it later, or
- delete the message

For each of these actions there is a specific procedure that you can call in the filter procedure.

Only when the filter procedure returns without any of these procedures being called, will the message be further processed and sent to the defined handler procedure.

... or default

By calling the function

Forwarding

- `MultiAgent::FilteredMessageForward(from_agent,to_agent)`

within a filter procedure, you send the current message to another agent in the community. The argument *to_agent* specifies the new destination agent. The argument *from_agent* can either be the agent that originally sent the message, or the agent that is forwarding it. In all cases, the new combination of *from_agent* and *to_agent* must be permitted by the message mapping defined in the **Multi Agent Community Setup** dialog box.

By calling the function

Saving

- `MultiAgent::FilteredMessageSave`

within a filter procedure, you save the message in a list of saved messages. The procedure returns an integer number that you can later use to access the saved message. The list of saved messages is discussed in Section 3.5.5.

By calling the function

Deleting

- `MultiAgent::FilteredMessageDelete`

within a filter procedure, you simply delete the entire message without further processing. If the message contains output arguments, and thus the sending agent is waiting for a response, the sending agent is notified that the message has been deleted and no output arguments are returned. The send procedure in the sending application will then return with an error.

3.5.4 Handling messages

If the message that is being processed passes the filter procedure, or if no filter procedure has been defined for the given message type, then the message is translated into a call to the handler procedure of the receiving agent. *Passed the filter*

Before calling the handler procedure, AIMMS opens the envelope, reads all the data that are inside, and puts these in the correct arguments of the handler procedure. *Opening the envelope*

If the message type contains output arguments, then you must make sure that, in the body of the handler procedure, you assign the requested values to the output arguments of the procedure. Once the handler procedure returns, AIMMS will read back the values of these output arguments and put them into a special response message that is immediately sent back to the sending agent. *Output arguments*

After the call to the handler procedure, and the possibly sending back of the output arguments, the processing of the current message is finished. If there are other messages pending, the next one will now be handled, otherwise AIMMS remains idle until a new message arrives. *Finished*

3.5.5 Saved messages

In a filter procedure you can choose to save a message so that you can process it later. These saved messages are all stored in a list. The multi-agent module offers several procedures to browse through this list and take specific actions on each message in it. *List of saved messages*

At any time you can call the function *Counting*

■ `MultiAgent::SavedMessageCount`

to obtain the number of messages that are currently in the list of saved messages.

Each message in the list has a unique identification number, through which you can access it. This number is returned when you save the message using the procedure `MultiAgent::FilteredMessageSave`, but if you do not want to store these numbers, you can also search the current list for messages with specific characteristics. *Identification number*

The procedure

- `MultiAgent::SavedMessageFind([from_agent],[msg_type])`

searches the list of saved messages for a message from a specific agent and/or of a specific message type. If you specify the argument *from_agent*, then it searches for the first message that was sent by that given agent. If you specify the argument *msg_type*, then it searches for the first message of that type. If you specify both arguments, then the first message that matches both criteria is returned. Omitting both arguments simply returns the identification number of the first message in the list. The integer return value of the procedure is the identification of the message found. If no message matches the criteria, or the list is empty, the procedure returns 0.

Finding saved messages

With the procedure

- `MultiAgent::SavedMessageGetInfo(nr,from_agent,msg_type)`

you can obtain the sending agent and the message type of a saved message. The input argument *nr* is the identification number of the saved message. If this number does not exist in the list, the procedure will return 0. Otherwise, the function returns 1 and the arguments *from_agent* and *msg_type* contain the requested information.

Getting info

The usual reason for saving a message is that you want to process it later. Probably because you did not have the resources, or the time, when the message arrived. The options for processing a saved message are similar to the filter actions:

- handle the message now,
- readdress and forward the message to another agent, or
- delete the message

For all three options there are special procedures in the multi-agent module, and each of these procedures has a first argument *nr*, the identification number of the saved message that you want to process. After calling any of these procedures the specified saved message is removed from the list.

Processing saved messages

The procedure

- `MultiAgent::SavedMessageHandleNow(nr)`

takes the saved message from the list and passes it to the handler procedure of the destination agent. From then on, the message is treated as if it has just passed the filter procedure.

Handle now

If you call the procedure

Forward

- `MultiAgent::SavedMessageForward(nr,from_agent,to_agent)`

the indicated saved message is taken from the list and forwarded to another agent in the community. The argument *to_agent* gives the new destination agent. The argument *from_agent* can either be the agent that originally sent the message, or the agent that is forwarding it. In all cases, the new combination of *from_agent* and *to_agent* must be permitted by the message mapping that is defined in the **Multi Agent Community Setup** dialog box.

If you call the procedure

Delete

- `MultiAgent::SavedMessageDelete(nr)`

the indicated saved message is removed from the list of saved messages and is deleted.

3.5.6 Synchronization

In a completely asynchronous agent community, agents are not dependent on the amount of time other agents need to perform some tasks. Each agent in an asynchronous community remains idle until a message from another agent arrives, it then handles that message and perhaps, as a result, sends messages to other agents.

Asynchronous

Usually, a community needs some sort of synchronization. For example, if a community works through several phases, it may be important to wait until all agents have finished the previous phase before moving on. Another type of synchronization occurs if an agent cannot continue without certain values arriving from other agents.

Synchronization

The multi-agent technology of AIMMS offers three ways to synchronize your community:

Three options

- using message types with output arguments,
- using special wait functions, and
- scheduling procedure calls.

Sending a message with output arguments implies some sort of synchronization. Namely, the send procedure for that message will block and wait for a specified number of seconds or until the destination agent has processed the message and returns the required output values. Any send procedure that includes output arguments has an optional last argument *TimeOutPeriod*, which indicates the maximum number of seconds to wait for the response message. By default it is set at 10 seconds.

Output arguments

The function

- `MultiAgent::WaitForMessage([from_agent][,msgType][,timeout])`

pauses the current execution sequence until a specific message that you are expecting to receive, arrived and is handled. The input argument *from_agent* specifies the agent from which you expect the message. If you leave this empty, it may come from any agent. The input argument *msgType* specifies the type of message that you expect, or similarly any type of message if you leave it empty. The last argument, *timeout*, gives the maximum number of seconds to wait. The default time-out period is 10 seconds. If the time-out period elapses, without receiving the expected message, then the procedure returns 0. If the message does arrive as expected, it is handled, and subsequently the `WaitForMessage` call returns with a value 1.

Wait for one message

The function

- `MultiAgent::WaitForMultipleMessages(nr,agnts,msgTypes[,tmout])`

halts the current execution sequence until a given number of messages, that you expect to receive, arrive and are handled. The first argument *nr* indicates how many messages you expect. The argument *agents* is a subset of agents from which you expect the messages. If you leave this empty, the messages may come from any agent. The input argument *msgTypes* specifies the subset of message types that you expect. Again, if you leave this empty, any type of message is acceptable. The last argument, *timeout*, gives the maximum number of seconds to wait. The default time-out period is 10 seconds. If the time-out period elapses, without receiving the expected number of messages, then the procedure returns the number of messages that were handled. If all the expected messages do arrive in time, then they are handled, and subsequently the `WaitForMultipleMessages` call returns with a value that is equal to the input argument *nr*.

Wait for multiple messages

During the time that either the procedure `WaitForMessage` or `WaitForMultipleMessages` is waiting, AIMMS only processes messages that match the given wait criteria, and priority messages. This means that during this wait time the FIFO rule for processing messages in the queue does not apply.

What about FIFO?

If you send a message without any output arguments, you can still wait until the receiving agent has actually handled the message. To achieve this you must mark the message with the property **Sends response when handled**. Messages with this property ensure that the sending agent gets a response message from the receiving agent as soon as the handler procedure is finished. Using the two functions

Wait until handled

- `MultiAgent::WaitForHandledMessage([from_agent][,msgType][,timeout])`
- `MultiAgent::WaitForMultipleHandledMessages(nr,agnts,msgTypes[,tmout])`

you can block the current execution until the expected number of response messages arrive. The arguments and behavior of both functions are similar to those of `WaitForMessage` and `WaitForMultipleMessages`, respectively.

When using the wait functions described above, you have to make sure that you do not inadvertently create an opportunity for waited-for messages to get handled prior to actually calling the wait function, as this may cause the wait function to timeout. This can happen whenever you wait for priority messages, because priority messages can be handled in between any two AIMMS statements. In addition, (response) messages can get handled if the AIMMS session enters an idle state anywhere between the moment you start to expect incoming (response) messages and the actual call to any of the wait functions.

*Waiting for
messages
handled already*

A drawback of the wait functions is that they do not put the AIMMS program into an idle state. In other words, during the wait period the program is still busy. This means that any other request to the program, either by user input or by another agent sending a message, is not immediately handled.

Not idle

Another mechanism for obtaining some sort of synchronization is to run a procedure at a specified moment in time. Within such a procedure you can check whether certain messages have arrived.

*Scheduling a
procedure*

You can use the interface function `ScheduleAt` to schedule a procedure in your model to be run at a certain moment in time. The given procedure must be without arguments.

*The ScheduleAt
function*

Another way to schedule a procedure to be run at a certain moment, is by sending a delayed message to an agent. Any send procedure that is created for a message type with only input arguments has a last optional argument *SendDelay*. By default, this delay is set to 0, but if you give it a value greater than 0, AIMMS will hold the message for the given number of seconds before sending it. Since it is permissible to send a message to yourself, you can use this to schedule a procedure *with* arguments (namely the message handler) to be run at a later time.

Send with delay

3.6 Starting the multi-agent community

So far, we have mainly discussed the mechanisms for sending and receiving messages between agents in a community where this community already exists and all agents are already logged on. This section describes how the community is created and how the agents get logged on.

*Community
instantiation*

In AIMMS' multi-agent technology, all individual agents communicate through the project/AIMMS session in which they reside. When you send a message to a particular agent, AIMMS works out in which project it resides, and sends the message to that project.

Network of projects

Each project has a unique address within the physical computer network. This address consists of two parts:

Project address

- the name of the host computer on which it runs, and
- a so-called queue name, which must be unique to the host computer.

To send a message to another project, AIMMS thus needs to know this unique address of the project.

At the startup of an AIMMS multi-agent project, the project must obtain the address of at least one other project that is already part of the community. If it connects successfully to that project, the addresses of all other agent projects are passed back and forth so that, afterwards, your agent can address any other agent in the community.

How projects connect

In many multi-agent communities there is one key controlling or central agent. Usually, this agent is started first, since without this agent it makes no sense running the multi-agent application. The project in which this agent resides is then also the project to which all other agent projects can connect, and hence only the address of this project needs to be made available to all the others.

Most common setup

3.6.1 Agent project startup

Every AIMMS agent application must first create a local queue before it can use any of the other procedures from the multi-agent module. The procedure that creates the local queue is

Create local queue

- `MultiAgent::CreateLocalQueue([queuename])`

The argument *queuename* is optional. You only need to specify it, when other projects will connect to your queue. In Figure 3.6 you will see a configuration of agent projects P1 through P5, where only project P1 and project P2 have chosen an explicit queue name. The other projects do not need an explicit name for their queues since they connect to either P1 or P2.

The local queue name

If you explicitly specify the argument *queuename*, you must ensure that this name is a unique queue name for the computer on which the project runs. If you do not specify it, AIMMS will automatically generate a unique queue name.

Unique queue name

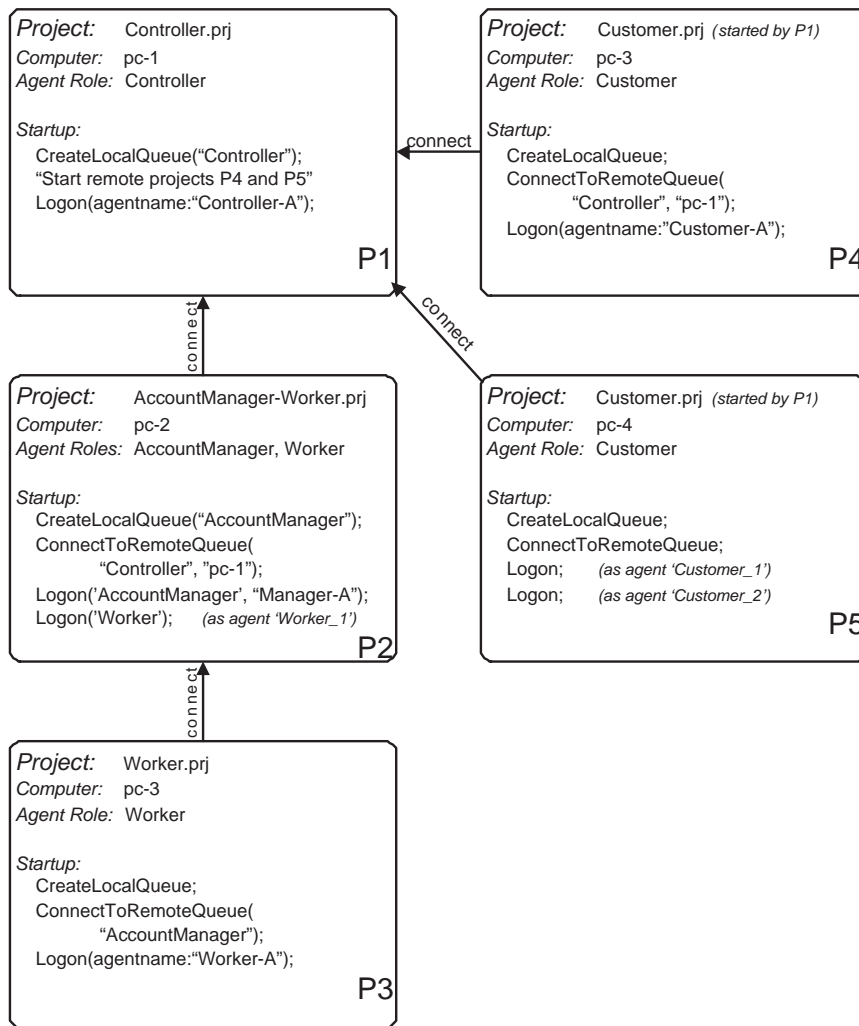


Figure 3.6: Starting up several AIMMS agent projects

If the procedure `CreateLocalQueue` is successful, it returns the value 1. On failure, it returns 0, and you can call the function `MultiAgent::GetLastErrorCode` to obtain the specific error code.

Return value

After your local queue has been created, you can connect to the queue of another project that already exists in the community. Naturally, the very first project that is started does not have to connect to anything. To connect to another project you call the procedure

Then connect

- `MultiAgent::ConnectToRemoteQueue(queueName,hostname)`

The string argument *queuename* is the name of the queue of the project to which you want to connect. The string argument *hostname* is the name of the computer on which that project runs. You may omit the argument *hostname* if the project to which you connect is running on the same computer as your own project.

Remote queue name

In the situations described above, it is assumed that each project is started individually (and manually). However, the AIMMS agent technology also has the facility to start remote agent projects from within your project (see also Section 3.6.2). A project that is started remotely should, during startup, create its local queue and connect. However, in the connect call, you may omit *both queuename* and *hostname* arguments. The project will then automatically try to connect to the queue of the project from which it was started.

Started remotely

In Figure 3.6, project P1 was the first project of the community and does not have a connect statement. Project P2 connects to the queue of project P1, while project P3 connects to the queue of project P2. Projects P2 and P3 run on the same computer and thus the connect call does not need a *hostname* argument. Both projects P4 and P5 are started remotely by project P1, project P4 connects explicitly to the queue of project P1 while project P5 uses a connect call without arguments and thus relies on an automatic connection to the queue of the project from which it was started.

Examples

3.6.2 Starting remote projects

The multi-agent module of AIMMS also incorporates a mechanism to start other projects from within you own AIMMS project. In order to do this you must know the exact location of the project file (with a .prj extension) and the name of the computer on which the program should run.

From within AIMMS

The remote project is started using the AIMMS COM interface. This requires AIMMS to be installed properly (i.e. installed using the AIMMS installation executable) on the computer that is going to run the remote project.

Uses AIMMS COM

For each remote project that you want to start you must add an element to the set `MultiAgent::AllRemoteProjects`. You are free to pick any name for these elements.

Set of remote projects

For each element in the set `MultiAgent::AllRemoteProjects` you must supply a full path to the project file that you want to open, and the name of the computer on which the project will run. If this host computer is not the same as the computer on which your project runs, you must supply the path in such a way that the host computer can find it. The path should be provided in the string

Project paths and hosts

parameter `MultiAgent::RemoteProjectPath(arp)`, and the host name should be provided in the string parameter `MultiAgent::RemoteProjectHost(arp)` (where `arp` is an index in the set `MultiAgent::AllRemoteProjects`).

For the example presented in Figure 3.6, these identifiers could be initialized as follows: *Example*

```
MultiAgent::AllRemoteProjects := data { P4, P5 };

MultiAgent::RemoteProjectPath := data {
  P4 : "c:\\projects\\P4\\Customer.prj",
  P5 : "c:\\projects\\P5\\Customer.prj"
};

MultiAgent::RemoteProjectHost := data {
  P4 : "pc-3",
  P5 : "pc-4"
};
```

If the remote project is to be run on the same computer as your current project, then you may leave the corresponding element in `RemoteProjectHost` empty, or set it to the empty string. *Hostname is optional*

Once you have completed the required path and host you can call the procedure *Starting a remote project*

- `MultiAgent::StartRemoteProject(remote_project)`

where *remote_project* is an element parameter in the set `MultiAgent::AllRemoteProjects`.

The procedure `StartRemoteProject` uses the AIMMS COM interface to start a new AIMMS session on the indicated host computer and open the given project file. The project is opened in end-user mode, and will execute its predefined startup sequence. During this startup it will ensure that it connects itself to the community as described in the previous section, and that it logs on its agents. *What it does*

All remotely started projects remain in control of the project that started them, and this controlling project should thus also take care that the remote projects are properly closed if they are no longer needed. A remote project that you have started previously, is closed by a call to the procedure *Closing a remote project*

- `MultiAgent::StopRemoteProject(remote_project[,interrupt])`

With the optional argument *interrupt* you can indicate whether you want to interrupt the current execution of the project to be stopped.

If you do not explicitly close the remote projects yourself, AIMMS closes them automatically when the project that started these remote projects is closed. *Automatic closing*

3.6.3 Agent log on

Once your project is part of the community, you can log on one or more agents with agent roles that you have implemented in your project. To log on an agent, you call the procedure *Procedure Logon*

- `MultiAgent::Logon([role],[agentname])`

If your project only has one agent role, you may omit the argument *role*. In this case it automatically defaults to that agent role. If your project involves multiple roles, then you must indicate which role the newly logged on agent has. *Optional role*

You may also omit the argument *agentname*. In this case AIMMS will generate a name that is unique within the current community of agents. This generated name is derived from the role name. For example if the role is 'Worker' the generated agent names will be 'Worker_1', 'Worker_2', etc. *Optional agentname*

If you explicitly specify the agent name, then you must make sure that this name is unique within the current community of agents. If it is not unique, the log on will fail and `MultiAgent::Logon` returns 0. *Unique agent name*

In the example shown in Figure 3.6 the five projects eventually log on a total of 7 agents. Only project P2 needs to supply the agent role explicitly in the Logon statement, since that project implements two different agent roles. Assuming that the projects are started in the order P1-P2-P3-P4-P5, the following agents will eventually be logged on: *Example*

- "Controller-A" which is a 'Controller' located in project P1,
- "AccountManager-A" which is an 'AccountManager' located in project P2,
- "Worker_1" which is a 'Worker' located in project P2,
- "Worker-A" which is a 'Worker' located in project P3,
- "Customer-A" which is a 'Customer' located in project P4,
- "Customer_1" which is a 'Customer' located in project P5, and
- "Customer_2" which is a 'Customer' located in project P5

If your agent has logged on successfully, the name of this new agent is automatically added as an element to the set `MultiAgent::AllAgents` in all the projects that are currently part of the community. Additionally, the element parameter `MultiAgent::AgentRole`, which is indexed over all agents, is updated accordingly, so that you can determine the role of any agent. *AllAgents*

After a successful call to `MultiAgent::Logon` in your project, the element parameter `MultiAgent::ThisAgent` refers to the most recently logged on agent. As long as you only log on one agent in your project, you can simply use this parameter as the 'from' parameter when sending messages. In addition, the set `MultiAgent::AllLocalAgents` which is a subset of `MultiAgent::AllAgents`, contains all the agents logged on in your project.

ThisAgent

If you want to log on multiple agents in your project, then you must make sequential calls to the procedure `MultiAgent::Logon`. If you need references to each logged on agent, then you may declare an element parameter for each of them, and assign the value of `MultiAgent::ThisAgent` to it following each call to `MultiAgent::Logon`.

Multiple log ons

When you close your project all local agents will be logged off automatically. To log off a local agent without closing the project, you call the procedure

*Procedure
Logoff*

- `MultiAgent::Logoff(agent)`

The optional *agent* argument should refer to one of the agents in the set `MultiAgent::AllLocalAgents`. In case the project contains one local agent you may omit the *agent* argument and the only local agent will be logged off.

Optional agent