

# AgentService

## *Two interacting agents...*

*An informal guide*

*The AgentService Team*

*Copyright @ 2007 – l.i.d.o. – DIST University of Genoa*

## **Abstract**

A very very short guide for a very very stupid multi-agent application with the main goal to show in a simple way how to create an AgentService multi-agent platform.

# Index of Contents

Abstract .....	2
Index of Contents .....	3
1. Introduction .....	4
2. Knowledge objects .....	4
3. Behaviours .....	5
3.1. Conversation .....	5
3.2. Concurrent access to a knowledge object .....	5
3.3. How to exchange messages .....	6
3.4. The rest of the behaviour .....	6
4. Agents .....	7
4.1. Handling a conversation .....	7
5. Batch .....	7
6. Platform execution .....	8
7. Notes .....	8

# 1. Introduction

Dear user,

I write for you this simple (and stupid) multi-agent application, in order to show how to implement an agent society by using *AgentService*. This application executes two agents (*Goofy* and *Pluto*) who talk by using a conversation (and *serialized* message contents).

- Goofy generates a random number and then send it to Pluto.
- Pluto receives the message,
  - gets the number
  - and uses it as seed for another random number,
  - then sends the new number to Goofy.
- Goofy receives the new message,
  - stores the number in his knowledge object
  - reads the number from the knowledge object (it is only a way to show you how to use *knowledge objects*).
  - generates another random number
  - and sends it to Pluto.
- ...and so on...

Now it's time to implement the application. Create a *class library solution* in *Visual Studio .NET* or *Sharp Develop* and import two assemblies containing the main structures of the platform: *AgentService.dll* and *AgentServiceInterface.dll* (you can find them in the *BUILD* folder in the *AgentService* solution, if you have already compiled the entire solution).

## 2. Knowledge objects

First, I suggest you to define your knowledge object for the *AgentOne* (the agent that will be instantiated with the name *Goofy*). See the *KnowledgeObject.cs* file in order to read my comments to the statements. Don't forget to add the *using* directives, to add the *[Serializable]* tag and the *Knowledge* parent class.

Generally, you can create several *knowledges* and reuse their definition in different agent behaviours, but remember that every agent has its instance of a knowledge object.

Inside a knowledge object you can put every type you want (but it must be *Serializable*). In this application we create a simple knowledge with only a

field: the integer *value*, but if you want, you can also add a *float*, a *string*, a *Dog* type etc.

Pay attention to the complex syntax for the *get* and *set* methods (obviously if you add other data field, you must add the respective *getters* and *setters*).

For the moment you can ignore the *pMode* parameter.

## 3. Behaviours

Now it is the time to implement the behaviours: in this application, one for each agent. Open the file *BehaviourForAgentOne.cs*. Include the *using* directives and create the *BehaviourForAgentOne* (remember that the *AgentOne* is *Goofy*) which inherits from *Behaviour*. Add the *[Knowledge]* tag with the supported types of knowledge. In this case *BehaviourForAgentOne* uses the *KnowledgeObject* knowledge type.

Implement the two *constructors* and then implement the *Body* method. Here you must insert all the *business logic* of the behaviour (you can also add private worker methods...).

### 3.1. Conversation

First, you must open a conversation with *Pluto*. You have to instantiate an *AID* (Agent Identifier) passing the *name* (*Pluto*) and the *HapName* (host application name: namely the platform name: for a default installation of *AgentService* it is... “*AgentService*” but you can change it in the *install.xml* file) of the interlocutor agent. A simple way to determine the application name is to use *this.Owner.HapName*.

Please note the use of a *while* loop in order to wait for the acknowledge of the peer agent (*Pluto*).

### 3.2. Concurrent access to a knowledge object

Then we enter in the *for* loop. Please pay attention to the procedure which manages the concurrent access to a knowledge object. Remember that a knowledge object is in the owner agent scope and every behaviour belonging to the agent could access it. So you must lock the knowledge you want (in fact you can have different knowledges), access the given value and then release the knowledge.

### 3.3. How to exchange messages

After the random number generation, you see how you can send a message. In this case you have to define only a *message body item* where the agent can save the number to send. You can define different *body items* for one message body, in order to send a set of objects (not only *strings* or *integers*, but also *user defined serializable types*).

Incidentally, in AgentService you can send a message to another agent by using *conversations* (as in this case) or by sending *single messages* (without opening a conversation). If you send single messages you must create the entire *AgentMessage* object (both *envelope* and *message body*). In case of conversations you have to define only the message body.

To receive a message you can use *WaitForMessage* that is a blocking method which waits until a new message arrives in the agent message queue. If you don't want to use the blocking version you can use *ReceiveMessage*.

### 3.4. The rest of the behaviour

At the end of the loop you see how to update a knowledge: you must lock the knowledge, set the new value and update the knowledge.

To display strings on the console we use *this.Runtime.Console.AppendMessage*, but you can also use the usual *System.Console.WriteLine*.

The behaviour activity is now finished, so the agent can close the conversation and then stop his activities.

The life of an agent terminates when no behaviours are running. In this case the agent ends when the unique behaviour stops.

We now concentrate on the *BehaviourForAgentTwo*. Note that this behaviour doesn't use knowledge objects.

The behaviour starts when the *AgentTwo* is instantiated (see section 4) and then it waits for an incoming conversation by using the while loop. When there is an incoming conversation you must get it from the list of incoming conversations (see paragraph 4.1).

Now you must define the interaction with *Goofy* (it is similar to the *Goofy's* activity).

## 4. Agents

Now you must define the two agents. Let's start from the *AgentOne*. The *AgentOne* class must be tagged with *[Serializable]*, the list of supported behaviours *[AgentBehaviours]*, and the list of knowledges *[AgentKnowledges]*. Be sure to derive this class from *AgentTemplate*.

Sets the default constructor and then concentrate on the *Initiate* method. As you see, it contains the statements which create *behaviours* and *knowledges*.

### 4.1. Handling a conversation

Regarding the *AgentTwo*, there are only two differences: *AgentTwo* doesn't support knowledges and must manage incoming conversations through *HandleConversation*. When a conversation is incoming, this method is called. In the body of the method you can accept or refuse the conversation. In case of an accepted conversation, you must now link the conversation to a behaviour, Remember that *BehaviourForAgentTwo* starts inside the *Initiate* method (and then is executed in its own thread) and it waits for an incoming conversation through that *while* loop.

You can create a behaviour (or a knowledge) also in the *HandleConversation* method; if you see the methods of *sender* you will see *CreateBehaviour* and *CreateKnowledge* (when you create multi instances of a behaviour remember to give unique behaviour names). Please note that if you create no behaviours in the *Initiate* method, the agent starts and immediately ends.

## 5. Batch

In order to instantiate agents you can run the platform in *batch mode*. Essentially you can always copy the statements you see in the example. The required changes are circumscribed in *line 22* (you must enter the name of the assembly containing your agents) and at *line 28* where you create instances of agents (in general, you can create multiple instances of the same agent). You must enter the *full type name (namespace + class name)*, the name of the agent, the path of the configuration file (skip this detail, but take into account that you can configure your agents with your user defined configuration files), and set his persistence mode (skip this detail too).

## 6. Platform execution

Finally the platform is ready to use. The last steps are the *abs* file and the execution.

To implement an *abs* file is very simple:

```
file=configuration.conf
conf=conf\platform.conf
type=DummyAgents.Batch
assembly=DummyAgents.dll
```

Set the *full type name* of the *Batch* class and set the assembly containing your agents.

And now, go to the folder containing your AgentService installation, copy the *abs* file (with whatever name) and the *DummyAgents.dll* in this folder.

Type `asdrv -x:<file_name>.abs` and you will see (I hope) the two interacting agents.

## 7. Notes

If you want to debug your multi-agent application, type `asdrv -d -x:<file_name>.abs` so you will be able to attach to the process and then debug the application.

The services offered by the platform to its agents are contained in the runtime object (*this.Runtime*) accessible in your behaviour classes:

- Logging service
- Message service
- Console
- Mobility (work in progress)
- Persistence
- Yellow pages service (Directory facilitator)
- White pages service.

Other interesting features of AgentService are the agent and behaviour thread management, the ontology service, the inter-platform communication infrastructure.

If you are interested in one of these features, we will expand the *Goofy* and *Pluto* application in order to illustrate a concrete application.