SPADE User's Manual

For SPADE 2.0

Gustavo Aranda
Javier Palanca
Natalia Criado

SPADE User's Manual: For SPADE 2.0

by Gustavo Aranda, Javier Palanca, and Natalia Criado

Published (TBA)

Table of Contents

Foreword

Gustavo Aranda
Valencia, October 31, 2007

The idea of a Jabber-based agent platform appeared one night at 4 A.M. when, studying the features of the Jabber architecture, we found out great similarities with the ones of a FIPA-compliant agent platform. The Jabber protocol offered a great architecture for agents to communicate in a structured way and solved many issues present when designing a platform, such as authenticating the users (the agents), provide a directory or create communication channels.

We started to work on our first prototype of this Jabber-powered platform and within a week we had a small working proof of concept by the name of *Fipper* which eventually allowed for dumb agents to connect and communicate through a common jabber server.

Since that day, things have changed a bit. The small proof of concept evolved into a full-featured FIPA platform, and the new SPADE name was coined. As usual, we later had to find the meaning of the beautiful acronym. We came up with **S**mart **P**ython multi-**A**gent **D**evelopment **E**nvironment, which sounded both good and geek enough.

We hope you like this book and have as much fun using it as we had writing it. Of course we also hope that it may become useful, but that is a secondary matter.

Chapter 1. Basic Concepts

This chapter is a short, casual introduction to Spade. If you're new to multi-agent systems, this chapter is definitely for you. We begin with a discussion of general multi-agent systems concepts, instant messaging and Jabber, and work our way into the specific ideas behind SPADE.

Multi-Agent Systems

A multi-agent system (MAS) is a system composed of several software agents, collectively capable of reaching goals that are difficult to achieve by an individual agent or monolithic system.

The exact nature of the agents is a matter of some controversy. They are sometimes claimed to be autonomous. For example a household floor cleaning robot can be autonomous in that it is dependent on a human operator only to start it up. On the other hand, in practice, some agents are under active human supervision, becoming interdependent systems.

MAS systems are also referred to as "self-organized systems" as they tend to find the best solution for their problems "without intervention". There is high similarity here to physical phenomena, such as energy minimizing, where physical objects tend to reach the lowest energy possible, within the physical constrained world.

The main feature which is achieved when developing MAS systems is flexibility, since a MAS system can be added to, modified and reconstructed, without the need for detailed rewriting of the application. These systems also tend to be rapidly self-recovering and failure proof, usually due to the heavy redundancy of components and the self managed features, referred to, above.

Kickin'it FIPA style

In order to prevent the proliferation of incompatible agent systems and to promote the agent-based technology and the interoperability of its standards with other technologies, the Foundation for Intelligent Physical Agents (FIPA) was formed. It is an IEEE Computer Society standards organization.

The FIPA has produced a set of documents (called "FIPA Specifications") that together conform the modern standard (or HOW-TO) in terms of agent technology. It starts with the definition of the agent platform, the "home" of the agents. It is the environment where the agents are placed and where their activities are performed. The software foundation for a MAS-based solution and the *network server* they connect to.

The FIPA defines a model for an agent platform and a communication language for the agents. All the platforms that want to be FIPA-compliant should follow this model and understand the language, and SPADE is no exception. The compliance requires basically four features:

- An Agent Communication Channel (ACC). A mechanism which allows the agents (and the platform itself) to communicate with one another.
- An Agent Management System (AMS). A way for the agents to be registered in the platform and to be reachable for contact (kind of a *White Pages* service).
- A Directory Facilitator (DF), which is a kind of public service in which agents publish the services they offer, something akin of a *Yellow Pages* service.

- Support for the FIPA Agent Communication Language (ACL), which is a common language for all agents to communicate with. Essentially, it is presented in a couple of flavors: one based on a Lisp-like syntax environment (using an obscene amount of parenthesis), and a clear, pure, practical and beautiful syntax based on XML.

SPADE implements all these features (and a few others) and thus is a FIPA-compliant platform.

Instant Messaging

Instant messaging (IM) is a form of real-time communication between two or more people based on typed text. The text is conveyed via computers connected over a network such as the Internet.

Instant messaging offers real-time communication and allows easy collaboration, which might be considered more akin to genuine conversation than email's "letter" format. In contrast to e-mail, the parties know whether the peer is available. Most systems allow the user to set an online status or away message so peers are notified when the user is available, busy, or away from the computer. This is called the *presence* of the user. On the other hand, people are not forced to reply immediately to incoming messages. For this reason, some people consider communication via instant messaging to be less intrusive than communication via phone. However, some systems allow the sending of messages to people not currently logged on (offline messages), thus removing much of the difference between IM and email.

Instant messaging allows instantaneous communication between a number of parties simultaneously, by transmitting information quickly and efficiently, featuring immediate receipt of acknowledgment or reply. In certain cases IM involves additional features, which make it even more popular, i.e. to see the other party, e.g. by using web-cams, or to talk directly for free over the internet.

XMPP and Jabber

Extensible Messaging and Presence Protocol (XMPP) is an open, XML-inspired protocol for near-real-time, extensible instant messaging (IM) and presence information (a.k.a. buddy lists). It is the core protocol of the Jabber Instant Messaging and Presence technology. The protocol is built to be extensible and other features such as Voice over IP and file transfer signaling have been added.

Unlike most instant messaging protocols, XMPP is based on open standards. Like e-mail, it is an open system where anyone who has a domain name and a suitable Internet connection can run their own Jabber server and talk to users on other servers. The standard server implementations and many clients are also free and open source software.

The Internet Engineering Task Force (IETF) formed an XMPP Working Group in 2002 to formalize the core protocols as an IETF Instant Messaging and presence technology. The four specifications produced by the XMPP WG were approved by the IESG as Proposed Standards in 2004. RFC 3920 and RFC 3921 are currently undergoing revisions in preparation for advancing them to Draft Standard within the Internet Standards Process. The XMPP Standards Foundation (formerly the Jabber Software Foundation) is active in developing open XMPP extensions.

XMPP-based software is deployed on thousands of servers across the Internet and by 2003 was used by over ten million people worldwide, according to the XMPP Standards Foundation.[1] Popular commercial servers include the Gizmo Project [http://www.gizmoproject.com/]. Popular client applications include the freeware clients offered by Google and the Gizmo

Project, multi-protocol instant messengers such as iChat [http://www.apple.com/ichat/] and Pidgin [http://www.pidgin.im/] (formerly Gaim), and free dedicated clients such as Psi [http://psi-im.org]. Google Talk [http://www.google.com/talk/] provides XMPP gateways to its service.

XMPP and Jabber Features

- **Open, public, and free.** The Jabber protocol is open, public, free and well-documented, as opposed to other popular IM systems. The XMPP technology is well-documented and public by means of several RFC standard documents developed by the Jabber Software Foundation. Any individual or organization can make use of XMPP technologies without having to pay royalties, licenses, etc ...

- **Asynchronous.** A Jabber server guaranties delivery of messages while the destination client is online. When a client goes offline, the Jabber server stores and forwards the message to a client when the client reconnects. This feature, apart from being useful in the case of messages between humans, helps with the fact that Jabber can be used as an architecture that enables systems to talk with other systems. Consider a scenario with a client that interacts via Jabber with a command control, sends commands, and receives responses. In this scenario, even thought the client goes offline, the system guarantees that the response from the command control is stored and sent to the client when the client goes online again.

- **Decentralized.** By definition, Jabber is decentralized. Therefore, a Jabber central server that provides the service to everybody does not exist. Jabber is a servers network that is independent but simultaneously interconnected. As a result, anyone can run their own Jabber server, creating networks and sub-networks that do not depend on a central backbone and add it as a component of the agent architecture. A user can choose a server that is trusted by her. She is not forced to use a server that generates suspicion. She may use a server provided by a commercial company of her trust (like her bank or univertity), the one of her employer, etc ... A user can even choose to set up her own Jabber server at home.

- **Secure.** As with all other network services (e.g., HTTP, SMTP, and FTP), security is an important concern with regard to the deployment of IM services based on Jabber technology. One aspect of these services is the ability to enable connections between Jabber servers, thus taking advantage of the "network effect" of existing Jabber servers on the Internet or on a company network.As specified in RFC 3920, the core transport layer for XMPP is an XML streaming protocol that makes it possible to exchange fragments of XML between any two network endpoints. Authentication and channel encryption happen at the XML streaming layer using the IETF-standard protocols for Simple Authentication and Security Layer (SASL) and Transport Layer Security (TLS). The XMPP architecture is a pure client-server model, wherein clients connect to servers and (optionally) servers connect to each other for inter-domain communications.

- **Extensible.** Using the power of XML namespaces, anyone can build custom functionality on top of the core protocol, as exemplified by the Jabber Enhancement Proposals (JEPs) coordinated by the Jabber Software Foundation. Jabber was designed from the beginning to be extended. A good example of this are the changes and improvements that have gradually appeared in the Jabber technology without breaking backward compatibilities. To mention some: Support of SSL and GPG encryption; precise information on the state of a message ('is being written', 'the contact has received it' or 'it has been kept in the server'); multi-conference chat rooms (MUC) that allow multiple users to interact simultaneously using virtual channels; a cooperative online 'blackboard' to draw graphics and share multimedia contents among the participants of a conversation (The Coccinella [http://hem.fyristorg.com/matben/]). These extended functionalities do not require the Jabber servers to be modified at all. They rely on the Jabber clients to implement the necessary features to support them. Thus, in Jabber the power of extending functionalities is in the client.

- **Flexible.** The most common misconception about Jabber is that it is solely for IM. Jabber is more than a bridge to other IM systems or a set of protocols. IM is just one of the countless ways that an XML-based messaging technology such as Jabber's can be applied. Jabber is a flexible and extensible client/server architecture that routes XML data between clients and services, which plug into Jabber servers as components. The original and core set of components provide the IM features (and support services) that have been discussed. The XML structures that make up the Jabber protocol are split into top-level tags (also referred to as elements). These tags do not necessarily carry human-generated IM message content; as long as the resulting XML is well-formed, anything goes. Furthermore, the Jabber protocol design makes use of an XML feature that allows total flexibility for extensions, called namespaces. Bearing this in mind, it is clear that Jabber can be deployed to provide solutions far beyond the IM space as well as within it. Jabber applications beyond IM include network management, content syndication, collaboration tools, file sharing, gaming, remote systems monitoring and, now, agent communication.

The SPADE Agent Platform

Simply put, SPADE is an agent platform based on the XMPP/Jabber technology. This technology offers by itself many features and facilities that ease the construction of MAS, such as an existing communication channel, the concepts of users (agents) and servers (platforms) and an extensible communication protocol based on XML, just like FIPA-ACL. Many other agent platforms exist, but SPADE is the first to base its roots on the XMPP technology.

The SPADE Agent Platform does not require (but strongly recommends) the operation of agents made with the SPADE Agent Library (see next section). The platform itself uses the library to empower its internals, but aside from that, you can develop your own agents in the programming language of your choice and use them with SPADE. The only requirement those agents must fulfill is to be able to communicate through the XMPP protocol. The FIPA-ACL messages will be embedded in XMPP messages. Be warned, however, that some features of the whole SPADE experience may not be available if you do not use the SPADE Agent Library to build your agents.

SPADE is written in the Python programming language. In order to fully understand and use SPADE, a bit of knowledge about Python is required, but, hey! you would not be reading this manual if you weren't a top-notch programmer, wouldn't you?

The SPADE Agent Library

The SPADE Agent Library is a module for the Python programming language for building SPADE agents. It is a collection of classes, functions and tools for creating new SPADE agents that can work with the SPADE Agent Platform. Using it is the best way to start developing your own SPADE agents. In the future, we would like to offer bindings of the SPADE Agent Library for more programming languages, like Java or C#, but for now only Python is supported.

If you have worked with Python before, you will find the SPADE Agent Library easy to understand and use, with classes ready to use and expand, and many functionalities already built in the library. If you have worked with other high-level behavioral agent platforms (like Jade or MadKit) you will find some similarities in the way the library works, although the agent model present in SPADE is a bit different.

What's Next?

If you want to get your hands dirty and start using SPADE right now, head on to the next chapter where a Quickstart Guide awaits you. However if you wish to learn more about SPADE and see how deep the rabbit hole goes, you can go directly to chapter XX.

Chapter 2. Quickstart Guide

This chapter is a short, quickstart guide for those brave enough to try SPADE with a blind faith in Multi-Agent Systems.

Requirements

Ok, first things first. Check that your computer has everything you need in order to be able to run SPADE:

- Python 2.5 or above
- libevent (optional, only if your system is Mac OS X 10.4 or above)
- TurboGears (optional, only if you want access to the GUI)
- Psyco python optimizer (optional), to speed up the execution of the platform and the agents

Installation

Grab the SPADE deliverable package at http://spade.gti-ia.dsic.upv.es. Uncompress it and launch the **configure.py** script in order to configure the platform. You can provide a parameter to this script, which is the machine's hostname and that will be used as the platform's name. Finally, launch the **runspade.py** script and see your platform rise before your eyes. After a few seconds you should see the green word **Done** on the screen confirming that the platform was succesfully launched. In a nutshell:

```
$ tar xvzf spade-2.0.0.tar.gz
$ cd spade
$ python configure.py myhost.myprovider.com
$ python runspade.py
```

And that's it. If everything went well, congratulations! You have succesfully installed SPADE! If something went wrong, don't panic!, head to chapter XX "Troubleshooting" for answers or follow the rest of this manual to know a lot more about SPADE.

Chapter 3. Fundamental Concepts

This chapter presents the basic concepts of the SPADE platform and its agents.

The Agent Platform

As we said in chapter XXX, the agent platform is the environment where the agents are deployed. Think of it both as the *"home"* of the agents and the main server of a network from a distributed application. In our case, that platform is SPADE, an agent platform based on the Jabber/XMPP protocols and technology.

The SPADE platform is installed and executed on a host. That host becomes the centre of the multi-agent system that we are going to build. In order to properly deploy the SPADE platform, it needs to be configured first to suit the host where it is going to be. Along with the SPADE distribution comes a tool called **configure.py** which is a script to automatically configure SPADE for you. In most scenarios, the only configuration you need to do for SPADE to work is to invoke this tool with the desired hostname to use as the platform hostname. E.g:

```
$ python configure.py myhost.myprovider.com
```

Then, SPADE will be configured to work as a platform serving at the **myhost.myprovider.com** domain.

Internally, the **configure.py** script creates the configuration files that SPADE needs to work. This configuration is stored in the **etc/** subdirectory inside the SPADE installation path. You can see how to perform advanced configuration procedures in chapter XXX.

After a succesfull configuration, you can try to run SPADE for the first time. To do so, use the **runspade.py** script that is provided with the SPADE distribution. This script will launch the SPADE platform and set it ready to start working with agents. You should see something along the lines of:

```
$ python runspade.py
SPADE 2.0 <garanda@dsic.upv.es> - http://spade.gti-ia.dsic.upv.es/
Starting SPADE...... [done]
```

You may also see some warnings and additional messages while the platform is starting. That's fine, we will discuss them later. You are now ready to start loading agents into the platform.

If you wish to know more about SPADE's configuration, please head to chapter XXX. If you wish to learn how to make your first SPADE agents, head on to the next chapter.

Chapter 4. Basic agents

This chapter explains the logics behind SPADE agents and how to make simple agents for your MAS.

The SPADE Agent Library

As stated earlier, the SPADE Agent Library is a software component to ease and help the development of software agents that work with the SPADE platform. It provides some classes, functions and data structures to help the programmer easily deploy SPADE-based solutions in a rapid-application-development fashion.

Currently, the SPADE Agent Library is only available for the Python programming language, and it is available as a module (that you can download at http://spade.gti-ia.dsic.upv.es). There is also a full reference documentation of the library that you can use to see the details of every element.

The SPADE agent model

The Agent Model is basically composed of a connection mechanism to the platform, a message dispatcher, and a set of different behaviors that the dispatcher gives the messages to. Every agent needs an identifier called Jabber ID *a.k.a.* **JID** and a valid password to establish a connection with the platform.

The *JID* (composed by a **username**, an @, and a **server domain**) will be the name that identifies an agent in the platform, e.g. `myagent@myprovider.com`. The agent address (which is another important field on the Agent Identifier) would be the JID of the platform's ACC (e.g. `xmpp://acc.myprovider.com`). We have defined the prefix `xmpp://` for the XMPP addresses.

Connection to the platform

Communications in SPADE are handled internally by means of the Jabber protocol. This protocol has a mechanism to register and authenticate users against a Jabber server. Since the SPADE platform includes a Jabber server component, SPADE agents use the aforementioned mechanism to register in the server as Jabber clients. After a succesful register, each agent holds an open and persistent Jabber stream of communications with the platform. This process is automatically triggered as part of the agent registration process.

The message dispatcher

Each SPADE agent has an internal message dispatcher component. This message dispatcher acts as a mailman: when a message for the agent arrives, it places it in the correct *"mailbox"* (more about that later); and when the agent needs to send a message, the message dispatcher does the job, putting it in the communication stream. The message dispatching is done automatically by the SPADE Agent Library whenever a new message arrives or is to be sent.

The behaviors

An agent can run serveral behaviors simultaneously. A **behavior** is a task that an agent can execute using repeating patterns. SPADE provides some predefined behavior types: Cyclic, One-Shot, Periodic, Time-Out, Finite State Machine and Event Behavior. Those behavior types help to implement the different tasks that an agent can perform. The kind of behaviors supported by a SPA DE agent are the following:

- Cyclic and Periodic behaviours are useful for performing repetitive tasks.
- One-Shot and Time-Out behaviors can be used to perform casual tasks.
- The Finite State Machine allows more complex behaviors to be built.
- Event behaviors respond to some event that the agent perceives.

Every agent can have as many behaviors as desired. When a message arrives to the agent, the message dispatcher redirects it to the correct behavior queue. A behavior has a **message template** attached to it. Therefore, the message dispatcher uses this template to determine which behavior the message is for, by matching it with the correct template. A behavior can thus select what kind of messages it wants to receive by using templates.

My first SPADE agent

It's time for us to build our first SPADE agent. We'll assume that we have a working SPADE platform at `myhost.myprovider.com` with the standard configuration. A basic SPADE agent is really a Python script that imports the `spade` module and that uses the constructs defined therein. For starters, fire up you favorite Python editor, create a file called `myagent.py` and write this:

```
import spade
```

With this sentence, the SPADE Agent Library is imported and all its features become available. Let's start defining the base class for the agent:

```
class MyAgent(spade.Agent.Agent):
        def _setup(self):
                print "MyAgent starting . . ."
```

As you can see, we have derived a class from `spade.Agent.Agent`, which is the base class for all SPADE normal agents. Also, note that we have defined a method called `_setup()` . Every SPADE agent should override this method. It is where the initialization (or setup) code of the agent must be placed.

As this is a toy example, we don't need anything else. We complete the script to execute the agent:

```
if __name__ == "__main__":
        a = MyAgent("agent@myhost.myprovider.com", "secret")
        a.start()
```

The first thing the script does is to make an instance of our agent class. Mind the two parameters we provide: First is the **JID** of the agent, it contains its name (before the @) and the name of the agent platform (after the @). Second is the **Jabber password** for this particular agent.

Note that an agent has to coherently use the same JID and password combination in order to succesfully connect to a platform over time, just like a human user has to use the same username and password combination over time to access a Jabber server.

The next thing the script does is to actually start the agent with the `start()` method. It is very important to understand that when an agent is created, it does **not** start working automatically. The `start()` method must be called first in order to trigger the agent execution. Actually, it is in that moment when the `_setup()` method will be called.

To finish the example, just execute the script and see the results:

```
$ python myagent.py
MyAgent starting . . .
Agent: agent@myhost.myprovider.com registered correctly (inform)
```

As you can see, the line we put in `_setup()` is printed first, before the agent actually connects to the platform. Then, the agent informs that it has registered correctly in the platform.

And that's it! We have built our first SPADE Agent in 7 lines of code. Easy, isn't it? Of course, this is a very very dumb agent that does nothing, but it serves well as a starting point to understand the logics behind the SPADE platform.

An agent with a behavior

Let's build a more functional agent, one that uses an actual behavior to perform a task. As we stated earlier, the real programming of the SPADE agents is done mostly in the behaviors. Let's see how.

Let's create a cyclic behaviour that performs a task. In this case, a simple counter. We can modify our existing `myagent.py` script:

```
import spade
import time

class MyAgent(spade.Agent.Agent):
        class MyBehav(spade.Behaviour.Behaviour):
                def onStart(self):
                        print "Starting behaviour . . ."
                        self.counter = 0

                def _process(self):
                        print "Counter:", self.counter
                        self.counter = self.counter + 1
                        time.sleep(1)

        def _setup(self):
                print "MyAgent starting . . ."
                b = self.MyBehav()
                self.addBehaviour(b, None)

if __name__ == "__main__":
        a = MyAgent("agent@myhost.myprovider.com", "secret")
        a.start()
```

As you can see, we have defined a custom behaviour called `MyBehav` that inherits from the spade.Behaviour.Behaviour class, the default class for all behaviours. This class represents a cyclic behaviour with no specific period, that is, a loop-like behaviour.

You can see that there is a method called onStart() in the behaviour. This method is similar to the _setup() method of the agent class. It is executed just before the main iteration of the behaviour begins and it is used for initialization code. In this case, we print a line and initialize the variable for the counter.

Also, there is the _process() method, which is very important. In most behaviours, this is the method in which the core of the programming is done, because this method is called on each iteration of the behaviour loop. It acts as the *body* of the loop, sort of. In our example, the _process() method prints the current value of the counter, increases it and then waits for a second (to iterate again).

Now look at the _setup() method of the agent. There, we make an instance of MyBehav and add it to the current agent by means of the addBehaviour() method. The first parameter of this method is the behaviour we want to add, and the second one is the template associated to that behaviour, but we will talk later about templates.

Let's test our new agent:

```
$ python myagent.py
MyAgent starting . . .
Starting behaviour . . .
Agent: agent@myhost.myprovider.com registered correctly (inform)
Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
Counter: 6
Counter: 7
```

. . . and so on. As we have not set any end condition, this agent would go on counting forever.

The default spade.Behaviour.Behaviour class is great for cycling tasks that need to be done in a loop fashion. However, in this case we are expecting the behaviour to execute the body of the loop more or less every second. There is a much better way of doing this, and it is by using the spade.Behaviour.PeriodicBehaviour class:

```
import spade

class MyAgent(spade.Agent.Agent):
        class MyBehav(spade.Behaviour.PeriodicBehaviour):
                def onStart(self):
                        print "Starting behaviour . . ."
                        self.counter = 0

                def _onTick(self):
                        print "Counter:", self.counter
                        self.counter = self.counter + 1

        def _setup(self):
                print "MyAgent starting . . ."
                b = self.MyBehav(1)
                self.addBehaviour(b, None)
```

```
if __name__ == "__main__":
        a = MyAgent("agent@myhost.myprovider.com", "secret")
        a.start()
```

Note the new parameter we pass on to the `MyBehav` object upon instantiation. It's the period of the execution of the task, measured in seconds. Also, note that in this case we do not use the `_process()` method (as it is internally used by the behaviour), but instead we use the `_onTick()` method that is executed exactly every period.

So what happens if you want a behaviour that doesn't execute neither a cyclic nor a periodic task? What if you want a behaviour that performs a given task once and then finishes? Is it possible? Well, SPADE is here to help you out. Please welcome the `spade.Behaviour.OneShotBehaviour` class.

As the name says, OneShot behaviours execute an spontaneous task once and then finish. Let's see this example:

```
import spade

class MyAgent(spade.Agent.Agent):
        class MyBehav(spade.Behaviour.OneShotBehaviour):
                def onStart(self):
                        print "Starting behaviour . . ."

                def _process(self):
                        print "Hello World from a OneShot"

                def onEnd(self):
                        print "Ending behaviour . . ."

        def _setup(self):
                print "MyAgent starting . . ."
                b = self.MyBehav()
                self.addBehaviour(b, None)

if __name__ == "__main__":
        a = MyAgent("agent@myhost.myprovider.com", "secret")
        a.start()
```

The `_process()` method returns, as it is where the main code of this kind of behaviours go. Note that a new method of the behaviours has appeared: `onEnd()` . It is very similar to `on-Start()` and it's there to work as a finalizer method. It is executed once the code on `_process()` has ended and just before the behaviour terminates its execution. Let's see the result of this script:

```
$ python myagent.py
Agent: agent@myhost.myprovider.com registered correctly (inform)
MyAgent starting . . .
Starting behaviour . . .
Hello World from a OneShot
Ending behaviour . . .
```

Pretty straightforward, eh? Let's move on to something else.

`spade.Behaviour.TimeOutBehaviour` is the class to build behaviours that execute a task once, but only after a specific amount of seconds have passed since the behaviour becomes active (i.e. a timeout is triggered). The code of the task must be placed in a `timeOut()` method of the behaviour:

```
import spade

class MyAgent(spade.Agent.Agent):
        class MyBehav(spade.Behaviour.TimeOutBehaviour):
                def onStart(self):
                        print "Starting behaviour . . ."

                def timeOut(self):
                        print "The timeout has ended"

                def onEnd(self):
                        print "Ending behaviour . . ."

        def _setup(self):
                print "MyAgent starting . . ."
                b = self.MyBehav(5)
                self.addBehaviour(b, None)

if __name__ == "__main__":
        a = MyAgent("agent@thx1138.dsic.upv.es", "secret")
        a.start()
```

Other behaviour types are `spade.Behaviour.FSMBehaviour`, which represents a Finite State Machine, and `spade.Behaviour.EventBehaviour`, which is a special type of behaviour set to be triggered after an event happens in the agent. These are advanced types of behaviours and are covered in chapter XXX.

Agent Communication: Sending and Receiving Messages

As you know, messages are the basis of every MAS. They are the centre of the whole *"computing as interaction"* paradigm in which MAS are based. So it is very important to understand which facilities are present in the SPADE Agent Library to work with agent messages.

First and foremost, there is the Agent Identifier or AID. It is the information that identifies an agent and provides some information on how to communicate with it, i.e., its addresses. Every agent has a unique AID. In SPADE, that AID is composed of a unique name and a set of addresses which can be used to communicate with the agent. In most cases, an agent will use its JID as its name, and will have only one single (Jabber) address. For example:

`Name = "agent@myhost.myprovider.com" . Addresses = ["xmpp://agent@myhost.myprovider.com"] .`

The SPADE Agent Library uses the class `spade.AID.aid` to represent AID information. Translating the previous example into actual code would produce something like this:

```
import spade

identification = spade.AID.aid(name="agent@myhost.myprovider.com", addresses=["xmp
```

Second, there is the class that represents a FIPA-ACL message. This class is `spade.ACLMessage.ACLMessage` , and you can instantiate it to create new messages to work with. The class provides several methods to construct and de-construct messages, based on the fields present in standard FIPA-ACL Messages. When a message is ready to be sent, it can be passed on to the `send()` method of the agent, which will trigger the internal communication process to actually send it to its destination. Here is a self-explaining example:

```
import spade

class MyAgent(spade.Agent.Agent):
        class InformBehav(spade.Behaviour.OneShotBehaviour):

                def _process(self):
                        # First, form the receiver AID
                        receiver = spade.AID.aid(name="receiver@myhost.myprovider.
                                     addresses=["xmpp://receiver@myhost.myprovider

                        # Second, build the message
                        self.msg = spade.ACLMessage.ACLMessage()  # Instantiate th
                        self.msg.setPerformative("inform")        # Set the "infor
                        self.msg.setOntology("myOntology")        # Set the ontolo
                        self.msg.setLanguage("English")           # Set the langua
                        self.msg.addReceiver(receiver)            # Add the messag
                        self.msg.setContent("Hello World")        # Set the messag

                        # Third, send the message with the "send" method of the ag
                        self.myAgent.send(self.msg)

        def _setup(self):
                print "MyAgent starting . . ."
                b = self.InformBehav()
                self.addBehaviour(b, None)

if __name__ == "__main__":
        a = MyAgent("agent@myhost.myprovider.com", "secret")
        a.start()
```

The previous example is also an opportunity to introduce the `myAgent` attribute that can be used in any behaviour. It is a reference to the agent that holds the behaviour and thus it can be used as a shortcut to access any method or attribute the agent object has. In this particular case, the command `self.myAgent.send(self.msg)` is accessing the `send` method of the agent through the `myAgent` shortcut.

There are two basic methods in message communication: `send` , which you already know, and `_receive` , which, as its name says, serves the purpose of receiving a message. At any given moment, a behaviour can make use of the `_receive` method in order to receive a message.

Regarding send and `_receive`>

WARNING: Please take note that whereas `send` is a method of the agent, `_receive` is a method of the **behaviour**. This difference is very important. All the behaviours send messages the same way, but, as stated earlier, every behaviour has its unique *message template* that specifies which kind of messages it may receive, and thus, each behaviour uses its own `_receive` method to receive its own messages.

Regarding messages at initialization time

WARNING: A SPADE agent cannot send nor receive messages until its behaviours are active. That is, do NOT place calls to the `send` or `_receive` methods inside the `_setup` and `takeDown` methods.

The `_receive` method accepts two parameters: a boolean representing wether the behaviour must block waiting for a message, and the number of seconds it must wait before resuming execution. Of course the latter is only needed if the former is set to "True". If only the first parameter is present and set to "True", the behaviour will block indefinitely.

If there is a message for the behaviour calling the `_receive` method, the call will return an object of the `spade.ACLMessage.ACLMessage` class representing such message. If there is no message and the call is non-blocking or it times out, the return value will be `None`.

In order to set a message template for a behaviour, you must first define such template. Or you can set a behaviour to be the *default behaviour*, which means it will receive all kind of messages (no need to specify a template). To define a template you can use the classes `spade.Behaviour.ACLTemplate` and `spade.Behaviour.MessageTemplate` . First, you need to instantiate a `spade.Behaviour.ACLTemplate` object (which quite the same methods as a `spade.ACLMessage.ACLMessage` object) and edit it to match your requirements (i.e. set the ontology, the language, the protocol, etc...). Then, you must wrap it around a `spade.Behaviour.MessageTemplate` object. That is the object that you will pass on to the `addBehaviour` method, along with the behaviour itself. Let's see an example:

```
import spade

class MyAgent(spade.Agent.Agent):
        class ReceiveBehav(spade.Behaviour.Behaviour):
                """This behaviour will receive all kind of messages"""

                def _process(self):
                        self.msg = None

                        # Blocking receive for 10 seconds
                        self.msg = self._receive(True, 10)

                        # Check wether the message arrived
                        if self.msg:
                                print "I got a message!"
                        else:
                                print "I waited but got no message"

        class AnotherBehav(spade.Behaviour.Behaviour):
                """This behaviour will receive only messages of the 'cooking' onto

                def _process(self):
                        self.msg = None

                        # Blocking receive indefinitely
                        self.msg = self._receive(True)

                        # Check wether the message arrived
                        if self.msg:
                                print "I got a cooking message!"
                        else:
                                print "I waited but got no cooking message"

        def _setup(self):
                # Add the "ReceiveBehav" as the default behaviour
                rb = self.ReceiveBehav()
                self.setDefaultBehaviour(rb)

                # Prepare template for "AnotherBehav"
                cooking_template = spade.Behaviour.ACLTemplate()
                cooking_template.setOntology("cooking")
                mt = spade.Behaviour.MessageTemplate(cooking_template)

                # Add the behaviour WITH the template
                self.addBehaviour(ab, mt)

if __name__ == "__main__":
        a = MyAgent("agent@myhost.myprovider.com", "secret")
        a.start()
```

In the previous example, there are a couple of behaviours. The `ReceiveBehav` behaviour will wait for a new message for 10 seconds and then print wether a message arrived or not (and then start over). The `AnotherBehav` will wait indefinitely for a *"cooking"* message. How do we specify a *"cooking"* message? In this case, by means of the `cooking` ontology. So, when the behaviours are instantiated in the `_setup` method of the agent, we create the `cooking_template` template (an object of the `spade.Behaviour.ACLTemplate` class) and set its ontology to `cooking`. This means that whenever a message arrives with its ontology set to `cooking`, it will match with the template. Then, we create an object of the `spade.Behaviour.MessageTemplate` class to wrap the `cooking_template` and add the `AnotherBehav` behaviour to the agent with the new message template associated (the second parameter of the `addBehaviour` method).

What's next

In this chapter we have learned how to make SPADE agents with basic functionality and, specially, how to set up behaviours and send and receive messages. Coming up next we will learn how to make use of the facilities the platform offers in order to contact agents and services.

Chapter 5. Platform services

This chapter presents the services and facilities the SPADE platform offers to its agents in the form of the standard AMS and DF FIPA components.

The SPADE Agent Management System (AMS)

The SPADE AMS is our implementation of the standard FIPA AMS component to manage, identify and search for agents in the platform. It is a special platform agent that is always present in all SPADE platforms and that is started automatically whenever the platform is initiated. It manages the registration of the agents in the platform. So, when you start one of your agents and the registration process is autmatically taken care of by the SPADE Agent Library, it is the AMS who is on "the other side of the counter", managing the registry of agents. It also gets to de-register an agent once its life cycle has ended.

Additionally, the AMS offers some services that any agent inside the platform can use. First is the Platform Information service, a basic query that gives the agent some information regarding the platform. It can be invoked with the `getPlatformInfo` method of the agents. Let's see an example:

```
import spade

class MyAgent(spade.Agent.Agent):
        class MyBehav(spade.Behaviour.OneShotBehaviour):

                def _process(self):
                        pi = self.myAgent.getPlatformInfo()
                        print "Platform Information: ", pi

        def _setup(self):
                print "MyAgent starting . . ."
                b = self.MyBehav()
                self.addBehaviour(b, None)

if __name__ == "__main__":
        a = MyAgent("agent@myhost.myprovider.com", "secret")
        a.start()
```

```
$ python agent.py
Agent: agent@myhost.myprovider.com registered correctly (inform)
Platform Information: (ap-description :name xmpp://acc.myhost.myprovider.com :ap-s
(set (ap-service :name xmpp://ams.myhost.myprovider.com :type fipa.agent-managemen
:addresses (sequence acc.myhost.myprovider.com))
```

As you can see, the method returns the information collected from the AMS in the FIPA-SL semantic language (more on that later).

Agents can also use the Search Agent service. This service allows them to look for a fellow agent registered in the platform using some search criteria. In order to introduce those criteria, the class `spade.AMS.AmsAgentDescription` must be used. It represents some data fields that can be filled in with agent information: the *name* of the agent, its *ownership* and its *state*. Once an object of the `spade.AMS.AmsAgentDescription` class has been filled with the search criteria, you can use it with the `searchAgent` method of an agent:

```
import spade

class MyAgent(spade.Agent.Agent):
        class MyBehav(spade.Behaviour.OneShotBehaviour):
                def onStart(self):
                        print "Starting behaviour . . ."

                def _process(self):
                        print "I'm going to search for an agent"
                        aad = spade.AMS.AmsAgentDescription()
                        search = self.myAgent.searchAgent(aad)
                        print search

                def onEnd(self):
                        print "Ending behaviour . . ."

        def _setup(self):
                print "MyAgent starting . . ."
                b = self.MyBehav()
                self.addBehaviour(b, None)

if __name__ == "__main__":
        a = MyAgent("agent@myhost.myprovider.com", "secret")
        a.start()
```

```
((result  (set (ams-agent-description
:name (agent-identifier
:name ams.myhost.myprovider.com
:addresses
(sequence
xmpp://ams.myhost.myprovider.com
)
)

:ownership SPADE
:state active)
 (ams-agent-description
:name (agent-identifier
:name df.myhost.myprovider.com
:addresses
(sequence
xmpp://df.myhost.myprovider.com
)
)

:ownership SPADE
:state active)
 (ams-agent-description
:name (agent-identifier
:name agent@myhost.myprovider.com
:addresses
(sequence
xmpp://agent@myhost.myprovider.com
)
)

:ownership agent@myhost.myprovider.com
)
 )))
```

Another service offered by the AMS is the Modify service. It allows for an agent to modify its own information, published in the AMS, in order to update it to reflect its current status or make it easier for other agents to find it. It can be invoked using the `modifyAgent` method of the agents and a `spade.AMS.AmsAgentDescription` object that includes the updated information of the agent:

```
import spade

class MyAgent(spade.Agent.Agent):
        class MyBehav(spade.Behaviour.OneShotBehaviour):
                def onStart(self):
                        print "Starting behaviour . . ."

                def _process(self):
                        print "I'm going to modify my data"
                        aad = spade.AMS.AmsAgentDescription()
                        aad.ownership = "FREE"
                        result = self.myAgent.modifyAgent(aad)
                        if result:
                                print "Modification OK"
                        print "I'm going to check the modification"
                        search = self.myAgent.searchAgent(aad)
                        print search

                def onEnd(self):
                        print "Ending behaviour . . ."

        def _setup(self):
                print "MyAgent starting . . ."
                b = self.MyBehav()
                self.addBehaviour(b, None)

if __name__ == "__main__":
        a = MyAgent("agent@myhost.myprovider.com", "secret")
        a.start()
```

```
$ python agent.py
Agent: agent@myhost.myprovider.com registered correctly (inform)
MyAgent starting . . .
Starting behaviour . . .
I'm going to modify my data
Modification OK
I'm going to check the modification
((result  (set (ams-agent-description
:name (agent-identifier
:name agent@myhost.myprovider.com
:addresses
(sequence
xmpp://agent@myhost.myprovider.com
)
)

:ownership FREE
)
 )))
```

The SPADE Directory Facilitator (DF)

SPADE's DF serves as *yellow pages* index where the services offered by the agents are published for others to find. The way of publishing services is by uploading to the DF agent a set of *service descriptions* that contain information about services and how to access them. Each agent can publish its own set of *service descriptions*.

The interface for an agent to publish its own services is a bit unconventional and stilted in order to comply with the FIPA specification. First, the agent has to describe its services by using the `spade.DF.ServiceDescription` class. The objects of this class store the information of a particular service: its name, type, protocols it requieres, ontologies associated, content languages, its ownership and some additional properties it may have. This information can be set up using the dedicated methodos of these objects.

Next, the agent has to instantiate a `spade.DF.DfAgentDescription` object (which is quite similar to a `spade.AMS.AmsAgentDescription` object) which will host the collection of `spade.DF.ServiceDescription` objects. This is the object that will be passed to the `registerService` method of the agent in order to register all the services. Each agent can upload only one set of services at a time. In order to remove a service, an agent would use the `registerService` method with a `spade.DF.DfAgentDescription` containing the active services minus the one that is to be removed (that is why we encourage agent developers to keep a local copy of a `spade.DF.DfAgentDescription` object with a set of the active services). Again, this interface is not straightforward but FIPA-compliant, and so an example is helpful:

```
SERVICES EXAMPLE
```

What's next

In this chapter we have learned what facilities do the AMS and DF offer to the agents. Next chapter, we will get our hands even dirtier as we venture through the black marshes of SPADE's advanced behaviour types.

Chapter 6. Advanced Behaviour Types

This chapter introduces the reader to the advanced behaviour types present in SPADE agents and how to use them to get impresive agents.

The Finite-State-Machine