Chapter 2

Agents

2.1 Agent definition

An agent is per definition an entity which acts autonomously or semi-autonomously. This means an agent acts on its own, rather than simply obeying direct commands. In law, an agent acts on behalf of a principal client. In other contexts, whenever we use the word agent, it is used in the same sense. For instance, a real estate agent acts on behalf of a seller, shipping agents, travelling agents and booking agents all act on behalf of someone else.

The Online Etymology Dictionary says:
Agent: *"One who acts", from L. agentem (nom. agens, gen. agentis), prp. of agere "to set in motion, drive, lead, conduct"*.[1]

In terms of software development, an agent is an autonomous, or semi-autonomous, piece of software. Agents can monitor and wait for a given scenario, and then act on behalf of their user. For instance, an agent could manage your mail inbox or time scheduler and notify you when something important is pending for your attention.

Daemon processes, web browsers and email clients, also fall under the agent definition. A system process is often referred to as a system agent and web browsers are often called user agents. The web browser interprets html, JavaScript and other web standards and presents the data visually to its user.

Agents always represents a somewhat intelligent piece of software and are often associated with the term *distributed artificial intelligence*. Most agents have a very limited purpose. However, the cooperation among several agents make up a global objective.

The webopedia says:

In computer science, there is a school of thought that believes that the human mind essentially consists of thousands or millions of agents all working in parallel. To produce real artificial intelligence, this school holds, we should build computer systems that also contain many agents and systems for arbitrating among the agents' competing results. [2]

Although agents traditionally performs quite narrow predefined tasks, there would be none or few limitations to what kind of business logic could be performed by a cluster of agents. Nowadays, with both public and privately held IT providers focusing more and more on standardization and aspect orientation[3], open agent platforms could easily fit the idea of separating different aspects of an IT solution.

2.2 A brief history of software agents

Since the late 80's, agent technology has been a field of interest among the largest computer software and research companies. In 1988 Apple released a video called Knowledge Navigator [4], showing the world how agent technology would be used twenty years into the future. The video has gained quite a reputation and is still subject of discussion this many years later. Apple predicted that everybody would be represented by an agent in the world of digital communications. Another agent could present schedule, mail, audio messages, news and video streams to the user. The very same idea was later adopted by the AgentCities project, where everybody would have an agent represent themselves in virtual cities on the internet.

Today, many of these visionary features are quite common. With the growing number of services available over the internet, including communications, personal planning, digital news, shopping and entertainment, most of these features are a part of peoples daily routines to some extent. The business is moving towards greater interoperability with focus on Service Oriented Architecture (SOA) based on non-proprietary middleware standards like XML-based web services. We surround ourselves with wireless equipment like mobile phones and handheld computers. These are often limited by small size and processing power. The agent abstraction could therefore be a paradigm shift in the way we use handheld or portable devices.

MIT media labs with their visionary leader Nicholas Negroponte spearheaded the Things That Think (TTT) consortium [5] back in 1995 in collaboration with HP Labs and Motorola. Their goal was to bring artificial intelligence into everyday appliances with the help of common standards and software agent technologies. One of the most famous product of this think tank is the refrigerator that would tell you when you are out of milk - a part of the intelligent house of the future. From Norway, both NTNU in Tronheim and UiTø in Tromsø, in collaboration with Telenor Research and Development [6], is a part of the TTT consortium. Recently, a

head project with the TTT consortium was the Telenor Electronic Shepherd project [7] - electronic monitoring of sheep using WiFi.

A few months back, Computerworld wrote that NASA has been uploading software into the Earth Observing-1 satellite[8], turning it into a testbed for autonomous agents. This NASA project is called the Autonomous Sciencecraft Experiment (ASE) [9].

2.3 FIPA

FIPA is a standards body for agent development and current key collaborators include Siemens, IBM, The Boeing Company, Toshiba and Mitsubishi[10]. FIPA started back in 1996. At the time, there were about 60 proprietary agent systems in competition. As the internet became available to the masses, software distribution took a new direction with more focus on open standards. The members of FIPA started describing possible scenarios and application types, semantics and services. Some members of the organization believed that their research also would have great significance in the development of physical entities such as robots and industrial automation devices. Therefore it was called the foundation for *physical* agents. Currently FIPA has about one hundred specifications in their repository [11] and as of June 2003, the organization had about 50 sponsoring members from around the world. The project has apparently gathered great support from research communities around the globe and has expanded to better include technologies such as web services and mobile platforms. Some of the key FIPA specifications include:

- Agent lifecycle management
- Message transport
- Message structure
- Inter-agent interaction protocols
- Ontology
- Security

The FIPA specifications describes a safe environment for agents and the language of communicaton between them. Both commercial and open-source projects like the JADE framework integrates tightly with the FIPA agent specifications. Common key implementations are:

- The Agent Management System (AMS)

- The Directory Facilitator (DF)
- The Message Transport Service (MTS)

These will be described in more detail in chapter 3.

2.4 Status of agent research

Around the year 2002, many of the greatest software companies in the world closed down their agent research projects. This could be the result of cutbacks due to the low profit years of the IT business in 2001/2002. Far from all of such projects would yield profit in within the near future and with the profit driven strategies of the IT companies in mind, agent labs would be an easy target of cutbacks. Brittish Telecom (BT) closed down their agent project Zeus in 2002. Hewlet Packard (HP) first closed down their application server project BlueStone and later closed down their Agent Labs. FIPA-OS, Grasshopper, the Java Agent Service (JAS) all seems to have been inactive for the last couple of years.

One guess would be that the business is moving onto more service driven applications. This somewhat rules out the need for agents scouring the internet for information themselves rather than service providers offering their content with non-proprietary xml based standards. This however is entirely based on the collective good will of the business and there might still be need for agent applications. For instance, crawler-like applications may always play an important role in the field of data gathering and data mining.

JADE (see chapter 3) is one of the few projects that still is in development, and a new version, 3.3, was released this spring with exciting new features.

BT and HP discontinued their agent projects in 2002. Norwegian telecom leader Telenor discontinued all their agent research around 2000.

Chapter 3

JADE

3.1 Introduction

The research on software agents has obviously been lacking involvement over the last couple of years. Agents are however still widely used and the need for solid and powerful frameworks is therefore still an issue.

Telecom Italia Labs (TILabs)' JADE is one of the frameworks which is still standing despite the downsized focus on agent development. The framework is under continuous development and a new version was released just prior to the start of this project (see section 3.5).

This chapter will plow deeper into the JADE framework and outline the architecture and its main features.

3.2 Architecture

The JADE architecture is defined by platforms, containers, agents and behaviours. These can be considered the main building blocks of the JADE framework and are essential in order enable agent execution with the JADE framework. The following sections will provide an overview and explanation of how these components are tied together to form the JADE framework.

3.2.1 The platform

The platform is the lower level of a JADE agent system. The platform is used for managing containers, which again contains agents. The platform concept is quite abstract, and its boundaries are only defined by the containers belonging to it. To

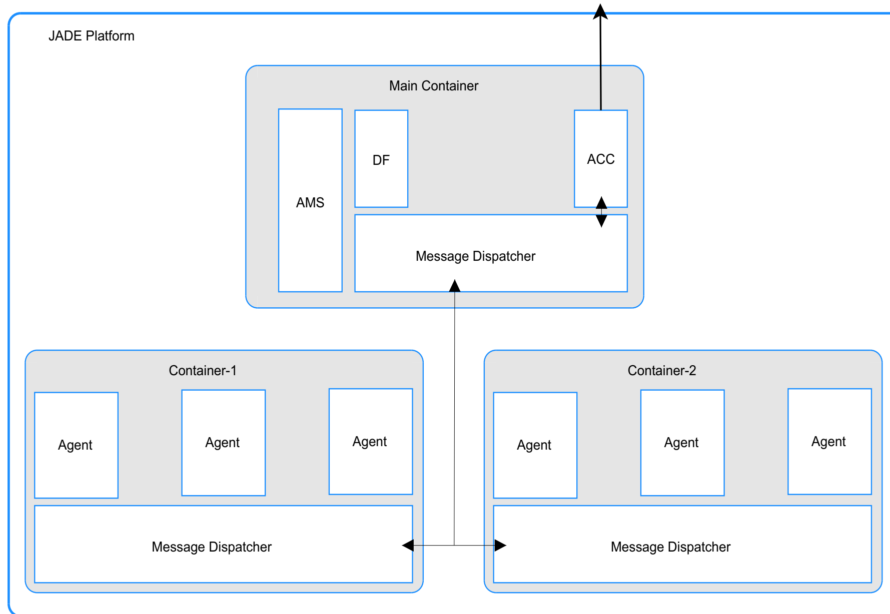

Figure 3.1: JADE agent platform model

put it simple: there can be no platform unless at least *one* container is active.

A platform can be distributed over several hosts and the platform has *one* main container which contains the Agent Management System (AMS) and the Directory Facilitator (DF). Other containers that want to be a part of the platform have to connect to the host where the platform was initialized and the main container was started. See figure 3.1 for an overview of the JADE platform architecture.

Platform communication

Inter-platform communication, communication between containers belonging to different platforms, are based on Common Object Request Broker Architecture (CORBA) and Internet Inter-ORB Protocol (IIOP) or the HyperText Transfer Protocol (HTTP). Since version 3.2 HTTP is the default protocol used for inter-platform communication. By using standard protocols agents can communicate with each other even if they are running on any other platform as long as they support the FIPA standards.

Communication between agents on the same platform is done in two ways. If the agents are in the same container, the Agent Communication Language (ACL) messages are passed using event objects. If the agents are on different containers

the ACL message objects are sent by using Remote Method Invocation (RMI)[12].

The Message Transport System (MTS), including the Agent Communication Channel (ACC) and message dispatchers, is the component within the platform controlling the exchange of messages between agents in the platform and also messages to and from other platforms.

3.2.2 The containers

The containers in the JADE architecture are what contains the agents. Containers in a platform can run on the same host, but often they are distributed over several hosts. Each container runs in a separate Java Virtual Machine (JVM), providing the agents with a complete runtime environment.

The main container is where the AMS and DF resides. There can never be more than one main container in a platform and all other containers have to connect to this container when they join the platform. Otherwise there are no limitations to how many containers there can be in a platform, other than limited resources.

3.2.3 The JADE agents

The JADE framework comes bundled with several agents which simplifies the platform management.[13]

One of the core JADE agents is the AMS which controls and manages all the other agents. This agent is the backbone of the platform; without it the platform breaks down. The AMS provides white-page and life-cycle services and maintains a directory of Agent Identifiers and states.

The DF is a yellow pages agent which provides the agent platform with a service where the agents can register themselves and their services. Other agents can then make requests to the DF to lookup agents which are providing the services they need. Use of the DF is explained in more detail in section 3.4 and is demonstrated in appendix A.4.

Remote Management Agent provides you with a graphical user interface for managing your JADE platform with all its containers and agents. From this interface you can easily start other agents and add containers and remote platforms for easy management.

Dummy Agent is an agent used for debugging other agents. You can create ACL messages and send them to your agents to examine the response closely.

Sniffer Agent can be used to ”sniff” ACL messages exchanged between agents in your platform. This agent also contains a graphical user interface for

displaying them in a diagram that resembles Unified Modeling Language (UML).

Inspector Agent is used for monitoring the life cycle and state of the agents running in the platform. You can inspect the behaviours of your agents and execute them step by step.

DF Agent is the Directory Facilitator. It provides the platform with a yellow pages service. The *DF GUI* is a complete graphical user interface which is used by the DF of JADE, but it can also be used by other DFs the user might need. This can be used to create a complex network of yellow pages. The DF user interface provides easy access to the DF's knowledge base, manage DF records and connect different DFs into a network.

LogManagerAgent is an agent that enables setting of logging information at runtime. This can be used to change the logging level.

SocketProxyAgent is an agent acting as a bidirectional gateway between the JADE platform and a standard TCP/IP connection. ACL messages from the platform are translated into simple ASCII strings and sent over a connection. The other way around; ACL messages can be tunneled via the TCP/IP connection into the JADE platform.

3.2.4 The behaviours

The `Behaviour` class of the JADE platform makes it possible to start only one thread per agent, a major advantage on limited platforms like cellular phones. The `Behaviour` class switches a lot faster than traditional threads in Java because synchronization and resource management is done by the JADE platform itself. Using behaviours will therefore provide better performance than Java threading[14].

All JADE behaviors are located in the `jade.core.behaviours` package.

The following sections describe the most common behaviours and show short and simple examples of their use.

Behaviour

A simple behaviour implements two methods: The business code method `action()` and the `done()` deconstructor. An efficient agent may have to include several behaviours.

The following example shows a simple implementation. It includes one behaviour that has three steps.

```java
public class ThreeStepBehaviour extends Behaviour{
    private int step = 0;

    public void action(){
        switch(step){
            case 0:
                //Perform first step
                step++;
                break;
            case 1:
                //Perform second step
                step++;
                break;
            case 2:
                //Perform third step
                step++;
                break;
            default:
                break;
        }
    }

    public boolean done(){
        return (step == 3);
    }
}
```

OneShotBehaviour

The `OneShotBehaviours.action()` method is only called once. The class already implements the `done()` method, which always returns true.

Here is a short example of the use of a `OneShotBehaviour`:

```java
public class MyOneShotBehaviour extends
    OneShotBehaviour{

    public void action() {

        //Code here will only be executed once.
    }
}
```

CyclicBehaviour

The `CyclicBehaviour` continues to execute until the `done()` method returns true. This type of behaviour is often used in agents who need to constantly check for incoming `ACLMessages` or perform other cyclic tasks.

Here is a short example showing how a `CyclicBehaviour` can be used to check for incoming messages:

```
public class MyCyclicBehaviour extends CyclicBehaviour {

    public void action() {

        ACLMessage msg = receive();
        if(msg == null)
            block();
    }
}
```

In the above example, the `block()` method puts the behaviour on hold until the messaging system has any messages to deliver to that agent. This saves resources which can be used to execute behaviours for other agents or even other behaviours waiting to be executed for this agent.

WakerBehaviour

The `WakerBehaviour` executes its `onWake()` method after a number of milliseconds.

The example below shows how to add a `WakerBehaviour` which wakes up after 10 seconds:

```
public class MyWakeupAgent extends Agent{

    protected void setup(){

        addBehaviour(new WakerBehaviour(this, 10000) {
```

```
            protected void onWake() {

                //Code to execute when waking up
            }
        });
    }
}
```

Note that outdated JADE documentation may report that `handleElapsedTimeout()` should be used to execute the code of the behaviour. According to the API documentation of JADE 3.3 the `handleElapsedTimeout()` method will soon be deprecated and `onWake()` should be used instead.

The `WakerBehaviour` is a good way to schedule tasks that are to be performed by agents. An example of how this is done is provided among the demonstration agents for this project. See appendix A.1 for more details.

TickerBehaviour

The `TickerBehaviour` repeatedly calls the method `onTick()` with a certain interval.

This example shows a very simple implementation of a `TickerBehaviour` which will output the text ”Tick!” once every second:

```
public class MyTickerAgent extends Agent {

    protected void setup() {

        addBehaviour(new TickerBehaviour(this,1000) {

            protected void onTick() {

                System.out.println("Tick!");
            }
        });
    }
}
```

CompositeBehaviour

The `CompositeBehaviour` is an abstract behaviour class for behaviours constructed by many parts. This class is the base class of the `SequentialBehaviour`, `ParallellBehaviour` and `FSMBehaviour`.

SequentialBehaviour

The `SequentialBehaviour` is a `CompositeBehaviour` that executes its child behaviors in sequential order.

This example shows how a `SequentialBehaviour` is constructed by adding a `WakerBehaviour` and a `OneShotBehaviour` which will be executed sequentially:

```
public class MySequentialAgent extends Agent {

   protected void setup() {

      SequentialBehaviour seq =
        new SequentialBehaviour();

      //Add a WakerBehaviour
      seq.addSubBehaviour(new WakerBehaviour(this,
        250) {

        protected void onWake() {
        }
      });

      //Add a OneShotBehaviour
      seq.addSubBehaviour(new OneShotBehaviour {

        public void action() {
        }
      });

      addBehaviour(seq);
    }
}
```

ParallelBehavior

The `ParallelBehaviour` is a `CompositeBehaviour` that executes its child behaviours in parallel.

Example:

```
public class MyParallelAgent extends Agent {

   protected void setup() {

      addBehaviour(new ParallelBehaviour{

         //CustomBehaviour is an imaginary
         //class created by the agent developer.
         addSubBehaviour(new CustomBehaviour());
         addSubBehaviour(new WakerBehaviour(this,
            5000){

            protected void onWake() {
            }
         });
      }
   }
}
```

FSMBehaviour

The Finite State Machine Behaviour (FMSBehaviour) is a `CompositeBehaviour` that executes its child behaviors as defined by the user. The order is set with methods `registerFirstState`, `registerState` and `registerLastState`

This example shows a basic implementation:

```
public class MyFSMAgent extends Agent {

   protected void setup() {

      addBehaviour(new FSMBehaviour(this){

         //Again, CustomBehaviour is an imaginary
```

```
            //class written by the agent developer.
            registerFirstState(new CustomBehaviour());
            registerState(new CustomBehaviour());
            registerLastState(new WakerBehaviour(this,
               5000){

               protected void onWake() {

                  //Perform task
               }
            });
         }
      }
}
```

3.2.5 Ontologies

Agents need to share the same language, vocabulary and protocols in order to communicate in a way that makes sense. The JADE framework provides some help with this through the ontology classes. However, to get an efficient way of communication between your agents, a custom ontology is needed. Defining an ontology for your agents means defining a vocabulary and semantics for the content of the messages exchanged between them[15].

The JADE framework provides three ways for implementing communication:

Simple string messages The Agents send strings which represents the message content.

Serialized Java Objects This way of communication is used when more complex data structures have to be sent. Using simple string messages for this would result in a lot of parsing which requires resources and could create a bottle neck in the system. Serializing Java objects is avialable for systems where all agents are implemented in Java.

Extension of predefined classes By using this method the objects are defined to be transferred as an extension of predefined classes so that JADE can encode and decode messages in standard FIPA format. Using this implementation enables JADE agents to interoperate with other agent systems, even the ones implemented in other languages than Java.

In JADE, an ontology is implemented by extending the class `Ontology`. Instead of serializing the Java objects directly and sending them to the receiving agent

Table 3.1: Ontology objects - use cases

Case	Use of ontology objects
One agent requests another to perform a specific *task*	According to FIPA the content of the request message must be an *action*. The *task* is defined as an AgentAction-implementing object and the *action* is defined by an Action-implementing class. The *action* class needs to hold an identifier argument, the AID of the receiver, and the object describing the *task*.
One agent requests another to check if a proposition is true	According to FIPA the content of the request message must be the object representing the proposition. A proposition is defined implementing `Predicate`.
Same as first example, but the second agent sends a *result* back.	When an agent is replying to a request with complex objects a Result object is wrapping the response objects. The response object is defined by implementing the `Concept` interface.

Source: JADE Tutorial and Primer [15]

they are wrapped into specific terms and concepts defined in the ontology class. There are three interfaces that are dealt with when defining ontologies: `Concept`, `AgentAction` and `Predicate`. These interfaces are implemented in *schema* classes which then are used in the ontology. These classes are `ConceptSchema`, `AgentActionSchema` and `PredicateSchema`. These three interfaces and classes are used for more sophisticated situations.

A schema and ontology for defining atomic elements such as *String*, *Integer* and *Float* among others are already provided by JADE through the `PrimitiveSchema` and `BasicOntology` classes.

When to use the ontology objects are briefly described in table 3.1.

3.3 Mobility

JADE provides support for intra-platform mobility. Agent mobility means that agents can move between containers within the same platform. Mobility is implemented by serializing the agents and needed resources to a Java ARchive (JAR) file and then sent to the location where the agent is moving.

Inter-platform mobility is not supported due to the need for standards. If the JADE platform were to let agents move out from the platform there would be no

guarantee that the receiving platform would be able to deserialize the received objects. If both platforms were running JADE, inter-platform would work, but such an implementation would deviate from the standards.

Agent mobility has many advantages, both when it comes to load balancing and processing benefitting from random patterns. An example of the last is a web crawler agent: It could easily generate a peak in the remote server's statistics if located on one single server, but if the agent were moving around; the peak would not appear as the traffic generated by the crawler would spread out over different hosts.

3.4 Directory Facilitator

The Directory Facilitator is the yellow pages service of the JADE platform. Through this service the agents can register them selves in order for other agents to find their services. The user of the DF opens for a much more independent connection between the agents. No agents need to have hard-coded, or configured references to the agents offering the services they need; they lookup this information in the DF whenever they need it and they get a list of agents providing these services in return.

Different DF services can also be combined to create very complex lookup directories.

There are three main classes which are used during a DF request:

`DFAgentDescription` is the class describing the agent, either the one being registered or the type of agent you want to find when querying the DF.

`ServiceDescription` is the class describing a service which the agent is providing. This class is fed with information about the service, such as type, what ontologies and languages it supports and various optional properties can be added as well.

`DFService` is the access point to the DF service. This is the class used for all operations on the DF.

For code examples on how to use these classes, please see the DF Register and DF Lookup demo agents in appendix A.4.

3.5 JADE 3.3

After the preliminary report for this project was finished a new version of JADE was released. Version 3.3 comes with new features and enhancements that make the framework even better. Among the improved features are scalability and database storage of DF information which means decreased search times.[16]

3.5.1 WSIG

A new add-on in the 3.3 release of JADE is the Web Service Integration Gateway (WSIG), developed by Whitestein Technologies AG. The 0.4 version shipped with this release of JADE should be considered a beta version[17], but it is a very interesting add-on to the JADE platform and therefore we find it worth mentioning in this report.

By implementing WSIG, the JADE framework introduces a seamless integration between web services and agents, which are often seen as competitors. WSIG enables agents to invoke web services as they would with any other agent service and web services to invoke agent services as they would with any other web service. The WSIG acts as a bidirectional invocation gateway for the two types of services. The gateway is implemented as a JADE agent responding to changes in the DF. On any changes in the DF, WSIG converts the new entry into a Universal Description, Discovery, and Integration (UDDI) entry, thus enabling web services to look up the agent services as it would with any other web service. This translation also applies for situations where a web service registers with the UDDI, thus enabling the agents in the agent platform to look up the web service as it would with any other agent service. [16]

WSIG uses the jUDDI project from Apache Software Foundation[1].

[1] http://ws.apache.org/juddi/