"Alexandru Ioan Cuza" University
Faculty of Computer Science

Agent Developing Platform

Agent Communication

Author
Cătălin Hrițcu

Coordinator
Lect. Dr. Sabin-Corneliu Buraga

July 2005

Table of Contents

1. Preface

Many people have prophesized over the last ten years that software agents will bring the next revolution in Computer Science and that, this revolution will have dramatic consequences, not only on the way we build software, but also on our every day lives. It is needless to say that this revolution never happened, at least not yet. We can all see this, every time we turn on our computers and use the same type of non-intelligent programs we were using ten years ago. If there was a revolution in the last decade that is the revolution of the World Wide Web and it's related technologies and not that of software agents.

The faith of agent technologies and their unfulfilled prophecies, is somehow similar to that of Artificial Intelligence, and this is in part caused by the fact that agents are supposed to be intelligent. Some progress is surely being made, but there is apparently no way to tell whether the answer to many of the fundamental problems of software agents are just around the corner or a million miles away. Some of these real problems are well explained in a very critical but realistic paper [Nwana and Ndumu, 1999]: information discovery, communication, ontology, collaboration, reasoning, monitoring, legacy software integration. The conclusion the authors draw in the paper is that for agent systems to reach their full potential, developers must avoid premature formalization and start implementing actual agent systems. And this is the pragmatic goal we have in mind for the Agent Developing Framework (ADF) we will describe in this work.

Knowing that we won't be able to deal with more than of the enumerated problems, we have chosen what we consider to be the most basic one: agent communication. We find this also to be the simpler than the other, because almost all the pieces of the puzzle seem to be present somewhere, in one form or another, and only need to be put in place. For two agents to communicate there is the need for a common transport protocol, a common communication language and a common understanding of the terms in use (e.g. a common ontology). Web technologies seem to offer at this time solutions for two of the three issues, while FIPA deals with the remaining one acceptably. SOAP based web services are able to communicate over any transport so "SOAP over anything" will be our "transport protocol", and because SOAP over HTTP is already ubiquitous it makes a very good instance of that. The common communication language will be FIPA ACL encoded as XML, due to lack of other choices that would assure interoperability. Finally, RDF and OWL should solve, at least in part, the ontology problem, so RDF will be the our content language. Since all these are XML-based (or XML-capable) technologies, XML will be the ligand to make everything fit together. In order assure communication is working the way it should however, we will also need at least the basic architecture of the multiagent platform. And because there is a match between multiagent platforms and a loosely-coupled service-oriented architecture, this is the architectural design we will use in ADF.

No matter the fact that the agent revolution did not happen, there is still great research potential in agent systems, not to mention that they are very interesting software to work on. So, what we can expect is agent evolution, and evolution in general has the tendency to take a lot of time, and a lot of hard work in this case.

2. Introduction

2.1 Goal

The purpose of this thesis is to introduce the reader to ongoing research regarding software agents, and present the personal contributions of the author to the development of a new agent developing framework (ADF). These contributions are focused in the area of agent communication, but also in redefining the overall architecture of ADF in order to make it withstand the challenges of the future. We will investigate the architectural issues involved and present a reference implementation of the framework based on Java 2 Enterprise Edition. We will also examine in great detail the current technologies that would make such an effort possible, and also analyze what the near future has to offer.

The goal of the ADF project is to build a complete multiagent framework. However, this is a very complex task, which was started more than one year ago [Nichifor and Buraga, 2004] and will surely take more time and effort to bring to an end. In order to avoid getting lost in the process, we have tried to identify the most important tasks and focused our efforts there. Areas such as agent mobility and security are not yet thoroughly investigated and will be the subject of future research.

2.2 Structure

This work is organized into eight chapters:

- Chapter 1 is a personal view of the author on the agent world and its evolution.
- Chapter 3 provides an advanced introduction to software agents and their communication. Many different technologies are discussed and many references to other works are made. The chapter covers the FIPA abstract architecture for agent frameworks, the FIPA agent communication language, the different types of message-oriented communication with an emphasis on reliability and persistence. The chapter also introduces service-oriented architectures, SOAP-based web services, the Resource Description Framework and the Java 2 Platform, Enterprise Edition.
- Chapter 4 introduces the ADF architecture, starting with the general goals and ending with the actual implementation. The goals of the framework are stated and discussed at large: interoperability, extensibility, platform independence, scalability, distribution transparency, ease-of-use, security and pragmatism. The reminder of this chapter discusses the ADF general design and implementation.
- Chapter 5 covers agent communication in ADF. The chapter is broken into two sections. One that covers transport protocols and the other describing the agent communication language and its encodings.
- The last three chapters provide appendices (Chapter 6), the conclusion of this

work (Chapter 7) and the extensive bibliography (Chapter 8).

2.3 Source Code

The source code of the Agent Developing Framework (ADF), described in this thesis, can be freely obtained from http://adf.sourceforge.net/. The framework is free software; and can be redistributed and/or modified under the terms of the GNU Lesser General Public License, version 2.1 as published by the Free Software Foundation.

This framework is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. Please see [LGPL] for more details.

2.4 Acknowledgments

Many people contributed to the realization of this thesis, whether they are aware of this or not. I would especially like to thank my coordinator, Lect. Dr. Sabin-Corneliu Buraga, for his amazing classes in Web Technologies and his substantial support on this project. I also express my gratitude to the other professors of the Faculty of Computer Science in Iasi, for all the important things they taught me, not only about Computer Science. Additionally, I thank Ovidiu-Cătălin Nichifor, the original developer of ADF, for fulfilling the last duty every open source developer has, when he looses interest in a project, that is: handing it off to a competent successor. His help was most precious until I got accustomed to the software. And last, but not least, I express my appreciation to the open source community as a whole, and to the people behind the excellent open source tools I used intensively for this project: JBoss, Jade, Eclipse, XDoclet, Hermes, SourceForge, OpenOffice, uml2svg, Apache Tomcat, Forrest, log4j and many, many others.

3. The Basics

This chapter introduces the main concepts and technologies that will be used in the reminder of this work. Agent frameworks are extremely complex software systems, bringing together knowledge from a lot of different fields like Distributed Systems, Software Engineering, Artificial Intelligence, Human-Computer Interaction, Linguistics and many other. We will cover a lot of interesting topics, but since our interest is in agent communication we will be biased towards Distributed Systems and Software Engineering. Strap yourself in!

3.1 What Is a Software Agent?

According to [Jennings and Wooldridge, 1998], a **software agent** is an autonomous process capable of reacting to, and initiating changes in its environment possibly in collaboration with users and other agents. This definition is not the only one, and, in fact, there is some controversy concerning what an agent is [Franklin and Graesser, 1996]. For our purposes however, this definition is adequate, so we will explain it in some detail.

To be considered an agent, a software object must be **autonomous**, that means it must be capable of making independent decisions and taking actions to satisfy internal goals based upon its perceived environment. This also means that agents are able to sense the environment and timely respond to changes (**reactivity**) and can initiate actions on their own to affect this same environment (**proactivity**). Other properties of agents that are less common to but still very meaningful are the ability to migrate from one host to another (**mobile agents**) and the capability to adapt based on past experience (**learning agents**).

Another important aspect of agents is that they usually **cooperate** with other agents being part of a multiagent system. For example, collaborative agents could be used to arrange meetings [Kozierok and Maes, 1993] or attend auctions [Chavez and Maes, 1996] on behalf of their human owners. The agents we usually have in mind from now on are **collaborative agents** and the emphasis will be on their communication.

3.2 Agent Frameworks

A software framework is a set of cooperating classes that make up a reusable design for a specific class of software [Johnson and Foote, 1988]. By extension, an agent framework provides a foundation for building agent-oriented applications. So instead of doing low-level work, like building naming, location and directory services, inventing communication protocols, mobility mechanisms or cryptographic algorithms, developers can concentrate on their particular problems and on the logic of the agent-oriented applications solving them.

Agent frameworks have an important role in building large-scale distributed systems, so there have been many attempts to build such frameworks, in both the research and business communities. Some of them were successful and are still very actively maintained like [Cougaar], [DIET Agents], [Jade], [Voyager] or [ZEUS], while other didn't stand the test of time, but nevertheless had an important contribution to the software agents field. In this later category we can name [Aglets], [Ajanta], [D'Agents], [FIPA-OS] or [Omega].

3.3 The FIPA Abstract Agent Architecture

With all the different agent frameworks being built by different people and with different goals in mind, it was no surprise that they were not interoperable at first. As a starting point, the The Foundation for Intelligent Physical Agents [FIPA] developed a set of specifications that would allow heterogeneous agents to interoperate. While these specifications are far from perfect and failed to achieve widespread support in the agent world, they are closer to a standard than anything else is, so will discuss them in this and the next sections[1].

FIPA has defined an abstract reference model, which must be obeyed by any FIPA compliant framework in order to assure interoperability [FIPA00001]. The internal design and implementation of agents is not mandated by FIPA, so there is a broad set of possible concrete architectures, which will interoperate because they share the common abstract design (see Illustration 1).

1On 8 June 2005, after years of inactivity and decline, FIPA was officially accepted by the IEEE as its eleventh standards committee, and will be known as the FIPA Standards Committee. It has also changed its original goal to moving standards for agents and agent-based systems into the wider context of software development in general. So probably the FIPA approach was not entirely wrong, but surely ahead of its time, and with the benefits provided by the umbrella of a large standards organization like IEEE, FIPA could be reborn.

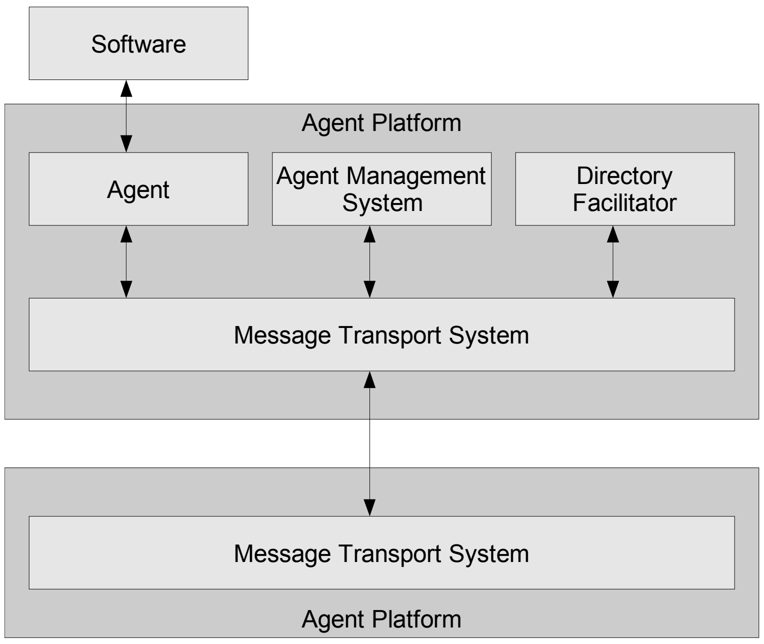


Illustration 1: The FIPA Abstract Agent Architecture Model

In this model the agent platform has to provide the basic services needed in any multiagent system. These facilities include those for creating, registering and deleting agents, facilities to locate agents and services, and last, but not least, facilities for inter-agent communication [FIPA00023]. Some areas that are not sufficiently abstract like agent lifecycle management, agent mobility or security related issues and are not part of the FIPA abstract architecture.

An agent management system (AMS) keeps track of the agents for the associated platform and provides services for agent creation, registration and deletion. It also provides a *naming service* by which a globally unique Agent Identifier (AID) is mapped to a local communication endpoint. The AID is an extensible collection of parameter-value pairs, which comprises at least:

- The *name* parameter, which is a globally unique identifier that can be used as a unique referring expression of the agent. One of the simplest mechanisms is to construct it from the actual name of the agent and its home agent platform (HAP) address, separated by the "@" character.
- The *addresses* parameter, which is a list of Uniform Resource Locators [RFC2396] where a message can be delivered.
- The *resolvers* parameter, which is a list of name resolution service agents.

Two AIDs are considered to be equivalent if their name parameters are the same. AIDs are primarily intended to be used to identify agents inside the envelope of a transport message, specifically within the sender and receivers parameters, so we will discuss more about them in the context of agent communication.

A Message Transport Service [FIPA00067] provides the default mechanism for FIPA agents to communicate by exchanging messages. In particular the Message Transport Service (MTS) is responsible for point-to-point communication with other platforms (see Illustration 2). When a message is sent it is first encoded using an encoding-representation appropriate for the transport, for example as a String [FIPA00070] or as an XML document [FIPA00071], and then included in a transport-message. The transport-message contains the encoded ACL message and an envelope including the sender and receiver transport-descriptions as depicted in Illustration 3. FIPA ACL will be presented in the next section, while agent communication is a main topic throughout this work.

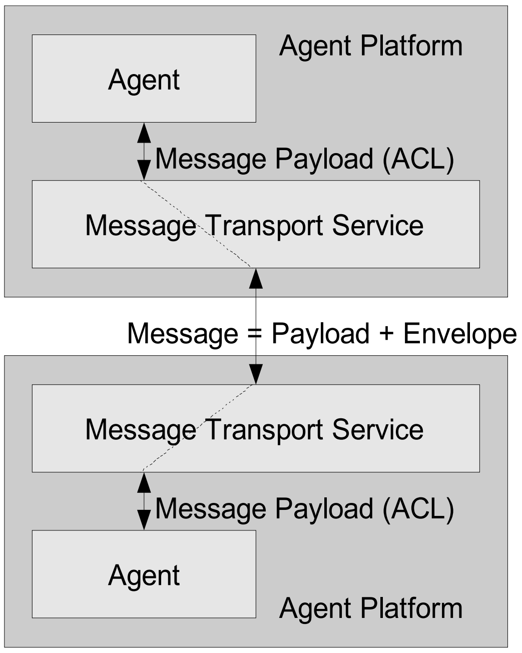
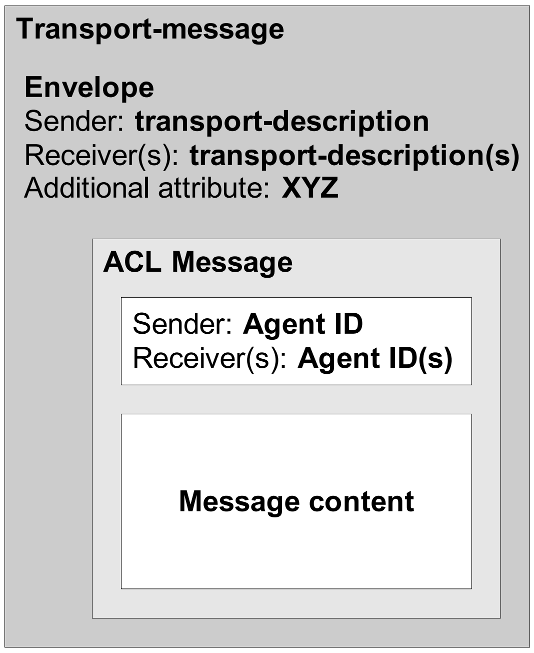


Illustration 2: FIPA Message Transport Model


Transport-message
Envelope
Sender: transport-description
Receiver(s): transport-description(s)
Additional attribute: XYZ
ACL Message
Sender: Agent ID
Receiver(s): Agent ID(s)
Message content
Illustration 3: The FIPA Transport-message

A Directory Facilitator (DF) is an optional component of the agent platform that provides *yellow pages* services to other agents. In addition to a number of standard services agents may register their services with the DF or search the DF to find out what services are offered by other agents. An agent providing a service is more constrained in its behavior than a general-purpose agent, loosing some of its autonomy. It could however refuse to provide the service if that violates its internal agenda.

3.4 The FIPA Agent Communication Language

One major difference between agent platforms and classic approaches to distributed systems, is that agents communicate by means of an application-level communication protocol, which is referred to, as an agent communication language. Popular agent communication languages are the FIPA ACL and the Knowledge Query and Manipulation Language [KQML].

Agent communication languages rely on the **speech act theory**, originally developed by

[Searle, 1969] and later enhanced by [Winograd and Flores, 1987]. In an ACL a strict separation is made between the *contents* of the of the message and its purpose, also known as the *performative* or the *communicative act*. The set of possible performatives is limited and their meaning is specified by the agent communication language and known by all the agents using the it. The content of the message is not standardized and varies from system to system, and the use of task-oriented ontologies is very common. To assure two agents understand each other they have to not only speak the same language, but also have a common ontology. An ontology is a part of the agent's knowledge base and describes what kind of things an agent can deal with and how they are related to each other.

Other than the special case of agents that operate alone and interact only with human users or other software interfaces, agents have to communicate with each other in order to perform the tasks for which they are responsible.

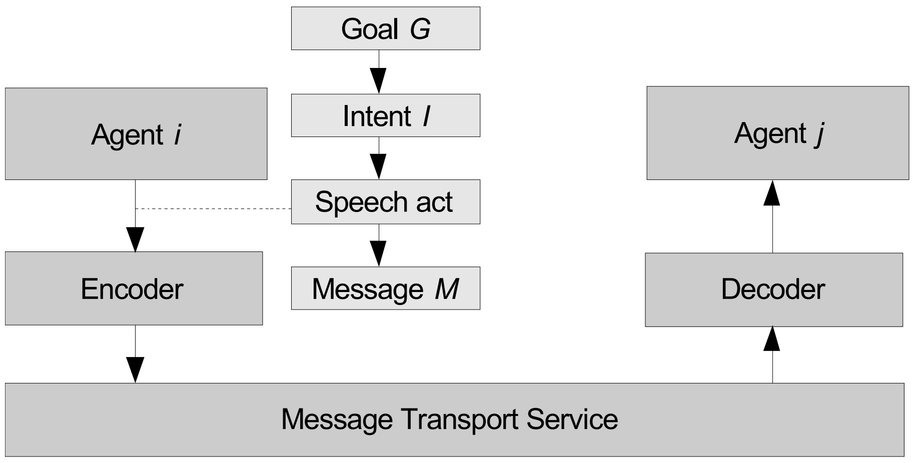
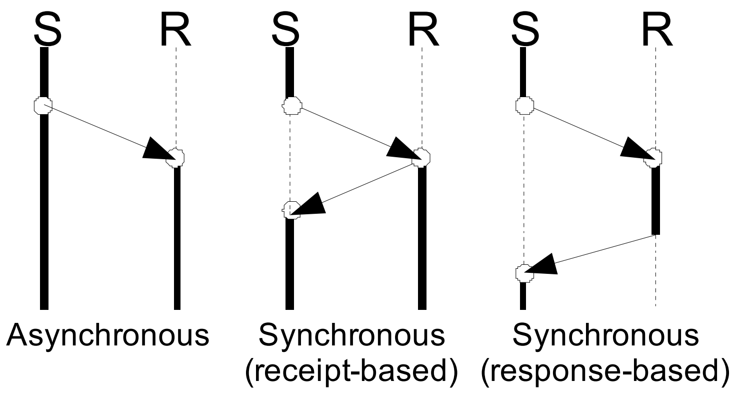


Illustration 4: Example of Agent Communication Using Speech Acts

Consider the situation depicted in Illustration 4, where agent *i* has amongst its *mental attitudes* some goal *G*. Deciding to satisfy *G* it adopts a specific intention, *I*. When agent *i* cannot carry out the intention by itself, the question becomes which message or set of messages should it send to another agent (*j* in Illustration 4) to cause intention *I* to be satisfied? If agent *i* is behaving *rationally*, it will send out a message whose effect is to attempt to satisfy the intention and hence achieve the goal.



For example, if a personal assistant agent *i* has to schedule a meeting at a certain time *T* between his owner *O* and one of his friends *F* (*G* = "arrange a meeting between *O* and *F*"). The agent can derive a sub-goal to find out whether *F* is available at time *T* (*G'* = "know if *F* is available at time *T*") and thus the agent intends to find out this information (*I* = "find out if *F* is available at time *T*"). Would it make any sense to ask the personal assistant agent of F, agent *j*: "Did *F* play a video game yesterday?". Well, no matter what the answer to this question would be it would not help agent *i* know whether *F* is available at time *T*. However, if agent *i* acts more rationally, he would ask agent j "Can you tell me if *F* is available at time *T*?", and thus act in a way it hopes it will satisfy his intention and meet his goal. agent *i* is thus assuming that agent *j* knows the answer and it will share the information with agent *i*. However, simply on the basis of having asked, agent *i* cannot assume that agent *j* will act to answer: agent *j* is independent and can have a different goals.

So, an agent plans to meet its goals by communicating with other agents. The agent will perform speech acts based on the relevance of their expected outcome or **rational effect** in relation to its goals. However, it cannot assume that the rational effect will inevitably result from sending the messages.

This communication model is at the heart of the FIPA model for agent systems. A message contains a set of one or more message parameters, out of which the only mandatory one is the *performative*, although it is expected that most ACL messages will also contain sender, receiver and content parameters. The complete list of parameters mandated by [FIPA00061] and is given in Appendix 8.1. Additional message parameters can be added by specific implementations as long as their name starts with "X-".

Returning to our example involving personal assistant agents, agent *i* can ask agent *j* if *F* is available at time *T* using the following ACL Message:

```
(query-if
     :sender (agent-identifier :name i)
     :receiver (set (agent-identitfier :name j))
     :content "((available (person F) (time T)))"
     :reply-with r09
     :language fipa-sl)
```

Agent *j* could reply that F is not available:

```
(inform
     :sender (agent-identifier :name j)
     :receiver (set (agent-identifier :name i))
     :content "((not (available (person F) (time T))))"
     :in-reply-to r09
     :language fipa-sl)
```

The two ACL messages above are encoded as strings [FIPA00070] and their contents confirms to the FIPA Semantic Language [FIPA00008]. The other content description language experimentally supported by FIPA are [KIF], [CCL] and, what we find more interesting [RDF] (RDF will be discussed in a following section). Although the

[FIPA00011] specification is still experimental it proves that it is meaningful to use RDF as a content language. We will examine this possibility in the next chapter. Here we will only give the answer of agent *i* expressed in RDF, using an XML encoding [FIPA00071] and a SOAP envelope. Note that the exact form of the SOAP envelope is not (yet) standardized.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
      <!-- non-standard header -->
      <soap:Header>
            <fipa:from xmlns:fipa="http://www.fipa.org"
                  soap:role="http://www.w3.org/2003/05/soap-envelope/role/next"
                  soap:mustUnderstand="false">http://host.of.i.com</fipa:from>
            <fipa:receiver xmlns:fipa="http://www.fipa.org"
                  soap:role="http://www.w3.org/2003/05/soap-envelope/role/next"
                  soap:mustUnderstand="false">http://host.of.j.com</fipa:receiver>
            <fipa:acl-representation xmlns:fipa="http://www.fipa.org"
                  soap:role="http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver"
                  >xml</fipa:acl-representation>
      <!-- other infromation may follow -->
      </soap:Header>
      <soap:Body>
            <fipa-message act="query-if">
                  <sender>
                        <agent-identifier>
                              <name id="i"/>
                        </agent-identifier>
                  </sender>
                  <receiver>
                        <agent-identifier>
                              <name id="j"/>
                        </agent-identifier>
                  </receiver>
                  <content>
                        <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
                              <ex:Person xmlns:ex="http://www.example.com">
                                    <ex:id>http://www.example.com/persons/F</ex:id>
                                    <ex:available-at>T</ex:available-at>
                              </ex:Person>
                        </rdf:RDF>
                  </content>
                  <reply-with>r09</reply-with>
                  <language>rdf</language>
            </fipa-message>
      </soap:Body>
</soap:Envelope>
```

All the FIPA ACL communicative acts and their exact semantic definitions are standardized by FIPA. They are are listed in Appendix 8.2. and summarized in Table 1. Moreover FIPA has standardized many common used interaction protocols using communicative acts, which we enlist in Appendix 8.3.

Performative	***Description***
accept-proposal	The sender accepts a previously submitted proposal to perform an action.
agree	The sender agrees to perform some action, possibly in the future.
cancel	Inform the receiver that the sender no longer has the intention that the receiver performs a previously requested action.
cfp	The action of calling for proposals to perform a given action.
inform	The sender informs the receiver that a given proposition is true.
not-understood	The sender did not understand a message previously received from the receiver.
query-if	The sender asks the receiver whether a given proposition is true.
refuse	The sender refuses to perform a given action, and explains the reason for the refusal.
reject-proposal	The sender informs the receiver that it has no intention that the recipient performs the given action under the given preconditions.
request	The sender requests the receiver to perform some action.

Table 1: Examples of Frequently Used FIPA ACL Performatives

In order to exemplify the way different communicative acts are used to create to complex interaction protocols we will further examine the FIPA Contract Net interaction protocol [FIPA00029]. Illustration 5 provides a graphical representation of the steps involved in the normal operation of this protocol. For simplicity, issues like error handling were omitted.

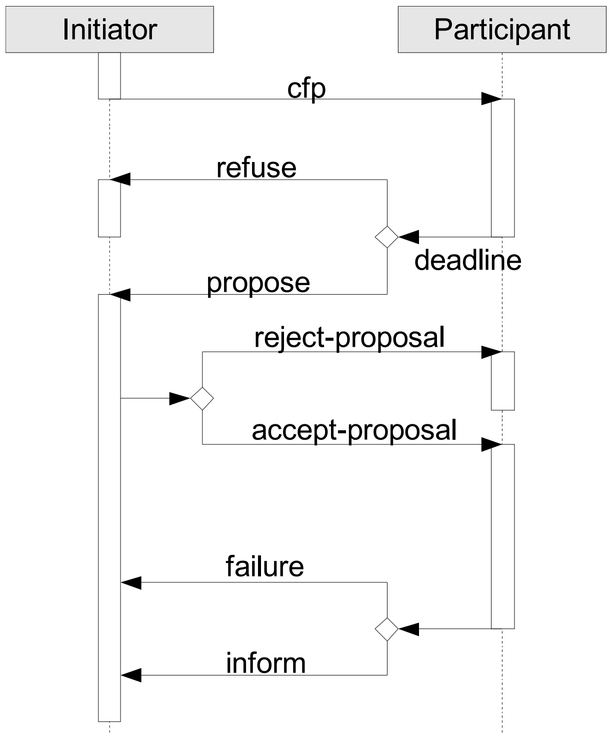


Illustration 5: FIPA Contract Net Interaction Protocol

The initiator of the FIPA Contract Net protocol requests proposals from other agents by issuing a *cfp* act (call for proposals). This message specifies the task, as well as any other supplementary conditions such as the maximum price or duration the it is placing upon the execution of the task etc., and a deadline by which replies should be received. The deadline helps prevent the case when the initiator would have to wait indefinitely for all the answers to arrive, so proposals received after the deadline are automatically rejected. The participants receiving the call for proposals are viewed as potential contractors and they may generate responses. Some of them will be proposals to perform the task, specified as *propos*e acts which includes the preconditions that the participant is setting out for the task, while other participants may *refuse* to make a proposal.

Once the deadline passes, the initiator evaluates the received proposals and selects the agents to perform the task (one, several or no agents may be chosen). The accepted agents will be sent an *accept-proposal* act and the remaining ones will receive a *reject-proposal* act. Once the initiator accepts the proposal, the participant has the obligation to perform the task and when it is has it, the participant sends an *inform* message to the initiator. However, if the participant fails to complete the task, a *failure* message is sent.

3.5 Message-Oriented Communication

Several abstractions can be provided over the interface of the transport layer in a computer network, and a quite popular one is message-oriented communication. This allows messages to be sent back and forth between two or more network hosts. The services provided by the communication system can range from almost nothing (e.g. plain UDP) to persistent and reliable communication. After a short digression on synchrony we will examine communication persistence and reliability.

Synchronous vs. Asynchronous

Asynchronous communication is very simple: the sender continues to execute immediately after it has submitted it's message for transmission.

Synchronous communication is a little more complex; it actually comes in two "flavors":

- receipt-based: the sender is blocked until the message has reached the destination and an acknowledgment has returned;
- response-based: the sender is blocked until the receiver has processed the message and the result of the processing has returned to sender (see Illustration 6).

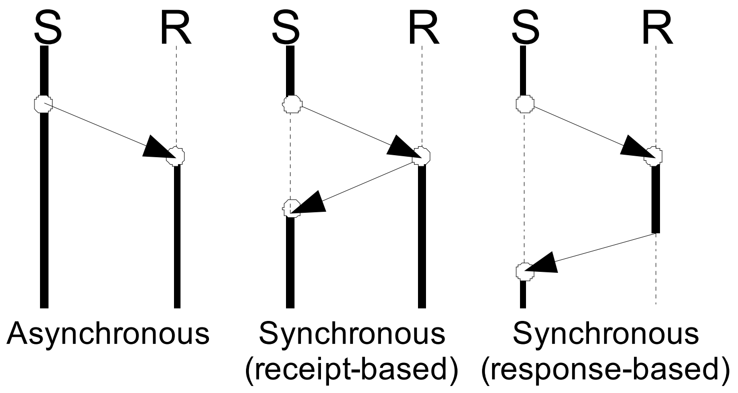


Illustration 6: Asynchronous vs. Synchronous Communication

Persistent vs. Transient

Almost all transport protocols are *transient*, that is: messages are stored by the communication system only for as long both sending and receiving applications are executing. For example the Internet Protocol offers transient communication only. If any router cannot find an appropriate destination to forward the packet immediately it will simply discard it.

When *persistent communication* is used, the transmitted messages are stored by the communication system as long as it takes in order to deliver them to the receiver. Neither the sender, nor the receiver have to be up and running for message transmission to occur. Persistence increases the reliability of the communication. If for any reason the receiver fails, the senders can continue to send messaged that will simply accumulate in a message store and will be processed when the receiver comes back on-line. This is particularly useful when dealing with unreliable networks or limited connectivity, very common for wireless networks. A good example of a persistent communication protocol is SMTP, the protocol used for electronic mail on the Internet [RFC2821]. Electronic mail is however different from the other persistent communication systems we will present, by the fact that it is primarily aimed at end users, and not applications.

More elaborated systems that provide persistent asynchronous communication are typically called Message-Queuing (MQ) Systems or Message-Oriented Middleware (MOM); such as the old IBM MQSeries (rebranded now [WebsphereMQ]). There are many other such systems built by various vendors or by the open source community. These systems are principally used for enterprise application integration (EAI), but they can also be used for electronic mail, workflow, groupware and many other tasks.

In a message-queuing system, applications communicate by inserting messages in specific queues owned by one or more applications. Messages can only be sent to (or read from) queues that are local to the sender. Each message is forwarded over a series of communication serves until it reaches its destination queue, identified by a unique queue identifier. The queue owner can then read the message at any given time.

The principal problem with message-oriented middleware the lack of interoperability due to the lack of standards. All the major vendors have their own implementations, each with its own API and proprietary management tools. J2EE provides a standard API for accessing MOM systems, called Java Message Services (JMS) that most vendors have implemented. However, this solution is not available to users of other programming languages. Since J2EE is the framework used by ADF we we will further examine JMS in Section 3.9.

Reliable vs. Unreliable

Unreliable communication is usual to best-effort protocols like the Internet Protocol. As the term best-effort implies, no guarantees whatsoever are given that a message (package in the case of IP) will actually make it's way the destination, or it will not arrive more than once, or that multiple messages will reach the destination in exactly the order they were sent.

Reliable communication provides this kind of quality-of-service guarantees, under one of the following three delivery semantics: At-Least-Once, At-Most-Once and Exactly-Once. If the communication system is not capable of delivering a message under the given constraints, it will notify the sender of the failure. One additional reliability provision is In-Order delivery. Stream-oriented protocols like TCP offer Exactly-Once and In-Oder delivery, while message-oriented transient protocols usually don't. SMTP [RFC2821], although persistent does not offer reliable end-to-end message delivery (it can, however, provide acknowledgments for successfully delivered messages). If an email router disastrously fails (e.g. hard drive failure) then all the ongoing messages it stored, will be irremediably lost, and the senders will not be informed in any way. This means that persistence does not automatically imply reliable communication.

Reliable asynchronous messaging is a key building block for service-oriented architectures (the subject of our next section) and for multiagent systems alike. If an agent needs a small piece of data from another agent then it might be acceptable for the first agent to just wait for a message. But what happens when the message gets lost or is delayed for any reason? Should the sender keep waiting forever? Should it assume that his request got lost and just submit another? Should it assume that the response got lost and wait for the other agent to retransmit it? There are no easy answer to these questions; other than using reliable asynchronous messaging.

HTTPR was an early attempt by IBM to provide a protocol on top of HTTP for the reliable transport of messages over the Internet [HTTPR]. Unfortunately, IBM was unable to rally sufficient industry agreement around HTTPR. The two current divergent directions for reliable messaging are [WS-ReliableMessaging] (a protocol supported by IBM, BEA and Microsoft) and [WS-Reliability] (an OASIS standard). And while both specifications address the same issues, they do so in different, incompatible ways. This situation helps no one and, unless a compromise is reached, it is very likely to continue.

3.6 Service-Oriented Architectures

A Service-Oriented Architecture (SOA) is an architectural style whose goal is to achieve **loose coupling** among interacting software agents. A service is a unit of work done by a service provider, to achieve desired results for a service consumer. A flexible mechanism permits a consumer to discover the providers that offer the services it needs. Both provider and consumer are roles played by software agents on behalf of their owners.

Loose coupling is obtained in a SOA by defining a small set of *simple generic interfaces* that are universally available to the participating software agents. The interaction between them is done via *descriptive messages* exchanged through the standard interfaces and constrained by an extensible schema, thus allowing new versions of services to be introduced without breaking existing services. This approach reduces artificial dependencies between the interacting components and is different from object oriented programming, which suggests that data and its processing should be bound together [He, 2003].

Many multiagent systems are built as Service-Oriented Architectures. Agents consume the services provided by other agents in order to be able to provide their own specialized services. This is meaningful because it allows every agent to specialize only one or several tasks it does very well, while delegating the other tasks to other expert agents. Most of us are smart enough to realize that we are not smart enough to be expert in everything. The same principle applies to software engineering where it is called *separation of concerns*.

Since large-scale distributed systems tend to be extremely heterogeneous, there are very few generic interfaces universally available, so the application-specific semantics must be expressed in the messages themselves. There are three fundamental properties the messages transmitted over the interfaces of a SOA must have: *descriptive*, *restricted* and *extensible*. The messages must be descriptive because the service provider is entirely responsible for solving the problem, so choosing the way it achieves this is solely its concern. Limiting the vocabulary and structure of messages is also a necessity for interoperability in any distributed system. The more restricted a message is, the easier it is for the receiver to understand the message. However, this comes at the expense of reduced extensibility, which is very important if the architectures are to evolve to meet their ever changing requirements. If messages are not extensible, consumers and providers will both be locked into one particular version of a service, so extensibility is not just a good design practice for a SOA but a fundamental necessity. Software designers must weigh the trade-off between interoperability and extensibility and come up with the just the right balance.

There are additional constraints that can be applied on a SOA in order to improve it's performance, scalability and reliability. The most important are *idempotent requests* and *stateless service*. Requests are idempotent if their duplication by a software agent has the same effects as a unique request. This allows providers and consumers to improve the overall service reliability by simply repeating the request if faults are encountered. A service is stateless when each message sent by a consumer contains all the necessary information for the provider to process it. This constraint makes the service provider more scalable because the provider does not have to store conversational state between different requests. There are no intermediate states to worry about, so recovery from partial failure is also relatively easy, making the service more reliable.

However, not all services can or should be made stateless. The other possibility is to establish a session between the consumer and the provider, and the compelling reason to do so is efficiency. For example, the overhead produced by having to exchange security certificates and do authentication, can be limited to the establishment of the session, which greatly improves performance. Also services that allow customization are very good candidates for being stateful. Stateful services require both the consumer and the provider to share the same consumer-specific context, which is either included in or referenced by messages exchanged between the provider and the consumer. The drawback of this approach is that it may reduce the overall scalability of the service provider, because it now needs to remember the shared context for each consumer. It also increases the coupling between a service provider and a consumer, and makes switching service providers more difficult. Still, in many cases there is not simple way to avoid this.

Because some people fail to notice the difference between SOAs and traditional distributed objects systems (like DCOM, Java-RMI or CORBA) we will mention it here one more time: unlike traditional object-oriented architectures, SOAs comprise loosely coupled, highly interoperable services that asynchronously exchange descriptive self-contained messages. Table 2 should make it clear where the differences are:

	Tightly Coupled	***Loosely Coupled***
Connection	Direct Connection	Message Broker
Communication	Synchronous	Asynchronous
Interaction Type	Remote Procedure Call	Message Passing
Type system	Strong Typed	Weak Typed
Control	Centralized	Distributed
Service discovery and binding	Static	Dynamic

Table 2: Differences Between Tightly Coupled and Loosely Coupled Architectures

We will further examine SOAs in the next sections, in the context of web services and message oriented middleware, and then in Chapter 4, when we present the ADF architecture. However, for a more in depth introduction to SOAs and their use in enterprise applications we refer the readers to [Krafzig et al., 2004] and [Chappell, 2004].

3.7 Web Services and SOAP

According to the Web Consortium [W3C], a Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP [RFC2616] with an XML serialization in conjunction with other Web-related standards [WSA]. This definition should give the reader a hint about what web services really are: a vague term describing a collection of protocols and open standards for exchanging data between software applications written in various programming languages and running on various platforms. [OASIS], the W3C and the [WS-I] are the steering organizations responsible for standardization of web services.

The basic technology behind web services is [SOAP] and its relation to the other Web standards is depicted in Illustration 7. SOAP provides a simple framework for exchanging XML messages between an initial SOAP sender and an ultimate SOAP receiver [SOAP Primer, 2003].

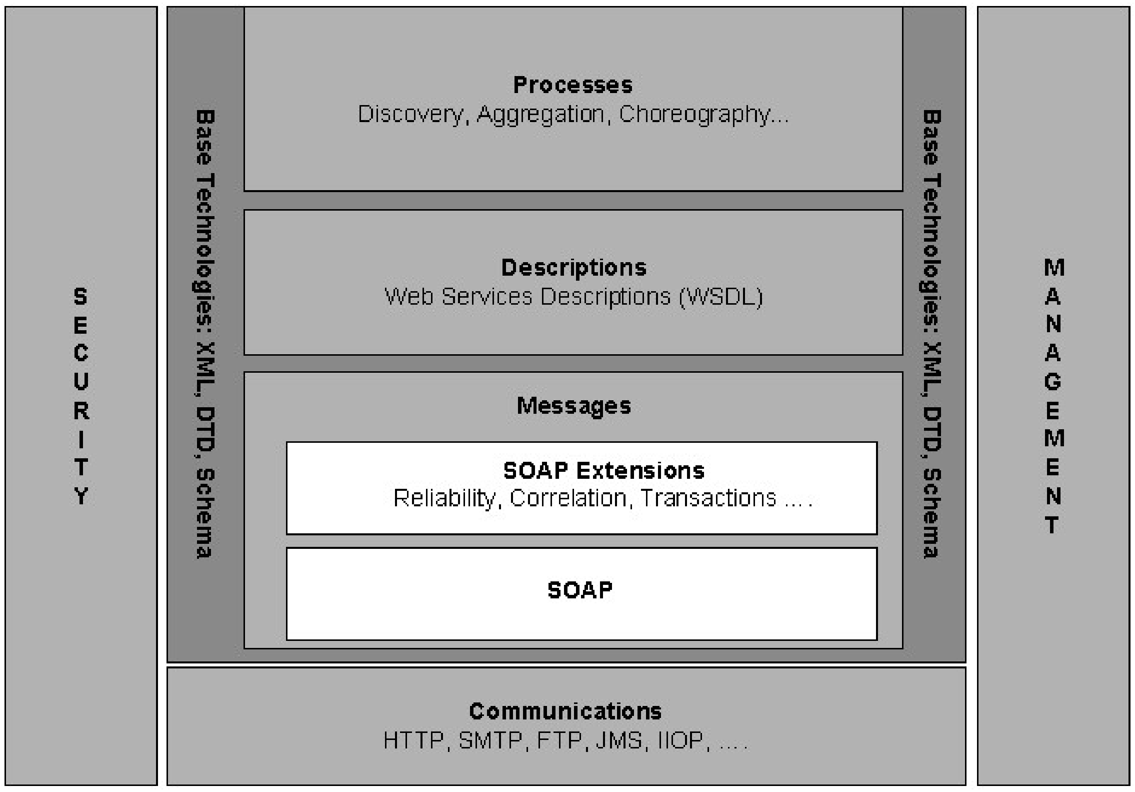

S E C U R I T Y
Base Technologies: XML, DTD, Schema
Processes
Discovery, Aggregation, Choreography...
Descriptions
Web Services Descriptions (WSDL)
Messages
SOAP Extensions
Reliability, Correlation, Transactions ....
SOAP
Base Technologies: XML, DTD, Schema
M A N A G E M E N T
Communications
HTTP, SMTP, FTP, JMS, IIOP, ...

Illustration 7: Web Services Architecture Stack

The simplest such exchange is a request-response pattern, which use was overemphasized by earlier SOAP versions as the means for conveying remote procedure calls (RPC). However, it is important to note that SOAP request-response exchanges do not have to to be modeled as RPCs. In fact SOAP RPC web services are not service oriented architectures, while document-centric SOAP web services are. This is because SOAP RPC "tunnels" application-specific RPC interfaces though an underlying generic interface. Effectively, it prescribes both system behaviors and application semantics. Because system behaviors are very difficult to prescribe in a distributed environment, applications created with SOAP RPC are not interoperable by nature. RPC also tends to be instructive rather than descriptive, which is against the spirit of SOAs. SOAP allows, however, for much richer conversational patterns, where the semantics are at the level of the sending and receiving applications. This is in fact a very important requirement of agent communication.

A SOAP message is an XML document with a simple structure. The outermost element is the SOAP envelope which contains an optional SOAP header and a mandatory SOAP body. The SOAP header holds a collection of SOAP header blocks, which can be targeted at any SOAP receiver within the SOAP message path. The header contains relevant information about the message, for example authentication information, encryption method or transactional context. The SOAP body contains the information explicitly targeted to an ultimate SOAP receiver in the form of XML data, or, in the case of error, a SOAP fault. A very simple SOAP message is given below (a more complex one was given in the previous section as an example).

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
      <env:Header>
            <t:transaction xmlns:t="http://travel.example.org/transaction"
                  env:encodingStyle="http://example.com/encoding"
                  env:mustUnderstand="true">673566</t:transaction>
      </env:Header>
      <env:Body>
            <m:chargeReservationResponse xmlns:m="http://travelcompany.example.org/"
                  env:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
                  <m:code>FT35ZBQ</m:code>
                  <m:viewAt>http://travel.example.org/reservation?code=069045</m:viewAt>
            </m:chargeReservationResponse>
      </env:Body>
</env:Envelope>
```

SOAP messages can be carried by a variety of network protocols such as HTTP, SMTP, FTP, RMI/IIOP, or a proprietary messaging protocol. However, the only normative binding for SOAP 1.2 messages is to HTTP 1.1 [RFC2616]. So, all the other choices for the transfer of SOAP messages are possible, but they are not standardized at this time. We will examine a JMS binding in chapter 5.

One major design goal for SOAP is simplicity and this is achieved by omitting, from the messaging framework, features that are very important in distributed systems, such as: reliability, security, correlation, transactions, routing, message exchange patterns and others. The good news it that the other design goal for SOAP is extensibility, so it is anticipated that these features will be defined as extensions to SOAP.

It is expected that the second-generation "WS-*" web services standards will fix these issues and allow web services to become a full-featured, loosely-coupled service-oriented architectures [Kaye, 2003]. New standards like [WS-Reliability]/[WS-ReliableMessaging], [WS-Coordination], [WS-Security], [WS-Transaction] or [WSBPEL], are very promising for business, but also for agent communication, and surely need further investigation [Erl, 2004]. However, until these standards gain enough community support and compliant implementations they are not very useful in practice.

Finally, a non-formal yet widely-used architectural style for building large-scale networked applications, that rivals with SOAP-based web services, is REST [Fielding, 2000]. REST defines identifiable resources via URIs, and methods for accessing and manipulating the state of those resources by means of standard HTTP. REST proponents argue that HTTP itself is sufficiently general to model any application domain. REST is an architectural style without a concrete specification, and while the pragmatic approach surely has its merits, we fear that it is not going to meet the need for interoperability of our agent framework. We will not examine REST in more detail, but for further information please consult the [RestWiki].

3.8 Java 2 Platform, Enterprise Edition

The Java 2 Platform, Enterprise Edition (J2EE) is an environment for developing and deploying enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multi-tier applications. J2EE multi-tiered applications are generally considered three-tiered because they are distributed over three locations: client machines, the J2EE server machine, and the enterprise information server (Illustration 8).

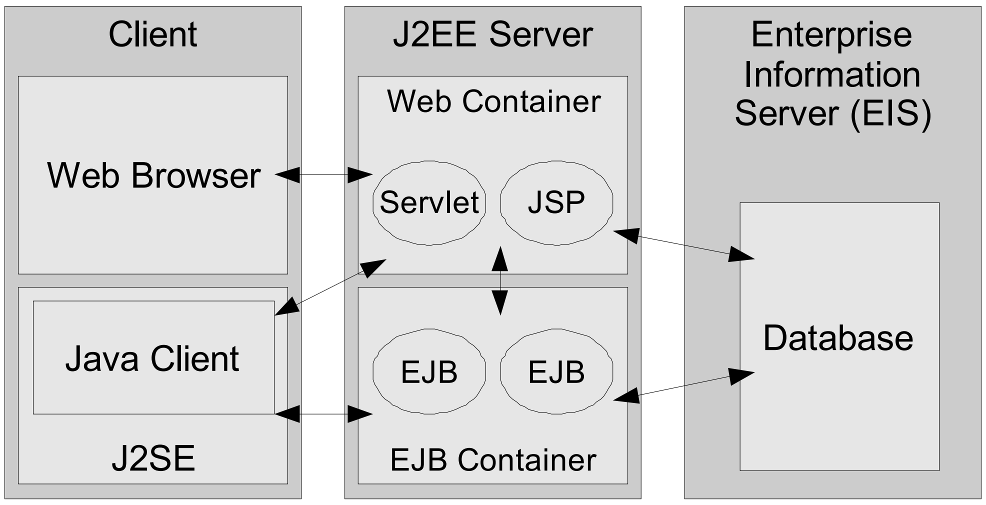


Illustration 8: J2EE Multi-Tiered Applications

J2EE applications are made up of self-contained components like:

- *Java Servlets* - web components that run in a Web container.
- *Enterprise JavaBeans* (EJB) - business components that run in an EJB container.

J2EE components are composed of compiled Java classes, deployment descriptors (XML files containing deployment information) and other resources, all archived into a single ZIP file (the extension is actually JAR or WAR). They are run and managed by a J2EE server such as the Sun Java System Application Server, the IBM WebSphere Application Server or the open source JBoss Application Server [JBoss]. An application server has to provide both an EJB container and a Web container such as Apache Tomcat [Tomcat].

Servlets

A Java servlet is a Java class that is designed to respond with dynamic content to client requests over a network. The generated content is commonly HTML and is served over HTTP, but servlets are more general than this, and XML data is also a popular choice.

Creating servlets is simple and requires just extending a class. Other technologies, such as *JavaServer Pages*, are compiled into servlets before actual use.

Enterprise JavaBeans

Enterprise JavaBeans are components that implement business logic. The EJB provides a standard distributed-component model that simplifies development and allows beans to be easily deployed on any J2EE compliant application server.

There are three types of enterprise beans: session beans, entity beans, and message-driven beans. Entity beans represent raws in a database table and are probably the most controversial J2EE technology. We will not use these type of beans so our discussion about entity beans ends here. Session and message-driven beans are presented in the next subsections.

Good reference books on enterprise beans in general are [Burke et al., 2004], [Matena et al., 2003], [Marinescu, 2002], [Sullins and Whipple, 2003]. However, many people find EJBs too old and heavyweight and propose a more lightweight approach [Gehtland and Tate, 2004], [Johnson and Hoeller, 2004] and [Walls and Breidenbach, 2005]. We will try to avoid the common pitfalls of EJB but use them nevertheless, unless until a more lightweight standard component technology takes it's place. A very good candidate for this is the upcoming EJB 3.0 specification that has the stated goal of improving the EJB architecture by reducing its complexity from the developer's point of view [EJB3].

Session Beans

Session beans are enterprise components that describe interactions between enterprise beans (taskflow) and implement particular tasks, thus shielding the client from the complexity of the business logic. Session beans are not persistent and come in two flavors: stateless and stateful.

Stateless session beans are lightweight and very efficient so a few stateless session bean instances can serve hundreds and possibly thousands of clients. Because they don't maintain any conversational state for a particular client, the EJB container can create multiple equivalent instances of a stateless session bean class, and choose any of them to service a particular method invocation. Stateless session bean instances can be simply discarded when they are not needed any more, or when the server is low on memory, thus removing the need for a passivation mechanism. Finally, stateless session beans are the only type of enterprise beans that can easily implement RPC-based web services.

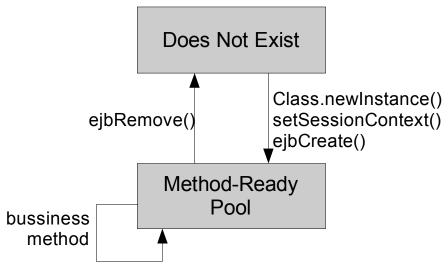


Illustration 9: The Life Cycle of a Stateless Session Bean

Stateful session beans are dedicated to only one client for the life of the bean's instance; as they maintain conversational state between requests. They are not swapped among clients or kept in an instance pool like and stateless session bean instances. Once a stateful session bean is instantiated and assigned to an EJB object, it is dedicated to that client for its entire life cycle (Illustration 11). When the EJB container needs to conserve resources, it can serialize the conversational state to secondary storage and evict stateful session beans from memory. When the client of the now passivated instance invokes a method, a new bean instance is created and populated with the state from the initial bean. This mechanism is called *passivation* respectively *activation*, depending on the viewer's perspective.

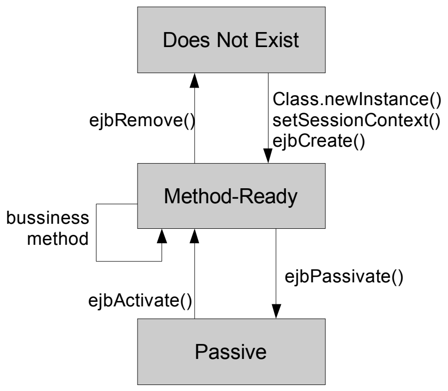


Illustration 10: The Life Cycle of a Stateful Session Bean

Java Message Service

The Java Message Service (JMS) is a standard API that provides access to a message queuing system (systems for persistent asynchronous communication were already examined in Section 3.6). JMS clients send messages asynchronously to a destination (topic or queue), from which other JMS clients can receive them at any time. Receiving messages is however blocking (a timer can be used to prevent deadlocks) for all JMS clients except for Message Driven Beans.

JMS supports both the point-to-point and the publish/subscribe messaging models, with a single, generic API. In the point-to-point model, each message has only one consumer and there are no timing dependencies between sender and receiver. In the publish/subscribe model each message can have multiple receivers, and a client can consume only messages published after he has created a subscription.

The principal advantage of JMS is that it provides asynchronous and persistent communication and thus and thus enforces loose-coupling between the communicating components. The major disadvantage is that it is hard to use outside an administrative domain because the underlying protocols are usually proprietary. For more information about JMS please consult [Haase 2002] or [Monson-Haefel and Chappell, 2001].

Message-Driven Beans

A *message-driven bean* is an enterprise bean that processes messages asynchronously. Historically, message-driven beans have been based on the JMS technology, but now other messaging technologies can be used as well. JMS-based message-driven beans are assigned during deployment to the queue or topic that they will process. When a message arrives to a particular destination, the container calls the corresponding message-driven bean's *onMessage* method that processes the message.

Like the stateless session beans examined earlier message driven beans are do not retain data or conversational state specific to a client. And because all instances of a message-driven bean are equivalent, the EJB container can instantiate as many beans as it finds necessary, in order to allow messages to be processed concurrently.

Java Naming and Directory Interface

As its name implies, the *Java Naming and Directory Interface* (JNDI) provides naming and directory functionality. Because JNDI is independent of any specific implementation, applications can use it to access multiple naming and directory services, including LDAP, NDS, DNS, and NIS. For more information on JNDI, please consult the [JNDI Tutorial].

4. The ADF Architecture

This chapter will present the architecture of the Agent Developing Framework (ADF). We start by introducing the main goals of the framework, and then we present the overall design, the technologies and the methodology that will allow us to achieve them. A lot more about the ADF architecture can be found in this specialized thesis [Nichifor, 2004], and we will not repeat here the aspects which we think were addressed there properly.

4.1 Goals

In this section we present in considerable detail the goals that should be met in order to make building multiagent systems worth the effort. Several goals were originally set for ADF by its initial developer and are given in this technical report [Nichifor and Buraga, 2004]. We will refine them here, and present several new goals, not mentioned in the above-mentioned technical report, but which are nevertheless very meaningful and in the spirit of the original ADF. We will also make a clearer distinction between the goals of the project and the general ways to achieve them. Please note, that the goals given here are set for the agent framework as a whole and not just for the agent communication part, implemented by the author and the main subject of this thesis.

Interoperability

Interoperability is a goal of extreme importance for ADF. It characterizes the extent by which ADF agents will be able to communicate and interact with entities in other systems, whether they are agents themselves, or more traditional applications, even with those that were not foreseen during the original development. In our view, using *open standards* is the only possible way to achieve true interoperability in a large-scale, thus heterogeneous, environment. However, the fact that a specification is public is often not enough to ensure interoperability, it should be also complete and neutral [Blair and Stefani, 1998]. Complete specifications make it unlikely for implementers to have the need to add implementation-specific extensions, that break interoperability. Neutral specifications do not prescribe how their implementations should look like and are not biased towards the technologies of a particular vendor.

Extensibility

The ability to easily add new features, or reimplement existing ones using different technologies, without impacting the operation of the system as a whole, makes a system extensible. In a rapidly evolving domain such as computer science, a system that is not extensible will become obsolete very quickly, in the extreme case even before it is released. When designing a software system, evolution should be regarded as something inevitable, that has to be planed for in advance (design to accommodate change).

Modular design, programming to standard interfaces and not implementations, separating policy from mechanism, supporting technology heterogeneity, orthogonality and loose-coupling between components are the most important ways to achieve extensibility.

Platform Independence

Platform independence is extremely desirable in large-scale thus highly-heterogeneous environments, such as the Internet. Agent systems have to accommodate all kinds of platforms, so if the same code can be executed with no regard to the platform, a great amount of developing time is saved. Our agent framework should not be dependent on a particular hardware configuration, operating system or an application server, and all the dependencies to third-party components should be modeled through standard interfaces.

Scalability

The scalability of a system can be measured on three dimensions [Neuman, 1994]: size, geographical and administrative. Scalability in respect to size is probably more familiar to the reader. For a multiagent system, scalability in respect to size is measured by the number of agents the system can accommodate while maintaining an acceptable performance. Centralization is "the worst enemy" of size scalability, whether we refer to centralized data, services or algorithms.

A multiagent system is geographically scalable if the agents may be very far from one another, and still they are able to interact efficiently. Geographical scalability is hindered too by centralization, but also by synchronous communication because of the very long waiting times involved.

Finally, a multiagent system is administratively scalable, if it can accommodate many different organizations that exert control over pieces of the system. The major problems involved here are management, payment, security and trust.

Several solutions to the scalability problem that can be applied to our agent framework are:

- *Distribution,* as opposed to centralization, suggests dividing a large problem into smaller parts that are to be solved by autonomous specialized agents, distributed all around the network.
- *Replication* is meaningful when an agent offers a service that is used by many other agents, thus making the first agent a communication bottleneck. If replicating the agent is possible without affecting the semantics of the service it provides (replication transparency) then the size scalability problem is solved.
- *Peer-to-peer* is an network topology that eliminates centralization, by removing the distinction between servers and clients. Peer-to-peer fits very well with the agent paradigm where agents are usually all treated as equals, but also for the underlying network connecting the agent hosts in a decentralized way.
- *Asynchronous communication* - minimizes waiting times when communication

delay is considerable. Asynchronous communication has also many other advantages not related to scalability, the most important one is the fact that is enforces lose-coupling.

- *Agent mobility* - can decrease the overall network load and the communication delays by localizing computation. Other good reasons for having mobile agents are given in [Lange and Oshima, 1999]: the ability to adapt dynamically, protocol encapsulation, natural heterogeneity, robustness and fault tolerance.

Transparency

Multiagent systems are inherently distributed across multiple networked machines. An important goal is being able to hide from the agents the fact that resources are distributed, and present the system as if it was running on a single machine [Tanenbaum and van Steen, 2002]. This concept of distribution transparency can be applied to many different aspects of a multiagent system:

- *Location transparency* refers to the fact that agents cannot easily tell where a resource is physically located in the system. Using logical names such as URIs [RFC2396] or other globally unique identifiers to reference resources while providing a naming service that will automatically convert these names into the corresponding addresses is usually enough in order to provide location transparency.
- *Concurrency transparency* would allow agents to share resources (e.g. processor time, memory space, communication channels, high-level services and other) in a cooperative way without noticing that the resources are, in fact, being shared. Resource replication (where possible), locking mechanisms and transactions are some of the ways to achieve concurrency transparency in an multiagent system.
- *Persistence transparency* deals with masking weather an agent is loaded in main memory (active) or written on disk (persisted). Persisting agents is a very important operation if scalability is to be achieved. When processing resources are scarce and there are simply too many agents to allow the whole system to function at normal throughput (or to function at all), it is meaningful to have only some of the agents active at any given time, while the others are persisted. Persistence transparency, means that the agents will be unaware whether they are being persisted or not. This kind of transparency is important, because it allows one to build a simple programming model for the agents, while not sacrificing scalability.
- *Failure transparency* means that the agents should not notice that a failure occurred and the system recovered from it, and the system as a whole should continue to operate even in the presence of failures. Perfect failure transparency is very hard to achieve, even impossible under realistic assumptions. One reason for this, is that it is impossible to tell for example if a resource is unavailable because of a failure, or it is slow, or it is simply overwhelmed with requests. However, in loosely-coupled systems, where communication is asynchronous and reliable, failures are much easier to mask than in traditional, highly-coupled

systems.

- A multiagent system in which mobile agents can migrate from one host to another, in a way that is transparent to the other agents, is called *migration transparency*. This would allow for example an agent to migrate without affecting its ongoing communication with other agents.
- Even stronger than migration transparency, and thus harder to achieve, is *relocation transparency*. Relocation transparency would allow a mobile agent to be migrated from one platform to another, without the agent itself being able to notice the that it is being migrated. This would allow load balancing to occur, without having to deal with it in any way when programming the agents.

Please note that even though aiming for distribution transparency is a good goal when designing a multiagent system, this issue should be considered only together with other issues, such as performance and scalability. There are also situations when hiding all the distribution aspects is nearly impossible to achieve (e.g. relocation transparency or failure transparency) or not a good idea at all. For example, allowing an agent to find out the physical location of several identical resources, could permit the agent to select the nearest one, thus greatly reducing communication delay.

Easy to use

No matter how complex the multiagent system is, this complexity has to be hidden form the users behind a simple and intuitive API. Programmers should be able to build simple agents easily, even if they don't know much about the framework or all its features, and then be able to evolve their skills gradually, as they use new and more advanced functionalities. Distribution transparency is a very good way to provide a simple programming model for the agents. Other important usability aspects include easy installation, tools facilitating tasks like administration and debugging, or visual agent design tools.

Security

Several important *security objectives* are illustrated in Table 3, adapted from [Menezes et al., 2001]. The other way of looking at security is to identify *security threats* such as: interception, interruption, modification and fabrication [Pfleeger and Pfleeger, 2002]. Very widespread these days are interruption attacks, such as distributed denial of service, which are very hard to prevent.

objective	***description***
access control	restricting access to resources to privileged entities.
confidentiality	keeping information secret from all but those who are authorized to see it.
data integrity	ensuring information has is not altered by unauthorized or unknown means.
anonymity	concealing the identity of an entity involved in some process.
non-repudiation	preventing the denial of previous commitments or actions.

Table 3: Several Objectives of Security

But finding security objectives and/or identifying the security threats, and then just stating that the system should be secure, is not the way to actually build a secure system. What is actually needed for this is a *security policy*: an exact description of the actions the entities (e.g. agents, users, framework components) are allowed to perform, and the actions that are restricted or prohibited. Once a security policy has been established, it is time to enforce it, by using *security mechanisms* such as: encryption, authentication, authorization and auditing.

The mechanisms to enforce security in an agent system are very similar to the general ones, used in classic distributed systems, and will not be further discussed here. However, in the presence of mobile agents, providing a secure executing environment for both agents and their hosts becomes a much more stringent problem. And, while being able to protect a host form a malicious agent is a problem that can be adequately dealt with nowadays, by using virtual machines and sandboxes, the problem of protecting an agent from the malicious host that executes it, is a much harder problem. The Ajanta platform was the testing ground for many innovative ways for protecting agents and their results are presented in [Karnik and Tripathi, 1999]. It is possible however, that the overhead of providing security for mobile agents, is bigger that the benefits of having mobile agents.

Pragmatism

While the theoretical foundations of the platform are important enough, more important is the practical applicability of the theoretical methods in the actual implementation. Some good pragmatic principles are:

- Solving real-world problems. Case studies and examples are good, but if we want the framework to be really useful, then actual implementation of real-world scenarios is needed.
- Technology re-use rather than re-invention - Use mature existing technologies whenever possible. Many of the "wheels" for building an agent framework already exist, in one way or another, and need only to be put together. Synthesizing existing research in the field is more effective and more honorable than reinventing the "wheel" (or worse, "a flat tire"). Only where the existing "wheels" are missing or are broken, it is meaningful to invent new ones. "The

whole then becomes greater than the sum of its different parts, and hence novel!" [Nwana and Ndumu, 1999].

- Test driven development. Base all assessment on working code, and in order to assure good working code write tests [Hamill, 2004].

4.2 Design Overview

ADF is built as a Java 2, Enterprise Edition (J2EE) application: a collection of enterprise beans and web components that work together in order to provide the high-level functionalities required by a multiagent platform (Illustration 11). ADF uses many of the services provided by a J2EE application sever, via standard interfaces like: JNDI, JMS, SAAJ, JMX, and other.

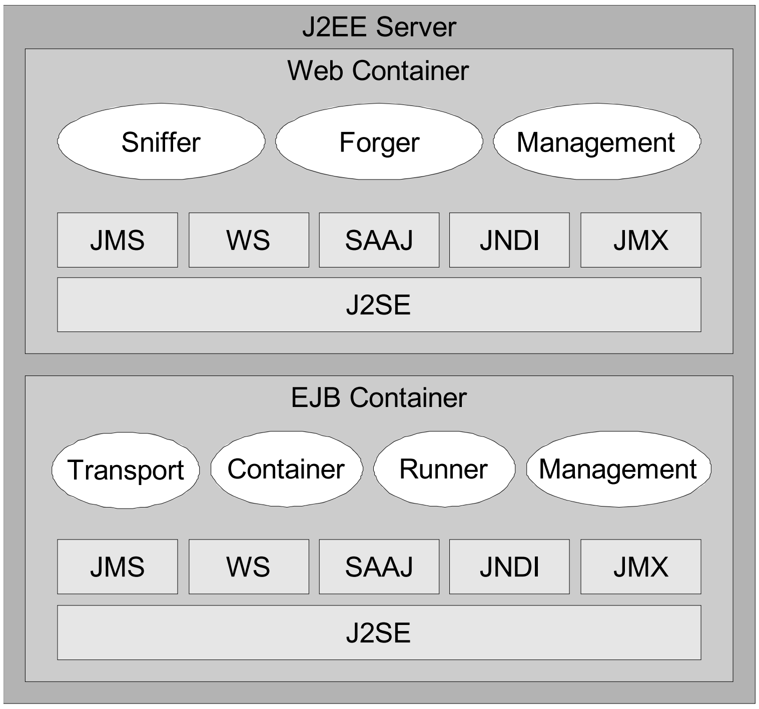


Illustration 11: The ADF Agent Platform

Proved platform independence is very important for any J2EE application. ADF was implemented and tested using JBoss 4.0, a J2EE 1.4 certified application server. This means that ADF will work with only minor changes (e.g. new deployment descriptors) on any other J2EE compliant application server. Several tests were already completed using the Sun Java System Application Server and support for more application servers is planed in the future. JBoss was chosen for the reference implementation because it is an open source, yet very powerful and widely used application server. For an introduction to JBoss we refer the reader to [JBoss Getting Started, 2005], and for a more in depth presentation we recommend the [JBoss Application Server Guide, 2005].

The ADF framework is divided into several collaborating components implemented either as enterprise beans (e.g. ManagementBean, ContainerBean, RunnerBean, LocalTransportBean and other) or as servlets (ManagementServlet, ForgerServlet, SnifferServlet).

4.3 The Agent Management System

The agent management system (AMS) is a mandatory component of the every FIPA compliant agent platform. It exerts supervisory control in the agent platform (Illustration 12).

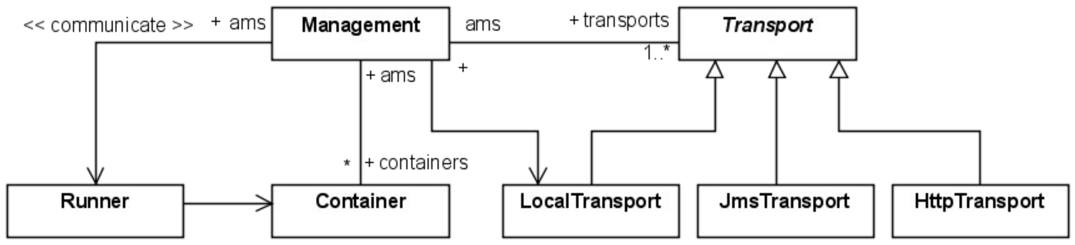


Illustration 12: Overview of the ADF Architecture

The agent management system is implemented in ADF as an enterprise bean, more exactly, a stateless session bean called ManagementBean. Multiple, concurrent requests can be serviced by several equivalent instances of this bean. This is a very important mechanism to achieve vertical scalability, and is intensively used in ADF. Although the bean itself is stateless, it uses the JNDI API to store information about the agents registered with the platform. This information is shared by all the instances of the bean, thus assuring a consistent behavior.

The ManagementBean is used internally by most other beans in the agent platform. Nevertheless, it is also used by application clients (e.g. servlets) in order to perform administrative tasks on behalf of a user. There are three steps to be followed, before this is actually possible:

- First, we obtain an initial JNDI context for the particular application server. For example, when using JBoss, this would be achieved by using the following code snippet:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "org.jnp.interfaces.NamingContextFactory");
env.put(Context.URL_PKG_PREFIXES,
    "org.jboss.naming:org.jnp.interfaces");
env.put(Context.SECURITY_PRINCIPAL, "myname");
env.put(Context.SECURITY_CREDENTIALS, "mypassword");
env.put(Context.PROVIDER_URL, "jnp://localhost:1099");
Context ctx = new InitialContext(env);
```

- Then, we obtain a reference to the bean's remote home interface using the initial context, and "narrow it" to the corresponding type:

```
Object o = ctx.lookup("ejb/Management");
ManagementRemoteHome home = (ManagementRemoteHome)
    PortableRemoteObject.narrow(o, ManagementRemoteHome.class);
```

- Finally, we run the create method on the remote home interface and acquire a new remote reference to the ManagementBean.

```
ManagementRemote ams = home.create();
```

The remote reference can then be used to perform tasks such as creating and terminating agents, or obtaining the results of their work. Access to these tasks can be restricted by a security policy.

Creating new a new agent is a simple task, completed by the ManagementBean. First we have to make sure the code of the agent is accessible to the application server. Since the simple agents we present here, are part of the ADF package, no further actions need to be taken. Otherwise, the mechanism to achieve this can vary depending on the application server. With JBoss a simple way to do this, is to copy the *jar* file containing the source of the agent to the *deploy* directory. Then we simply call one of the *create* methods, like this:

```
ams.createAgent("MyHelloWorldAgent", "net.sf.adf.samples.HelloWorldAgent");
```

Or, if arguments need to be passed:

```
ams.createAgent("MyHelloAgent", "net.sf.adf.samples.HelloAgent",
   new String[] {"Marta", "Sergiu"} );
```

This method will create both the agent and its container, and then schedule the agent for execution by the RunnerBean. No further concerns need to be addressed by the agent programmers in order to have their agents "alive".

The ManagementBean offers a white pages service to the agents and programs. This allows an agent to be looked up by name in order to obtain a reference to its container and opens the way to many other interesting operations:

```
ContainerRemote container = ams.lookup("MyHelloAgent");
```

The ManagementBean also allows clients to list the names of all the agents registered with the platform:

```
java.util.List<String> agents = ams.listAgents();
```

Once an agent finishes its work, the results can be obtained by simply invoking the *getResult* method:

```
Serializable result = ams.lookup("MyHelloAgent").getResult();
```

Trying to obtain the result of an agent before it is finished will result in an exception being thrown. In this case, and many others, it can be useful to verify the state of the agent first. We will examine the possible states in more detail in the next section. For now, we will just give an example illustrating how the state of the agent can be checked:

```
if (ams.lookup("MyHelloAgent").getState() == AgentState.FINISHED) {
   result = ams.getResult("MyHelloAgent");
}
```

Finally, the ManagementBean has a very important role in the administration of the different transport protocols, that are used simultaneously by the platform. We will discuss this in more detail in the next chapter.

4.4 The Agent Container

The agent container is responsible for holding an agent and providing it controlled access to the basic services of the framework. The agent container is the only entity to hold a reference to the agent it contains and it uses it to manage its lifecycle. This way, the agent's methods can only be called by the agent itself and its container, thus **the autonomy of the agent is guaranteed**. The agent is not just a mere object, because it is has complete control over its own lifecycle. The only exception to this general rule is when the agent management system decides to immediately terminate the agent. This may happen because the agent has not respected an important policy (e.g. the security policy) or when the owner of the agent has explicitly asked for its termination (e.g. when the agent is no longer responding to messages). Please note, however, that this is just an exception, and the general rule is that the agent executes autonomously for as long as it takes for it to complete its tasks.

The agent container functions as a façade that hides all the functionality of the framework from the agent, and offers it a very simple API (the *AgentContainer* interface). This API is generic and has no dependencies on other libraries (such as the J2EE API) so that agents are not only simple to write, but the whole agent framework could be reimplemented using any other technologies without affecting the existing code of the agents. In fact, the code of the agents doesn't even need to be recompiled in this case, a big plus for the extensibility of the framework. The agent container functions in fact as a façade the other way around too, the *Container* local interface hides the agent from the agent platform, in order to assure its autonomy (Illustration 13).

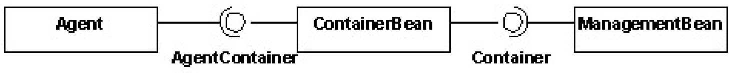


Illustration 13: The ContainerBean Seen as two Façades

We give below the current *AgentContainer* interface to illustrate its simplicity. For convenience the Agent class provides these methods too, delegating the work to its container. Many of the these methods will be further discussed in this section.

```
package net.sf.adf.agent;

```

```
import net.sf.adf.acl.ACLMessage;
import net.sf.adf.agent.task.Task;

public interface AgentContainer {
    AID getAID();
    org.apache.log4j.Logger getLogger();
    java.io.Serializable getArguments();
    void setResult(Serializable result);

    void send(ACLMessage message);
    java.util.Queue<ACLMessage> getMessageQueue();

    void addTask(Task t);
    void removeTask(Task t);
    void setTaskWaiting(Task t);
    void setTaskReady(Task t);
}
```

The agent container is implemented as a stateful session bean called ContainerBean. It holds the only reference to the agent and references to many other, very important objects: the agent's identifier, state, arguments, results, message queue, task scheduler and logger (Illustration 14).

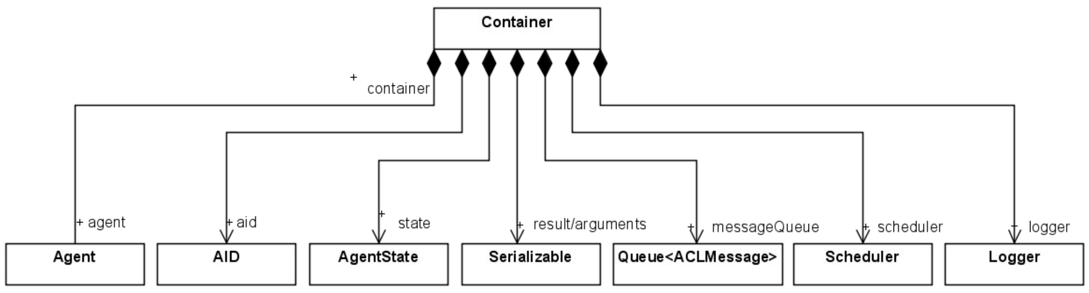


Illustration 14: The Contents of the ContainerBean

As previously mentioned, the agent container is responsible of managing the lifecycle of the agent (Illustration 15).

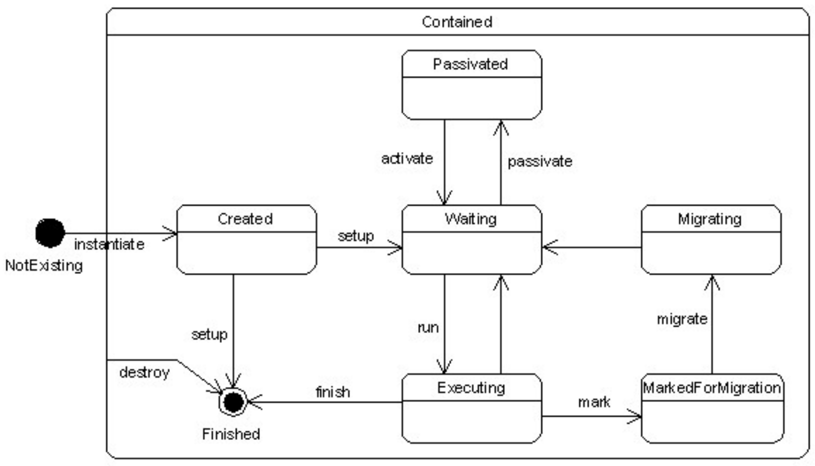


Illustration 15: The Lifecycle of an ADF Agent

ADF agents are always created by the ManagementBean and are immediately associated with a container that will host them for as long as they live. It is only mobile agents that can change their container, but still in a controlled way. Every agent is in the *NotExisting* state, until the ManagementBean instantiates it and puts it in the *Created* state. Immediately thereafter, the agent container invokes the *setup* method on the agent. The default implementation of the *setup* method simply returns, but every useful agent will override this method. The most usual thing to do in the setup method is to add tasks that will be run later, because otherwise the agent will be finished immediately after setup completes. If the agent has planed tasks however, it will be passed into the *Waiting* state once the setup method completes. An agent will execute (move into the *Executing* state) when the container receives a call from either the RunnerBean or a message from one of the transport beans, and will return to the *Waiting* state once processing is done. Mobile agents can mark themselves for migration (the *MarkedForMigration* state) and they will be migrated (the *Migrating* then the *Waiting* state) once processing is done. Finally, when the load on the J2EE application server is high, it might decide to passivate the instances of stateful session beans, in our case, the agent containers. When this happens the agent is set into the *Passivated* state, and it will be made active again, as soon as the container is referenced, for example when a a new message is received.

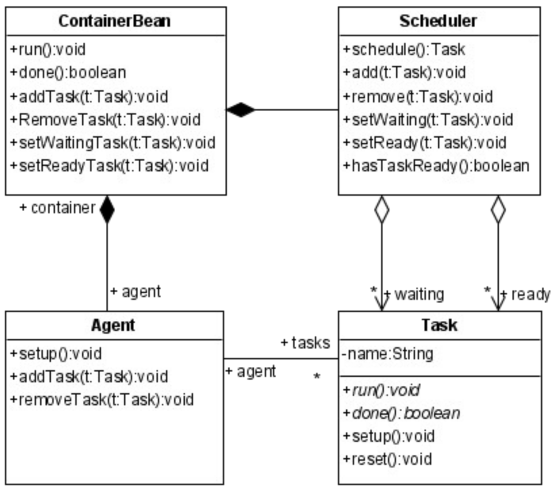


Illustration 16: The Task Scheduling Model in ADF

Another service provided by the agent container to the agent is task scheduling. The work an agent has to complete is divided into small tasks. This allows the agent code to run non-preemptively into a single thread and, at the same time, be responsive to asynchronous events, such as receiving a message or a notification of an expired timer. Our task model is very similar to that adopted by [Jade], which we find very simple, yet powerful (Illustration 16).

An active agent has a number of associated tasks at any given time, separated into two categories: *ready* and *waiting*, according to whether they can make progress, or not. A ready task is scheduled for execution every time the RunnerBean calls the run method on the ContainerBean. The exact task that is to be run is chosen by the Scheduler, according to a task scheduling algorithm, Round Robin for example. According to the Jade model all waiting tasks are made ready every time a message is received. While this may be simple and efficient for agents with several tasks, it doesn't scale well for tens or hundreds of tasks. We believe, that it might be possible to improve this model by using message patterns to decide which tasks need to be made ready, and which ones don't. In such a model each task would only be woken up, when it can do something meaningful, so programming tasks would be easier.

Now that we explained the agent lifecycle and task mechanism it is time to write our first agent - the "Hello World" agent. The agent will have only one task, that will be run only once, writing "Hello World" into a log file (managed by the container). The task will be added during setup, and it will extend the *OneShotTask* class, a subclass of *Task* that overrides the *done* method so that it always returns true.

```
package net.sf.adf.samples;

import net.sf.adf.agent.Agent;
import net.sf.adf.agent.task.OneShotTask;

public class HelloWorldAgent extends Agent {
    public void setup() {
        addTask(new OneShotTask() {
            public void run() {
                getLogger().info("Hello World");
            }
        });
    }
}
```

This is a very simple agent, so there is not much to explain, maybe except for the anonymous inner class. That is just a shortcut here, we could have just given it a name and put it outside the *HelloWorldAgent* class. The way to run this (and the following) agent is already given in the previous section. There, we have also shown, how to pass parameters to an agent, and retrieve its result. Here we present another simple agent that uses the arguments and provides a result:

```
package net.sf.adf.samples;

import net.sf.adf.agent.Agent;
import net.sf.adf.agent.task.OneShotTask;

public class HelloAgent extends Agent {
    public void setup() {
        String[] args = (String[])getArguments();
        for (final String arg : args) {
            addTask(new OneShotTask() {
                public void run() {
                    getLogger().info("Hello "+arg);
                }
            });
        }
        addTask(new OneShotTask() {
            public void run() {
                setResult("My name is "+getAID());
            }
        });
    }
}
```

The HelloAgents receives an array of Strings as an argument representing people names, and it greets every one of them by logging to a file. Then it transmits his name by running the *setResult* method. Once the agent is finished, the owner of the agent will be able to retrieve this information and use it, for example by printing it on screen.

Probably the most important service the agent container provides to the agent is access to asynchronous messaging. The container holds a message queue, where the agent can receive messages, and also forwards the messages sent by the agent to the corresponding transport(s). This mechanism is further examined in the next chapter.

4.5 Agent Runner

One other important component we have just mentioned previously is the RunnerBean.

It is a message-driven bean that is used internally by the framework to allow agents to be run concurrently. The RunnerBean is a simplification of the *Service Activator* EJB design pattern [Alur et al., 2003] that allows the invocation of business services, plain Java objects, or EJB components in an asynchronous manner.

Every time a new agent is created and initialized it is put into the *Waiting* state. After this, the ManagementBean sends an asynchronous message, containing a local stub for the agent container, to the queue served by the RunnerBean. An instance of the RunnerBean will process the message, deserialize the agent container stub and call its *run* method, which will in turn run one task of the agent. Once the run method completes, the *done* method of the agent container is executed, and, in the case it returns false, the message is resent to the RunnerBean's own queue. This guarantees that the cycle is restored, and the agent will be run again. If the *done* method returns true however, the message will not be resent. Because the agent has no ready tasks to be run the RunnerBean will no longer call the ContainerBean until the ContainerBean explicitly sends one more message to the RunnerBean's queue, for example when more tasks become ready because a message was received.

5. Agent Communication in ADF

Agent communication in ADF is message-oriented and closely follows the standard model mandated my FIPA. A major concern when designing ADF was the independence (orthogonality) between four different issues: the transport protocol, the ACL message encoding, the message contents and the interaction protocol. This not only assures much flexibility for the moment, but also provides a maximum degree of extensibility. Message content languages and interaction protocols were already examined in Chapter 3, so in this chapter we will focus only on transport protocols and message encodings.

5.1 Transport Protocols

Messages are transported from one agent platform to the other using a transport protocol. We already discussed two alternatives in Chapter 3: JMS and HTTP, and now we will examine their implementation is ADF. But firsts, we will discuss how a ADF transport works in general, and exemplify this on the simplest one: the local transport that sends messages inside an agent platform.

First of all, ADF defines a transport as a class that implements the net.sf.adf.transport.Transport interface, and this means providing an implementation to the send method there. Sending messages is however not everything a transport does. Generally, it will also receive messages in an asynchronous way. How exactly the transport does this, is intentionally left unspecified, as it is very much dependent on the particular transport protocol implemented. All three protocols we will present, also receive messages, but each of them does it in a conceptually different way.

There are however, many similarities in the way different transports work in ADF, and this is because the framework hides their inner workings from the agents and their containers. When an agent wants to send a message it first sets the intended receiver agent(s) and then runs its *send* method. This method forwards the message to the agent container by default.

```
AID receiver = new AID("receiver_name", false);
receiver.getAddresses().add(new URI("http://example.com/adf"));
ACLMessage message = new ACLMessage(Performative.INFORM);
message.getReceiverSet().add(receiver);
/* probably set other message parameters as well */
send(message);
```

The agent container goes through the intended receiver list, and passes the agent identifiers one at a time, to the ManagementBean. The ManagementBean holds a mapping between URL-prefixes and their corresponding transports and will return, a (possibly empty) set of transports. If the addresses explicitly given in the agent identifier have no corresponding transports, or if there are no such addresses given (very likely in real-world situations) then a global name resolution service could provide more addresses for the agent.

In our example the ManagementBean will most likely return just the HTTP transport (HttpSenderBean). However, if more than one address is present in the agent identifier or is obtained form the naming service, it is very likely that more than one address will be returned. No matter how many transports are returned, the agent container will try to send the message to each of them in turn, until one of the sends succeeds (throws no exception), or there are no more transports to try.

Generally speaking there are no guarantees whatsoever, that once a message is sent, it will arrive to its destination. When using a persistent message-oriented transport the chance that a message will be lost is usually very small, but still present. When using a reliable transport protocol (as described in section 3.5) a message that gets lost will always cause the sender to be notified, via a negative acknowledgment. The agent container however, has to send the message as soon as possible, usually to many different destinations (please note that the agent is blocked at this time because it shares the same execution thread with its container). This means that if a message is lost, and reliable communication is used, the negative acknowledgment will be later sent by the transport to the agent itself, which will have to deal with it accordingly. The whole process of sending a message to a single receiver is presented in Illustration 17.

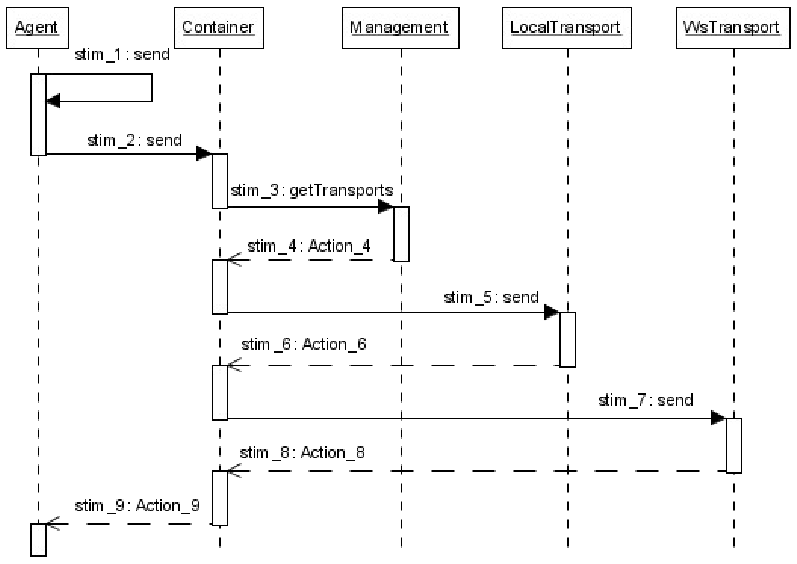


Illustration 17: Example of Sending a Message in ADF

When receiving messages, a similar (but reversed) interaction pattern is followed. The transport has to forward it to the corresponding agents and to do this, it first uses the *agentLookup* method of the ManagementBean to get a local reference to each agent container. The transport then passes the message to each of the corresponding containers, by calling their *receive* method. Once a message is received by the container, it is added to the message queue. Receiving a message can also have other side-effects for the agent, for example when all the waiting tasks that could process the received message become ready again. Anyway, the agent can choose whether it processes the messages immediately, or just ignores them for some time, waiting for some other event to occur (Illustration 18).

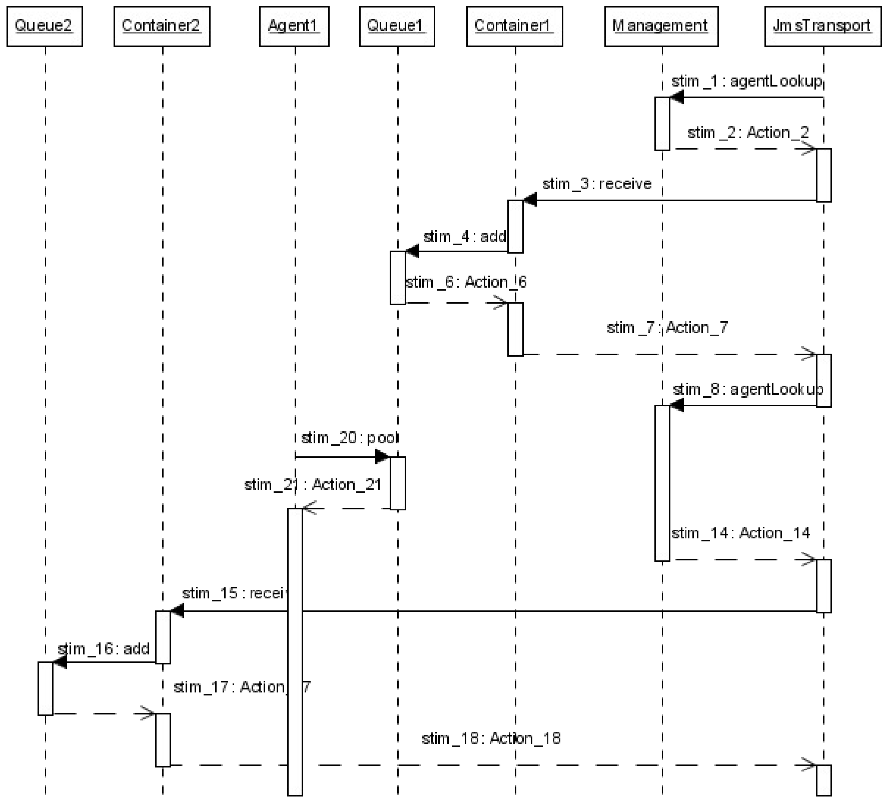


Illustration 18: Example of Receiving a Message in ADF

Local Transport

The LocalTransportBean is implemented as a stateless session bean, and deals with messages sent to agents that are registered within the same platform. The ManagementBean does not hold a URL prefix mapping for this transport, but always holds a local reference to the LocalTransportBean. When a message has to be sent to an agent that has the same platform identifier as the ManagementBean, the ManagementBean will just return the LocalTransportBean to the container. True location transparency is assured by the fact that the agent container has no easy way to distinguish between a reference to a the local transport and one to a non-local transport. The ContainerBean just sends the message to the transport as usual.

When a message is sent for transport to the LocalTransportBean, the bean acts as if it received the message from the outside, it finds out the receiving local containers and forwards the message to them. This process is transparent to all the beans with the exception of the ManagementBean and the LocalTransportBean itself. The only way for the receiving agent to find out whether a local transport is probably used, is by comparing his platform identifier with the platform identifier of the sender.

HTTP Transport

Because the World Wide Web is already ubiquitous, it is no surprise that HTTP makes a very good candidate for message delivery. This is accentuated by the fact that most enterprise firewalls filter much of the Internet traffic for security reasons. Typically almost all ports are closed to incoming and outgoing traffic, but port 80 has to be open because it is used for web browsers. Web services tunnel everything through port 80, and this is one of the reasons that makes the technology so appealing. Anyway, HTTP is not limited in any way to HTML, XML or SOAP. So we have actually abstracted from the encoding of the message itself, and built a generic HTTP transport.

Sending a message over HTTP requires first establishing a TCP connection to port 80 by on the receiving host and the transmitting data: first the "POST / HTTP/1.1" string, followed by a number of informational headers, followed by the actual message using an arbitrary encoding. The *Content-type* header is important because it is used to describe the encoding used for the message, for example "text/plain" for string encoding, "application/xml" for the XML encoding or "application/soap+xml" for the SOAP encoding.

The HTTP transport is implemented as two distinct components: a stateless session bean that sends messages (HttpSenderBean) and a servlet that receives them (HttpReceiverServlet). The HttpSenderBean uses a HttpURLConnection to send messages as follows:

```
ACLMessage message = /* ... */;
ACLCodec codec = /* ... */;
URI receiverURI = /* ... */;
URL receiverURK =  receiverURI.toURL();
HttpURLConnection connection =
    (HttpURLConnection)address.openConnection();
connection.setDoOutput(true);
connection.setRequestMethod("POST");
connection.setRequestProperty("Content-type", "text/plain");
PrintWriter out = new PrintWriter(connection.getOutputStream());
out.print(codec.encode(message));
out.close();
connection.disconnect();
```

The HttpReceiverServlet is even simpler. It extends HttpServlet and just overrides the *doPost* method:

```
protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
    /* ... */
    String contentType = request.getContentType();
    ACLCodec codec = ACLCodecFactory.getCodec(contentType);
    // content length is in bytes
    int contentLength = request.getContentLength();
    if (contentLength == -1) {
        contentLength = MAX_CAPACITY;
    }
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(request.getInputStream()));
    CharBuffer buffer = CharBuffer.allocate(1+contentLength/2);
    reader.read(buffer);
    ACLMessage message = codec.decode(buffer.toString());
    reader.close();
    /* ... */
}
```

The two examples above are surely contrived, since we left out some aspects such as exception handling. However, they should demonstrate that transporting messages over HTTP is quite an easy task, especially with the very good support offered by Java. We already mentioned other advantages of HTTP, so what is still missing? The basic problem is that HTTP itself doesn't provide guaranteed delivery. We already discussed about reliable message-oriented communication in Section 3.5, and the fact that we are still a long way to go from achieving this using just HTTP. In the next section we will present the JMS transport, which is surely more reliable than HTTP, because it provides persistent communication. However, we will see that this increase in reliability doesn't come without it's costs, especially when considering performance and scalability.

JMS Transport

Many enterprise applications require very high reliability for the individual transactions, and simple protocols such as HTTP are not enough. We already examined briefly the Java Message Service API in Section 3.8 and persistent communication in Section 3.5 so here we will focus only on the implementation of the JMS Transport.

The JMS Transport has two components: a stateless session bean for sending messages (JmsSenderBean) and a message-driven bean for receiving them asynchronously (JmsReceiverBean).

Sending JMS messages is a quite complex task, so we will split it into more steps:

- First of all, we need to get the initial context for the JMS provider of the receiving agent. Unfortunately, we do not know a standard way to do this, so we will stick with JBoss and its message queuing system JBossMQ:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "org.jnp.interfaces.NamingContextFactory");
env.put(Context.URL_PKG_PREFIXES,
    "org.jboss.naming:org.jnp.interfaces");
env.put(Context.PROVIDER_URL, "jnp://example.com:1099");
InitialContext initialContext = new InitialContext(env);
```

- Second, we use the JNDI initial context to look up the main queue of the ADF platform, not accidentally called "queue/ADFQueue". We use the generic JMS interfaces, so it makes in fact no difference whether we are dealing with a queue or a topic.

```
Destination dest = (Destination)initialContext.lookup("queue/ADFQueue");
```

- Third, we set up a connection, a session and a producer. The connection to the remote JMS provider is created by using a connection factory. In the case of JBoss there are plenty connection factories we can choose from, and because the actual selection happens in the JBoss-specific deployment descriptor of the bean, we will just assume an appropriate factory is bound to "java:comp/env/jms/ConnectionFactory" in the local initial context. Please note, that these initialization steps are quite time consuming and it would not be wise performing them for every JMS message. In fact the connection can be kept open for a longer period of time.

```
ConnectionFactory connectionFactory = (ConnectionFactory)
    localContext.lookup("java:comp/env/jms/ConnectionFactory");
Connection connection = connectionFactory.createConnection();
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
producer = session.createProducer(dest);
```

- Once this initial setup is done, messages can be created and sent to the destination.

```
String string = codec.encode(aclMessage);
Message message = runnerSession.createStringMessage(string);
message.setStringProperty("ContentType", codec.getMimeType());
producer.send(msg);
```

The JmsReceiverBean is used for receiving JMS messages asynchronously from the "queue/ADFQueue" destination. Message-driven beans only have to override the *onMessage* method in order to process the received messages:

```
public void onMessage(Message message) {
    /* ... */
    if (message instanceof TextMessage) {
        TextMessage textMessage = (TextMessage) message;
        String ContentType =
            textMessage.getStringProperty("ContentType");
        ACLCodec codec = ACLCodecFactory.getCodec(contentType);
        ACLMessage message = codec.decode(textMessage .getText());
    } else {
        logger.error("Unsuported message type"+message.getJMSType());
    }
    /* ... */
}
```

Some JMS providers can send also messages over HTTP, so using JMS does not necessarily mean giving up all the advantages of HTTP. In case the performance overhead introduced by a message queuing system is less important then reliability, the protocol stack from Illustration 19 could be meaningful.

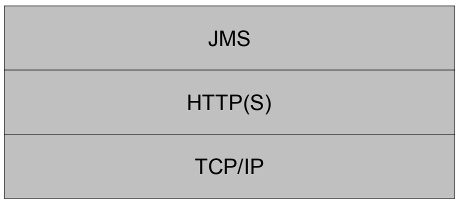


Illustration 19: JMS over HTTP

5.2 ACL Encodings

The agent communication language used by ADF agents is FIPA ACL (consult Section 3.4 for details). Fixing the message structure is very important in order to achieve interoperability. The structure of a FIPA ACL message is however extensible, and it allows custom properties to be added.

Inside agent platforms, the ACL messages are represented as instances of the ACLMessage class. This class provides access to all the properties described in Appendix 7.1, and also to any custom one. However, when ACLMessages need to be exchanged between different agent platforms, they have to be encoded before they are actually send to the destination and then decoded when they are received. FIPA has defined three standard encodings for ACL messages: String [FIPA00070], XML [FIPA00071] and bit efficient. ADF already fully supports the first two, and additionally provides a SOAP encoding. Finally, the ACLMessageFactory can be used to instantiate the appropriate codec for a given MIME Type [MIME Types].

String Codec

The StringCodec class provides support for encoding ACL messages to string and decoding them from strings. Encoding a message to a string simply implies appending them one at a time to a StringBuffer. Decoding them, however is more complex and a parser was generated with javacc according to the standard grammar given by [FIPA00070]. Here is an example ACL message encoded as a string:

```
(propose
:sender (agent-identifier :name agent1@example.com)
:receiver (set
    (agent-identifier :name agent2@example.com)
    (agent-identifier  :name agent3@mycompany.com
         :resolvers (sequence (
              agent-identifier :name ams@mycompany.com
         :addresses (sequence http://gabriela.com/adf/a2 )) )
         :X-custom value) )
:reply-to (set
    (agent-identifier :name agent1@example.com)
    (agent-identifier :name agent4@example.com))
:content "((action j (sell plum 50))(= (any ?x (and (= (price plum) ?
x) (< ?x 10))) 5)"
:language fipa-sl
:ontology fruit-market
:protocol fipa-contract-net
:conversation-id 31465210548055
:reply-with 42652015871989
:in-reply-to cfr46520548055
:reply-by 20050518T042600765Z
:X-custom stuff)
```

XML Codec

The XMLCodec uses standard DOM operations in order to encode and decode ACL Messages as XML.

- First, we have to obtain a DOMImplementation, in a parser dependent way. For the Xerces implementation that is part of JDK1.5 we use the following code:

```
DOMImplementation domImpl=(DOMImplementation)com.sun.org.apache
.xerces.internal.dom.DOMImplementationImpl.getDOMImplementation();
```

- Then we have to make sure that the DOMImplementation supports Loas/Save 3.0, because those features will be used for serializing/deserializing DOM trees.

```
if (!domImpl.hasFeature("LS", "3.0")) {
    throw new ParserConfigurationException(
        "Load/Save 3.0 not supported");
}
```

- We use the DOMImplementation to create new documents:

```
Document doc = domImpl.createDocument(namespaceURI,
    rootElementQualifiedName, docType);
```

- We cast the DOMImplementation to a DOMImplementationLS:

```
DOMImplementationLS domImplLS = (DOMImplementationLS)domImpl;
```

- Using the DOMImplementationLS we parse and validate existing documents using DTDs:

```
LSInput input = domImplLS.createLSInput();
input.setCharacterStream(reader);
LSParser parser = domImplLS.createLSParser(
        DOMImplementationLS.MODE_SYNCHRONOUS,
        validating ? "http://www.w3.org/TR/REC-xml" : null);
parser.getDomConfig().setParameter("validate", validating);
Document doc = parser.parse(input);
```

- Finally, we also serialize DOM trees with the DOMImplementationLS:

```
LSOutput output = domImplLS.createLSOutput();
output.setCharacterStream(writer);
LSSerializer serializer = domImplLS.createLSSerializer();
serializer.write(doc, output);
```

SOAP Codec

The SOAPCodec is actually not a standalone class, but a decorator to be used on an XMLCodec. The SOAP codec uses the SOAP with Attachments API for Java (SAAJ) in order to add a SOAP envelope and relevant SOAP headers, to a standard XML-encoded message.

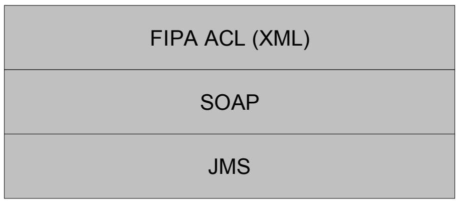


Illustration 20: SOAP over JMS

Using the SOAP encoding over the HTTP transport is a very natural choice. However, SOAP can be used on any other transport, including JMS. This choice is described in several articles including [Du and Liu, 2004] and [Carbone, 2002], and is also made available in ADF (Illustration 20).

6. Conclusion and Future Work

The results we achieved so far are encouraging. However, there are still some missing pieces needed in order for agent communication to be a solved problem, not only for ADF but for any agent platform:

- A standard reliable asynchronous transport protocol such as [WS-Reliability] or [WS-ReliableMessaging], but with the support of the whole community. Having only one, widely accepted standard is the only way to achieve interoperability.
- A better XML encoding for FIPA ACL messages. The DTD presented in Appendix 7.4 fixes some of the more obvious flaws. However, in order to ensure interoperability FIPA itself should fix this encoding and make update the standard accordingly.
- The ontology problem. Although RDF and OWL are a big step forward, and some task-oriented ontologies are already de-facto standards, more works needs to be put into this.

Future Work

While agent communication can be regarded as a completed task for ADF, there are still some things that can further be improved in order to make the framework viable, such as better developing and administration tools. Two other important issues not addressed to date by ADF are agent mobility and security.

Final quote

"We will always be stuck in the middle of an endless journey, not knowing whether the final answer is just around the corner or a million miles away" - Dennis Overbye

7. Appendices

7.1 The FIPA ACL Message Structure

Parameter	***Description***
performative	The communicative act of the ACL message (see)
sender	The agent performing the communicative act
receiver	The intended recipient agent(s)
reply-to	The agent to receive subsequent messages in this conversation
content	The object of the action implied by the communicative act
language	The language in which the content parameter is expressed
encoding	The specific encoding of the content language expression
ontology	The ontology(s) used to give a meaning to the symbols in the content
protocol	The interaction protocol that the sending agent is employing with this ACL message. (see Table)
conversation-id	Introduces an expression which is used to identify the ongoing sequence of communicative acts that together form a conversation.
reply-with	Introduces an expression that will be used by the responding agent to identify this message.
in-reply-to	Denotes an expression that references an earlier action to which this message is a reply.
reply-by	Denotes a time and/or date expression which indicates the latest time by which the sending agent would like to receive a reply.

7.2 The FIPA ACL Communicative Acts

Performative	***Description***
accept-proposal	The sender accepts a previously submitted proposal to perform an action.
agree	The sender agrees to perform some action, possibly in the future.
cancel	Inform the receiver that the sender no longer has the intention that the receiver performs a previously requested action.

Performative	***Description***
cfp	The action of calling for proposals to perform a given action.
confirm	The sender informs the receiver that a given proposition is true, where the receiver is known to be uncertain about the proposition.
disconfirm	The sender informs the receiver that a given proposition is false, where the receiver is known to believe, or believe it likely that, the proposition is true.
failure	The action of telling another agent that an action was attempted but the attempt failed.
inform	The sender informs the receiver that a given proposition is true.
inform-if	A macro action for the agent of the action to inform the recipient whether or not a proposition is true.
inform-ref	A macro action for the sender to inform the receiver the object which corresponds to a descriptor, for example, a name.
not-understood	The sender of the act informs the receiver that it perceived that the receiver performed some action, but that i did not understand what the receiver just did. A particular common case is that the sender did not understand a message previously received from the receiver.
propagate	The sender intends that the receiver treat the embedded message as sent directly to the receiver, and wants the receiver to identify the agents denoted by the given descriptor and send the received propagate message to them.
propose	Submit a proposal to perform a certain action, given certain preconditions.
proxy	The sender wants the receiver to select target agents denoted by a given description and to send an embedded message to them.
query-if	The sender asks the receiver whether a given proposition is true.
query-ref	The sender asks the receiver for the object referred to by a referential expression.
refuse	The sender refuses to perform a given action, and explains the reason for the refusal.
reject-proposal	The sender informs the receiver that it has no intention that the recipient performs the given action under the given preconditions.
request	The sender requests the receiver to perform some action. One important class of uses of the request act is to request the receiver to perform another communicative act.
request-when	The sender wants the receiver to perform some action when some given proposition becomes true.

Performative	*Description*
request-whenever	The sender wants the receiver to perform some action as soon as some proposition becomes true and thereafter each time the proposition becomes true again.
subscribe	The act of requesting a persistent intention to notify the sender of the value of a reference, and to notify again whenever the object identified by the reference changes.

7.3 FIPA Interaction protocols

- FIPA Request Interaction Protocol Specification [FIPA00026]
- FIPA Query Interaction Protocol Specification [FIPA00027]
- FIPA Request When Interaction Protocol Specification [FIPA00028]
- FIPA Contract Net Interaction Protocol Specification [FIPA00029]
- FIPA Iterated Contract Net Interaction Protocol Specification [FIPA00030]
- FIPA English Auction Interaction Protocol Specification [FIPA00031]
- FIPA Dutch Auction Interaction Protocol Specification [FIPA00032]
- FIPA Brokering Interaction Protocol Specification [FIPA00033]
- FIPA Recruiting Interaction Protocol Specification [FIPA00034]
- FIPA Subscribe Interaction Protocol Specification [FIPA00035]
- FIPA Propose Interaction Protocol Specification [FIPA00036]

7.4 The ADF Message DTD

```
<!ENTITY % communicative-acts "accept-proposal | agree | cancel | cfp
| confirm | disconfirm | failure | inform | not-understood | propose |
query-if | query-ref | refuse | reject-proposal | request | request-
when | request-whenever | subscribe | inform-if | inform-ref | proxy |
propagate">

<!ENTITY % msg-param "receiver | sender | content | language |
encoding | ontology | protocol | reply-with | in-reply-to | reply-by |
reply-to | conversation-id | user-defined">

<!ELEMENT fipa-message (%msg-param;)*>
<!ATTLIST fipa-message
    act (%communicative-acts;) #REQUIRED
>

<!ELEMENT sender (agent-identifier)>

<!ELEMENT receiver (agent-identifier+)>
```

```
<!ELEMENT content (#PCDATA)>

<!ELEMENT language (#PCDATA)>

<!ELEMENT encoding (#PCDATA)>

<!ELEMENT ontology (#PCDATA)>

<!ELEMENT protocol (#PCDATA)>

<!ELEMENT reply-with (#PCDATA)>

<!ELEMENT in-reply-to (#PCDATA)>

<!ELEMENT reply-by EMPTY>
<!ATTLIST reply-by
    time CDATA #REQUIRED
>

<!ELEMENT reply-to (agent-identifier+)>

<!ELEMENT conversation-id (#PCDATA)>

<!ELEMENT agent-identifier (name, addresses?, resolvers?, user-
defined*)>
<!ELEMENT name EMPTY>
<!ATTLIST name
    id CDATA #IMPLIED
>

<!ELEMENT addresses (url+)>
<!ELEMENT url EMPTY>
<!ATTLIST url
    href CDATA #IMPLIED
>
<!ELEMENT resolvers (agent-identifier+)>

<!ELEMENT user-defined ANY>

<!-- Proprietary extension (fixing completeness problem)
http://java.sun.com/dtd/properties.dtd -->
<!ELEMENT properties ( comment?, entry* )>
<!ATTLIST properties version CDATA #FIXED "1.0">
<!ELEMENT comment (#PCDATA)>
<!ELEMENT entry (#PCDATA)>
<!ATTLIST entry key CDATA #REQUIRED>
```

8. Bibliography

[*Aglets*] *Aglets*. http://aglets.sourceforge.net.

[*Ajanta*] *Ajanta: Mobile Agents Research Project*. http://www.cs.umn.edu/Ajanta.

[*Alur et al., 2003*] Deepak Alur, John Crupi, Dan Malks - *Core J2EE Patterns: Best Practices and Design Strategies, Second Edition*. Prentice Hall PTR, 2003.

[*Blair and Stefani, 1998*] Gordon Blair, Jean-Bernard Stefani - *Open Distributed Processing and Multimedia*. Addison-Wesley, 1998.

[*Burke et al., 2004*] Bill Burke, Sacha Labourey, Richard Monson-Haefel - *Enterprise JavaBeans, Fourth Edition*. O'Reilly, 2004.

[*Carbone, 2002*] Gunnison Carbone - *Enhancing Web Services Infrastructures with JMS*. 2002. http://www.onjava.com/pub/a/onjava/2002/06/19/jms.html

[*CCL*] *Constraint Choice Language*. http://www.lsi.upc.es/~steve/Publications/ccl_specification_v0201.pdf.

[*Chappell, 2004*] Dave Chappell - *Enterprise Service Bus*. O'Reilly, 2004.

[*Chavez and Maes, 1996*] Anthony Chavez, Pattie Maes - *Kasbah: An Agent Marketplace for Buying and Selling Goods*. 1996. http://www-ec.njit.edu/~bartel/NegoPap/KasbahMIT.pdf

[*Cougaar*] *Cognitive Agent Architecture*. http://cougaar.org/.

[*D'Agents*] *D'Agents: Mobile Agents at Dartmouth College*. http://agent.cs.dartmouth.edu.

[*DIET Agents*] *DIET Agents*. http://diet-agents.sourceforge.net/.

[*Du and Liu, 2004*] Helen Du, Jeffrey Liu - *Building a JMS Web service using SOAP over JMS and WebSphere Studio*. 2004. http://www-128.ibm.com/developerworks/websphere/library/techarticles/0402_du/0402_du.html

[*EJB3*] *JSR 220: Enterprise JavaBeans 3.0*. http://www.jcp.org/en/jsr/detail?id=220.

[*Erl, 2004*] Thomas Erl - *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Prentice Hall PTR, 2004.

[*Fielding, 2000*] Roy Fielding - *Architectural Styles and the Design of Network-based Software Architectures, Chapter 5: Representational State Transfer (REST)*. 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[*FIPA*] *The Foundation for Intelligent Physical Agents*. http://www.fipa.org/.

[*FIPA-OS*] *FIPA-OS Agent Toolkit*. http://fipa-os.sourceforge.net.

[*FIPA00001*] *FIPA Abstract Architecture Specification*.

http://www.fipa.org/specs/fipa00001/SC00001L.html.

[*FIPA00008*] *FIPA SL Content Language Specification*. http://fipa.org/specs/fipa00008/SC00008I.html.

[*FIPA00011*] *FIPA RDF Content Language Specification*. http://www.fipa.org/specs/fipa00011/XC00011B.html.

[*FIPA00023*] *FIPA Agent Management Specification*. http://www.fipa.org/specs/fipa00023/SC00023K.html.

[*FIPA00026*] *FIPA Request Interaction Protocol Specification*. http://fipa.org/specs/fipa00026/SC00026H.html.

[*FIPA00027*] *FIPA Query Interaction Protocol Specification*. http://fipa.org/specs/fipa00027/SC00027H.html.

[*FIPA00028*] *FIPA Request When Interaction Protocol Specification*. http://fipa.org/specs/fipa00028/SC00028H.html.

[*FIPA00029*] *FIPA Contract Net Interaction Protocol Specification*. http://fipa.org/specs/fipa00029/SC00029H.html.

[*FIPA00030*] *FIPA Iterated Contract Net Interaction Protocol Specification*. http://fipa.org/specs/fipa00030/SC00030H.html.

[*FIPA00031*] *FIPA English Auction Interaction Protocol Specification*. http://fipa.org/specs/fipa00031/XC00031F.html.

[*FIPA00032*] *FIPA Dutch Auction Interaction Protocol Specification*. http://fipa.org/specs/fipa00032/XC00032F.html.

[*FIPA00033*] *FIPA Brokering Interaction Protocol Specification*. http://fipa.org/specs/fipa00033/SC00033H.html.

[*FIPA00034*] *FIPA Recruiting Interaction Protocol Specification*. http://fipa.org/specs/fipa00034/SC00034H.html.

[*FIPA00035*] *FIPA Subscribe Interaction Protocol Specification*. http://fipa.org/specs/fipa00035/SC00035H.html.

[*FIPA00036*] *FIPA Propose Interaction Protocol Specification*. http://fipa.org/specs/fipa00036/SC00036H.html.

[*FIPA00061*] *FIPA ACL Message Structure Specification*. http://www.fipa.org/specs/fipa00061/SC00061G.html.

[*FIPA00067*] *FIPA Agent Message Transport Service Specification*. http://www.fipa.org/specs/fipa00067/SC00067F.html.

[*FIPA00070*] *FIPA ACL Message Representation in String Specification*. http://fipa.org/specs/fipa00070/SC00070I.html.

[*FIPA00071*] *FIPA ACL Message Representation in XML Specification*. http://fipa.org/specs/fipa00071/SC00071E.html.

[*Franklin and Graesser, 1996*] Stan Franklin, Art Graesser - *Is it an Agent, or just a*

Program?: A Taxonomy for Autonomous Agents. 1996. http://www.msci.memphis.edu/~franklin/AgentProg.html

[*Gehtland and Tate, 2004*] Justin Gehtland, Bruce A. Tate - *Better, Faster, Lighter Java*. O'Reilly, 2004.

[*Haase 2002*] Kim Haase - *Java Message Service API Tutorial*. Sun Microsystems, 2002.

[*Hamill, 2004*] Paul Hamill - *Unit Test Frameworks*. O'Reilly, 2004.

[*He, 2003*] Hao He - *What is Service-Oriented Architecture?*. 2003. http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html

[*HTTPR*] *HTTPR Specification*. http://www.ibm.com/developerworks/library/ws-httprspec.

[*Jade*] *Java Agent DEvelopment Framework*. http://jade.tilab.com.

[*JBoss*] *JBoss Application Server*. http://www.jboss.org.

[*JBoss Application Server Guide, 2005*] JBoss - *The JBoss 4 Application Server Guide, Release 3*. JBoss, 2005. http://docs.jboss.org/jbossas/jboss4guide/r3/html/

[*JBoss Getting Started, 2005*] JBoss - *Getting Started with JBoss 4.0, Release 4*. JBoss, 2005. http://docs.jboss.org/jbossas/getting_started/v4/html/

[*Jennings and Wooldridge, 1998*] N. R. Jennings and M. Wooldridge - *Applications of Intelligent Agents*. 1988. http://www.cs.umbc.edu/agents/introduction/jennings98.pdf

[*JNDI Tutorial*] *The JNDI Tutorial*. http://java.sun.com/products/jndi/tutorial/index.html.

[*Johnson and Foote, 1988*] R. E. Johnson and B. Foote - *Designing reusable classes*. 1988.

[*Johnson and Hoeller, 2004*] Rod Johnson, Juergen Hoeller - *Expert One-on-One: J2EE Development without EJB*. Wiley, .

[*Karnik and Tripathi, 1999*] Neeran M. Karnik, Anand R. Tripathi - *Security in the Ajanta Mobile Agent System*. 1999. http://www.cs.umn.edu/Ajanta/papers/security-ajanta.ps

[*Kaye, 2003*] Doug Kaye - *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003.

[*KIF*] *Knowledge Interchange Format*. http://logic.stanford.edu/kif/kif.html.

[*Kozierok and Maes, 1993*] Robyn Kozierok and Pattie Maes - *A Learning Interface Agent for Scheduling Meetings*. 1993.

[*KQML*] *Knowledge Query and Manipulation Language*. http://www.cs.umbc.edu/kqml/.

[*Krafzig et al., 2004*] Dirk Krafzig, Karl Banke, Dirk Slama - *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall PTR, 2004.

[*Lange and Oshima, 1999*] D. Lange and M. Oshima - *Seven good reasons for mobile*

agents. 1999.
http://portal.acm.org/citation.cfm?id=298136&coll=portal&dl=ACM&CFID=47928167&CFTOKEN=2067051

[*LGPL*] *GNU Lesser General Public License*. http://www.gnu.org/copyleft/lesser.html.

[*Marinescu, 2002*] Floyd Marinescu - *EJB Design Patterns*. Wiley, 2002.

[*Matena et al., 2003*] Vlada Matena, Sanjeev Krishnan, Linda DeMichiel, Beth Stearns - *Applying Enterprise JavaBeans, Second Edition*. Addison Wesley, 2003.

[*Menezes et al., 2001*] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone - *Handbook of Applied Cryptography*. CRC Press, 2001.
http://www.cacr.math.uwaterloo.ca/hac/

[*MIME Types*] *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. http://www.ietf.org/rfc/rfc2046.txt.

[*Monson-Haefel and Chappell, 2001*] Richard Monson-Haefel, David A. Chappell - *Java Message Service, First Edition*. O'Reilly, 2001.

[*Neuman, 1994*] B. Clifford Neuman - *Scale in Distributed Systems*. 1994.
http://www.isi.edu/people/bcn/papers/pdf/94--_scale-dist-sys-neuman-readings-dcs.pdf

[*Nichifor and Buraga, 2004*] Ovidiu Nichifor, Sabin-Corneliu Buraga - *ADF - Abstract Framework forDeveloping Mobile Agents*. 2004.
http://thor.info.uaic.ro/~busaco/publications/tr/adf.pdf

[*Nichifor, 2004*] *Age of Agents - Software Agent Framework*. .

[*Nwana and Ndumu, 1999*] Hyacinth S. Nwana, Divine T. Ndumu - *A Perspective on Software Agents Research*. 1999. http://agents.umbc.edu/introduction/hn-dn-ker99.pdf

[*OASIS*] *Organization for the Advancement of Structured Information Standards*. http://www.oasis-open.org.

[*Omega*] *Omega*. http://thor.info.uaic.ro/~busaco/projects/#omega.

[*Pfleeger and Pfleeger, 2002*] Charles Pfleeger, Shari Lawrence Pfleeger - *Security in Computing, 3rd Edition*. Prentice Hall PTR, 2002.

[*RDF*] *Resource Description Framework*. http://www.w3.org/RDF/.

[*RestWiki*] *RestWiki*. http://rest.blueoxen.net/cgi-bin/wiki.pl.

[*RFC2396*] *Uniform Resource Identifiers: Generic Syntax. Request for Comments*. http://www.ietf.org/rfc/rfc2396.txt.

[*RFC2616*] *Hypertext Transfer Protocol -- HTTP/1.1*.
http://www.ietf.org/rfc/rfc2616.txt.

[*RFC2821*] *Simple Mail Transfer Protocol*. http://www.ietf.org/rfc/rfc2821.txt.

[*Searle, 1969*] John Searle - *Speech Acts: An Essay in the Philosophy of Language*. 1969.

[*SOAP*] *SOAP Version 1.2*. http://www.w3.org/TR/soap.

[*SOAP Primer, 2003*] *SOAP Version 1.2 Part0: Primer*. http://www.w3.org/TR/2003/REC-soap12-part0-20030624.

[*Sullins and Whipple, 2003*] Benjamin G. Sullins, Mark B. Whipple - *EJB CookBook*. Manning, 2003.

[*Tanenbaum and van Steen, 2002*] Andrew S. Tanenbaum, Maarten van Steen - *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[*Tomcat*] *Apache Tomcat*. http://jakarta.apache.org/tomcat/.

[*Voyager*] *Voyager Java Development Platform*. http://www.recursionsw.com/voyager.htm.

[*W3C*] *World Wide Web Consortium*. http://www.w3.org/.

[*Walls and Breidenbach, 2005*] Craig Walls, Ryan Breidenbach - *Spring in Action*. Manning, 2005.

[*WebsphereMQ*] *WebSphere MQ*. http://www-306.ibm.com/software/integration/wmq/.

[*Winograd and Flores, 1987*] Terry Winograd and Fernando Flores - *Understanding Computers and Cognition - A New Foundation for Design*. Addison Wesley, 1987.

[*WS-Coordination*] *Web Services Coordination*. ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf.

[*WS-I*] *Web Services Interoperability Organization*. http://www.ws-i.org.

[*WS-Reliability*] *OASIS Web Services Reliable Messaging TC*. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm.

[*WS-ReliableMessaging*] *Web Services Reliable Messaging*. http://www-128.ibm.com/developerworks/library/specification/ws-rm/.

[*WS-Security*] *OASIS Web Services Security TC*. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

[*WS-Transaction*] *Web Services Atomic Transaction*. ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf.

[*WSA*] *Web Services Architecture*. http://www.w3.org/TR/ws-arch/.

[*WSBPEL*] *OASIS Web Services Business Process Execution Language TC*. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[*ZEUS*] *Zeus Agent Toolkit*. http://sourceforge.net/projects/zeusagent/.