Multi Agent Systems A Comparative Study of Intelligent Collaborating Agents' Standards

By
Captain Muhammad Ali Amer
(10 FF Regt)

Dissertation for Partial Fulfillment of requirements of the National University of Sciences and Technology for award of B.E degree in Software Engineering

Department of Computer Science
Military College of Signals, National University of Sciences and Technology, Rawalpindi

October 2001

ABSTRACT

Aim of the project is to investigate different agent platforms, choosing an implementation method and developing an intelligent agent application prototype using the FIPA standard. The implementation is based on FIPA open source standards. The project investigates two methods of agent realization, using available solutions of FIPA and IBM Aglets. All the options are studied and their suitability for the project are evaluated and based on this study the best method (FIPA) is chosen for implementation. An electronic market place is selected for the implementation and a prototype is implemented using *agent role modeling* as the design method. Report to all the stages of the implementation is given. Discussion about the future work justifies the chosen implementation method, and lists some future work that could be done on the subject.

ACKNOWLEDGEMENTS

I thank Allah. Many people have contributed to this project, and without their help, I couldn't have done it. In no order of importance: Many thanks to my fellow student Captain Aslam Adnan, who was very helpful to me during the entire project. I'm very grateful to Dr Shaiq A. Haq for steering me in the correct direction and making me start a unique research oriented project. The constant checks and repeated presentations on progress to Dr. Arshad Ali kept me on my toes throughout the entire development period. I could never have managed to come this far had it not been for his elderly support and firm resolve to get work done. Of course, I'd like to thank Dr. Iosif Legrand for the constant flow of new targets and his want for innovation. With his specialty in agent related development, he has an ocean of knowledge to offer. I am proud to be part of the CERN team with him and Dr Arshad Ali. I would also thank my family for bearing my untimely hours of work, specially my mother (who always ensured an abundant supply of food in my tummy). Her prayers did and still continue to give me direction and strength, Thank you Maa Ji.

Contents

List of Illustrations

Code Listings

List of Tables

Chapter 1

Background and Concepts

1 Introduction

"An agent is a piece of software capable of acting intelligently on behalf of a user or users in order to accomplish a task."

1.1 Historical Background

Agents and agent mobility goes hand in hand with distributed computing and the Internet.

1.1.1 Internet

The global Internets predecessor was the Advance Research Projects Agency Network (ARPANET) of the US department of defence, started in early sixties. Although ARPANET was a military project; it was formed with emphasis towards research. The objectives of the project were from the beginning to study networked computers and increase computer research through resource sharing – very much the principles the Internet was later founded on. In the sixties, there was no general standard on how the computers systems worked. It was common that computer labs each used different systems (some of them unique!), and one of the research issues was to come up with a protocol that the computers could use to communicate. In 1969 first Interface Message Processor (IMP – the standard developed by ARPA) was installed at the University of California Los Angeles, making it the first node in the network. By the end of the year, there were four nodes in the network 1. In the seventies, numerous computers were added to the network. ARPANET and other early networks were, however, purpose built and dedicated to closed communities like universities. There was no pressure for the individual networks to be compatible – in addition commercial companies were pursuing their interests as well. US National Science Foundation (NSF) saw the problems arousing from this, and started its own project in 1985 called NSFNET (project had existed in fractions before); they announced that their intent was to serve the entire academic community. For example, NSF would give universities grants for getting a connection to the network, but only if they agreed to let all qualified users in the campus to use the network. They also made the critical decision in 1985 to make TCP/IP mandatory protocol in the NSFNET. The real growth of the Internet begun in 1989 2 – while it had took 20 years for the number of hosts in NSFNET to reach 100 000, this

figure now tripled in one year alone. The same year the first commercial Internet provider started – appropriately named “The World” – and the next year first commercial services were provided in the Internet by Clari-Net, soon followed by big companies like Compuserve and MCI. In 1989-90 the most used services in the Internet where email, FTP and Telnet, graphical World Wide Web was just a theory in minds of a few CERN scientists. One of them was Tim Berners-Lee – often titled as the father of the WWW, now the director of W3C – who developed concept of hypertext (to be later called HTML) that was demonstrated first time in 1990. Next year Berners-Lee introduced first GUI browser (also an editor) called “Nexus”, and in 1993 CERN announced that it was not going to patent the idea of WWW opening the way to success for the idea. The affect of this revelation can be seen in number of HTTP Web servers around the world (in 40 countries) rises from 50 in the start of the year to over 600 in the end of it. The rise of numbers of Web servers contributes also to the publication of $Mosaic_3$ that by the end of the year was available for both UNIX and Windows. Media got permanently interested in the Internet, and the phenomena kept on growing in exponential rate – in 1994 more commercial .com sites are registered than universities .edu sites. Many Internet shops open on the same year, and the first genuine Internet company, Netscape – evolved from Mosaic – is launched, and next year while Netscape went public, commercial companies such as Yahoo! and Lycos were born. In 1995 the amount of traffic in HTTP first time succeeded the amount of traffic in FTP. Microsoft launched a competitor for Netscape called Internet Explorer, and although Netscape is still on the lead with its new Netscape Navigator with frames, animated GIFs and JavaScript there is for the first time some serious competition. In 1995 the Open Source movement starts with Apache Group who introduce a high performance web server called Apache that quickly became the most used server in the Internet. The same year Sun Microsystems announced new programming language called Java, and a browser based on it called HotJava. The browser never took on, but the language did, and next year Sun released first Java development kit (JDK), and Netscape integrated Java support into Navigator. On the other hand, development on the last few years has been business as usual – World Wide Web Consortium (W3C) releases new versions of HTML, and developed XML and RDF. On the other hand, governments are now restricting the Internet trying to control the

information available. In the US, the controversial Communications Decency Act (aimed at cleaning up the Internet) became a law and was later declared against the constitution. Similar acts have been seen around the world, as governments want to govern the amount of information available to its citizens.[1]

1.1.2 JAVA

Sun was developing in early nineties a prototype hand-held device called *7, a consumer-computer convergence device that was seen to be a portable home. A version of UNIX (running in less that 1 MB) was ported to the device. *7 had a flash memory and an embedded file system was created that would work on it. The device was envisaged to be able to dynamically take new application libraries – new software modules could be loaded on to *7 on-the-fly without having to install programs (true plug-and-play). This would allow *7 to control all the devices at home; user would only plug in required module for each device. Originally, C++ was to be used in the project, but significant difficulties aroused because the language couldn't deliver the requirements for the project. Portability and security were among the problems arousing from the language.
WebRunner, Oak was renamed to Java once introduced in the net in 1995 its popularity has never stopped growing. Curiously, in the demonstration for *7 (later known as Javas) mascot called Duke was waving its hands – and Duke was actually called an agent, that would perform tasks for the user. It is no accident, that Java even now is considered to be the most suitable language to program software agents – the features it offers are closely connected to distributed computing, the environment where agents live.

1.1.3 Agents

The evolution of agents can be divided into two strands: to the study that has been done before nineties and studies after that. In the seventies, Carl Hewitt proposed a concept of *"…self contained, interactive and concurrently executing object…"* that he called "actor". This object had an internal state and could communicate with other similar objects. This work was the base for studies done with multi agent systems that was mainly concerned with macro issues. The aim of the studies was to analyse and specify

systems containing multiple collaborative agents. This approach gives emphasis to society of agents over individual agents. Later issues researched where theoretical issues like architectural and language problems. Although there is inevitably some overlap with the issues researched in the nineties, new type of research has clearly emerged. Previously only research was done on macro level, but now new research has been done on broader range of agent types (or classes). This can also be credited to the fact that more and larger companies have started getting interested in agents – also the term ‘agent’ is being used more and more broadly than before. Nowadays almost every piece of software has some ‘intelligence’ in it – be it mail filtering, adaptive interfaces or help with writing a letter – and especially if the intelligence can be seen as an entity, it’s usually called an agent. It has been predicted, that in few years time most of the consumer products (not just software) will have some kind of embedded agents in them.

1.2 Agent Concepts

The first commercial implementation of the mobile agent concept, General Magic's Telescript technology, attempted to allow automated as well as interactive access to a network of computers using mobile agents. The commercial focus of General Magic technology, the electronic marketplace, requires a network that will let providers and consumers of goods and services find one another and transact business electronically. Although the electronic marketplace still does not exist fully, the Internet has already encouraged its beginnings. Telescript's creators envision the electronic marketplace as only a small piece of the agent world that will exist in coming years. There, agents will act on their user's behalf to research information for work, find the best hotel for vacation, provide up-to-the-minute scores of sporting events, or simply send and receive messages between friends.

Since General Magic's realization of the basic concept of agent architecture through Telescript is an easy example to follow, we'll use it to introduce how an agent can work successfully. Telescript implements the following principal systems associated with remote programming: places, agents, travel, meetings, connections, authorities, and permits.

1.2.1 Places

Agent technology models a network of computers, however large, as a collection of places offering a service to the mobile agents that enter. For an agent, a mainframe computer might function as a shopping center housing, for example, a ticket place where agents can purchase tickets to theater and sporting events, a flower place where agents can order flowers, and a directory place where agents can learn about any place in the shopping center. The network might encompass many independently operated shopping centers, as well as many individually operated shops, many of the latter on personal computers (see figure 1).

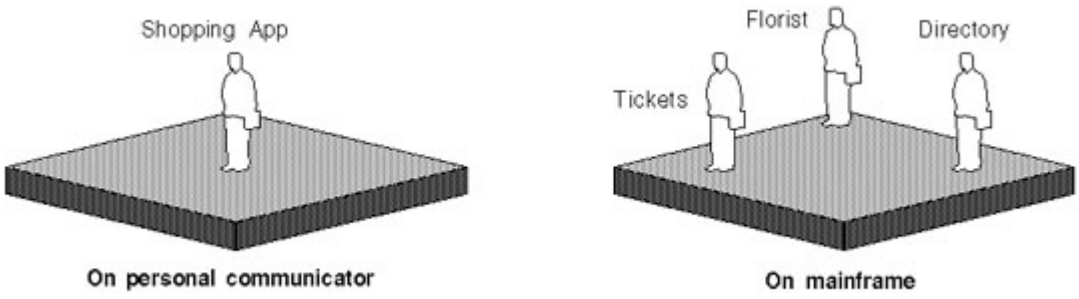


Figure 1: A Network Shopping Center

Servers provide some places and user computers provide others. For example, the home place on a user's personal communicator might serve as the point of departure and return for agents that the user sends to servers.

1.2.2 Agents

Communicating applications are modeled as a collection of agents. Each agent occupies a particular place. However, an agent can move from one place to another, thus occupying different places at different times. Agents are independent in that their procedures are performed concurrently.

The typical place is permanently occupied by one, distinguished agent. This stationary agent represents the place and provides its service. For example, the ticketing agent provides information about events and sells tickets to them, the flower agent provides

information about floral arrangements and arranges for their delivery, and the directory agent provides information about other places, including how to reach them (see figure 2).

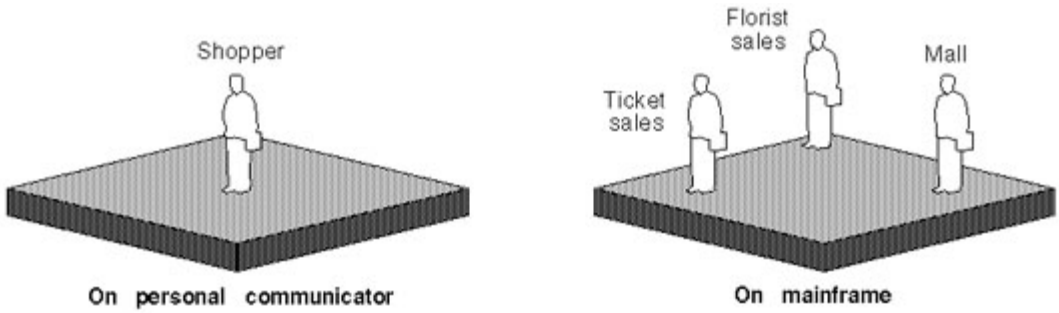


Figure 2: Agents mark their places

1.2.3 Travel

Agents are allowed to travel from one place to another, however distant, the hallmark of a remote programming system. Thus, travel allows an agent to obtain a service offered remotely and then return to its starting place. A user's agent, for example, might travel from home to a ticketing place to obtain orchestra seats for a theater show. Later, the agent might travel home to describe to its user the tickets it obtained (see figure 3).

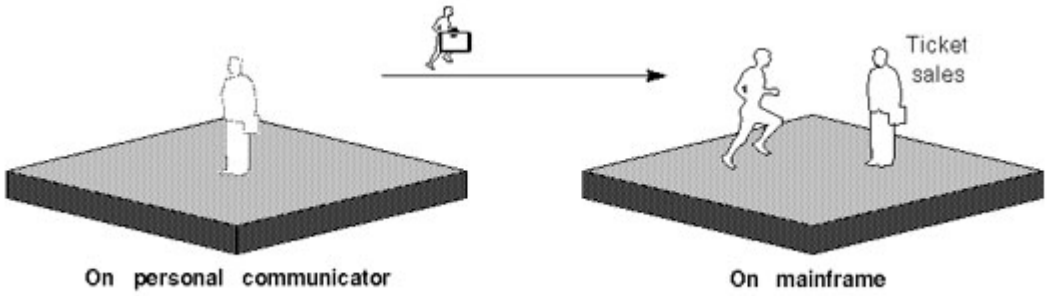


Figure 3: An Agent carries out a mission

Moving software programs between computers using a network has been commonplace for twenty years, or more. In such a case a local area network is employed to download a

program from the file server, where it is stored, to a personal computer, where it runs. Contrary to this process, agents move programs while they run, rather than before. A conventional program written, for example, in C or C++ cannot be moved under these conditions because neither its procedure nor its state is portable. An agent can move from place to place throughout the performance of its procedure because the procedure is written in a language designed to permit this movement. The Telescript language lets a computer package an agent, its procedure and its state, so that the agent can be transported to another computer. The agent itself decides when such transportation is required.

To travel from one place to another, an agent executes Telescript's `go` instruction. The instruction requires a ticket, data that specify the agent's destination and the other terms of the trip (for example, the means by which it must be made and the time by which it must be completed). If the trip cannot be made (for example, because the means of travel cannot be provided or the trip takes too long), the `go` instruction fails and the agent handles the exception as it sees fit. However, if the trip succeeds, the agent finds that its next instruction is executed at its destination. Thus, in effect, the language reduces networking to a single instruction.

This `go` instruction lets the agents of different users co-locate themselves so they can interact efficiently. With this capability an agent can arrive at a single place and either converse with a stationary agent, one that resides only in one place, or, if necessary, converse with one or more agents who have also arrived at the same place.

1.2.4 Meetings

Two agents are allowed to meet if they are in the same place. A meeting lets agents in the same computer call one another's procedures. The Telescript `meet` instruction used to accomplish this allows the co-located agents of users to exchange information and carry out transactions.

Meetings motivate agents to travel. An agent might travel to a place in a server to meet the stationary agent that provides the service the place offers. An agent in pursuit of

theater tickets, for example, may travel to and then meet with a ticket agent. Alternatively, two agents might travel to the same place to meet each other to, say, participate in a venue used for buying and selling used cars (see figure 2.6).

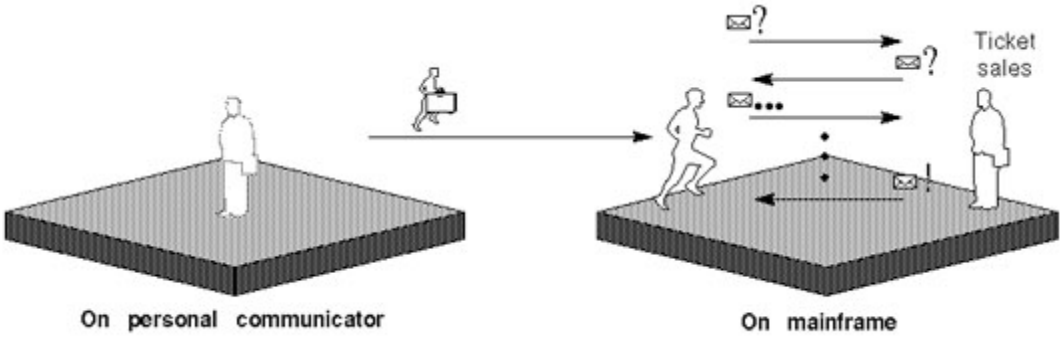


Figure 4: Carrying out a meet instruction

To meet a co-located agent, an agent executes the "meet" instruction. The instruction requires a petition, data that specify the agent to be met and the other terms of the meeting such as the time by which it must begin. If the meeting cannot be arranged (for example, because the agent to be met declines the meeting or arrives too late), the `meet` instruction fails and the agent handles the exception as it sees fit. However, if the meeting occurs, the two agents are placed in programmatic contact with one another.

1.2.5 Connections

A connection, when two agents in different places communicate, is often made for the benefit of the human users of interactive applications. For example, an agent that travels in search of theater tickets might send to an agent at home a diagram of the theater, showing the seats available. The agent at home might present the floor plan to the user, and, in turn, send the locations of the seats the user selects to the agent on the road (see figure 5).

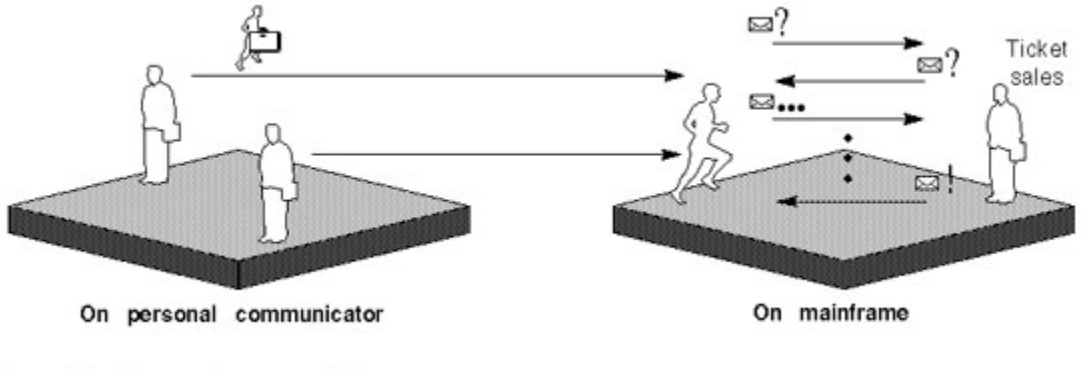


Figure 5: Two Agents communicate

To connect to a distant agent, an agent executes a Telescript `connect` instruction. This instruction requires a target and other data that specify the distant agent, the place where that agent resides, and the other terms of the connection, such as the time by which it must be made and the quality of service it must provide. If the connection cannot be made (for example, because the distant agent declines the connection or is not found in time or the quality of service cannot be provided), the connect instruction fails, and the agent handles the exception as it sees fit. However, if the connection is made, the two agents are granted access to each other (see figure 5).

In the agent world, the `connect` instruction allows the agents of users to exchange information at a distance. Sometimes, as in the theater layout phase of the ticketing example, the two agents that make and use the connection are parts of the same communicating application. In such a situation, the protocol that governs the agents' use of the connection is of concern only to that one application's designer. It need not be standardized.

1.2.6 Authorities

This agent system lets one agent or place discern the authority of another. The authority of an agent or place in the electronic world is the individual or organization in the physical world that it represents. Agents and places can discern, but neither withhold nor falsify their authorities, precluding anonymity.

To control access to its files, a file server must know the authority of any procedure that instructs it to list or delete files. This need, important for any network's security, is the same whether the procedure is stationary or mobile. The system verifies the authority of an agent whenever it travels from one region of the network to another. A region is a collection of places provided by computers that are all operated by the same authority. Unless the source region can prove the agent's authority to the satisfaction of the destination region, the agent is denied entry to the latter. In some situations, highly reliable, cryptographic forms of proof may be demanded (see figure 6).

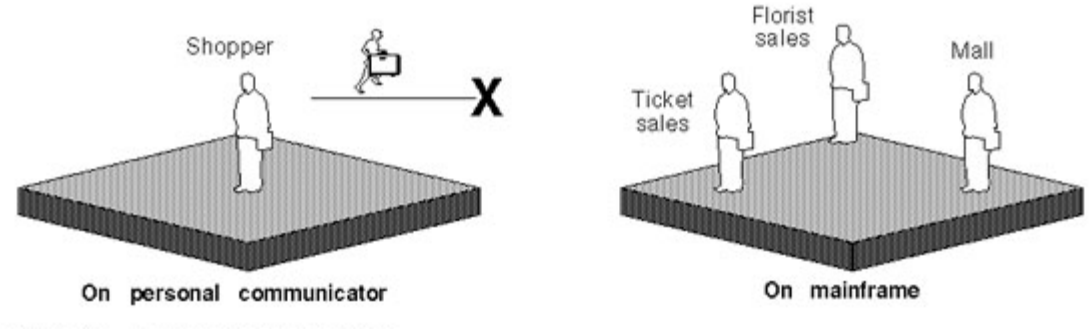


Figure 6: Denying entry to an agent

To determine an agent's or place's authority, an agent or place executes Telescript's `name` instruction. This instruction is applied to an agent or place within reach for one of the reasons discussed below. The result of the instruction is a telename, data that denote the entity's identity as well as its authority. Identities distinguish agents or places with the same authority.

Authorities let agents and places interact with one another on the strength of their ties to the physical world in three different ways. First, a place can discern the authority of any agent that attempts to enter it, and can arrange to admit only agents of certain authorities. Second, an agent can discern the authority of any place it visits, and can arrange to visit only places of certain authorities. Finally, an agent can discern the authority of any agent with which it meets or to which it connects, and can arrange to meet with or connect to only agents of certain authorities.

The `name` instruction can permit programmatic transactions between agents and places which stand for, say, financial transactions between their authorities. A server agent's authority can bill a user agent's authority for services rendered. In addition, the server agent can provide personalized service to the user agent on the basis of its authority, or can deny it service altogether. More fundamentally, the lack of anonymity helps prevent viruses by denying agents that contain the characteristics of a virus.

1.2.7 Permits

Authorities can limit what agents and places can do by assigning permits to them. A permit is data that grants capabilities. An agent or place can discern its capabilities, what it is permitted to do, but cannot increase them.

Permits grant capabilities of two kinds. A permit can grant the right to execute a certain instruction; for example, an agent's permit can give it the right to create other agents. Having done this, the agent must share its allowances of these capabilities and can grant only those capabilities it itself possesses to any agent it creates. An agent or place that tries to exceed any of these qualitative limits is simply prevented from doing so. A permit can also grant the right to use a certain resource in a certain amount. For example, an agent's permit can give it a maximum lifetime in seconds, a maximum size in bytes, or a maximum amount of computation within the limits of its own allowance. An agent or place that tries to exceed one of these quantitative limits is destroyed (see figure 7). An agent can even impose temporary permits upon itself. The agent is notified, rather than destroyed, if it violates one of these temporary permits. With this feature of the Telescript language, an agent can recover from its own misprogramming.

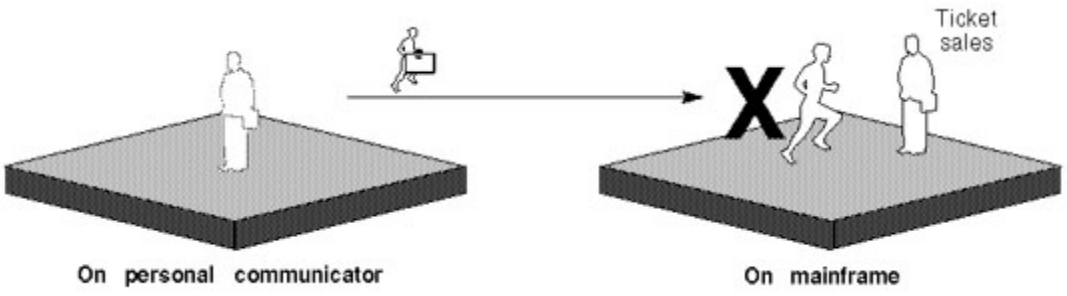


Figure 7: Keeping an agent within limits

To determine either an agent's or a place's permit, an agent or place executes Telescript's `permit` instruction.

Permits protect authorities by limiting the effects of errant and malicious agents and places. Such a rogue agent threatens not only its own authority, but also those of the place and region it occupies. For this reason, the technology allows each of these three authorities to assign an agent a permit. The agent can exercise a particular capability only to the extent that all three of its permits grant that capability. Thus, an agent's effective permit is renegotiated whenever the agent travels. To enter another place or region the agent must agree to its restrictions. When the agent exits that place or region, its restrictions are lifted, but those of another place or region are imposed.

The `permit` instruction and the capabilities it documents help to guard against the unbridled consumption of resources by ill-programmed or ill-intentioned agents. Such protection is important because agents typically operate unattended in servers rather than in user computers where their misdeeds might be more readily apparent to the human user.

1.2.8 Putting things together

An agent's travel is not restricted to a single round-trip. The power of mobile agents becomes fully apparent when one considers that an agent may travel to several places in

succession. Using the basic services of the places it visits, such an agent can provide a higher-level, composite service.

Recalling our ticketing example from above, traveling to the ticket place might be only the first of the agent's responsibilities. The second might be to travel to the flower place and there arrange for a dozen roses to be delivered to the user's companion on the day of the theater event. Note that the agent's interaction with the ticket agent can influence its interaction with the flower agent (see figure 8). For example, if instructed to get tickets for any available evening performance, the agent can order flowers for delivery on the day for which it obtains tickets.

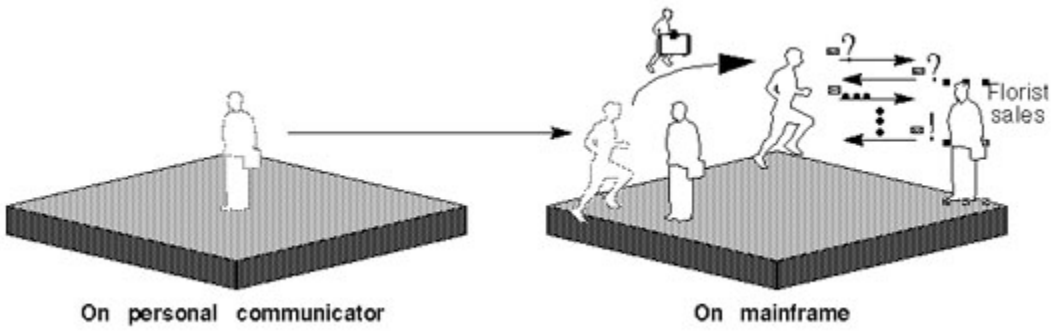


Figure 8: Interactions Influence Outcome

This simple example has far-reaching implications. The agent fashions from the concepts of tickets and flowers the concept of special occasions. While the agent does this for the benefit of an individual user in our example, a variation of the example suggests that the agent could also take up residence in a server and offer its special-occasion service to other agents. Thus agents can extend the functionality of the network, conveying the sense that the network is also a platform.

Next we take a look at the two different methods for implementing agents, and evaluate their possible uses for implementing an Electronic Market Place.

Chapter 2
The FIPA Architecture

Overview

Although the concept of a FIPA mobile agent itself is not very new, the formal specification is. FIPA 97 does not comment on it, FIPA 98 describes the basic framework and FIPA 99 adds new concepts. There has been lots of research in the area, but the mobility specification is still very much under development. This projects purpose is to concentrate on the theoretical side of agents and implement an example of a (FIPA compliant) agent.

2.1 High-level Architecture

FIPA-OS is a component-orientated toolkit for constructing FIPA compliant Agents using mandatory components (i.e. components required by ALL FIPA-OS Agents to execute), components with switchable implementations, and optional components (i.e. components that a FIPA-OS Agent can optionally use). Figure 9 highlights the available components and there relationship with each other (NOTE: The Planner Scheduler is not currently available).

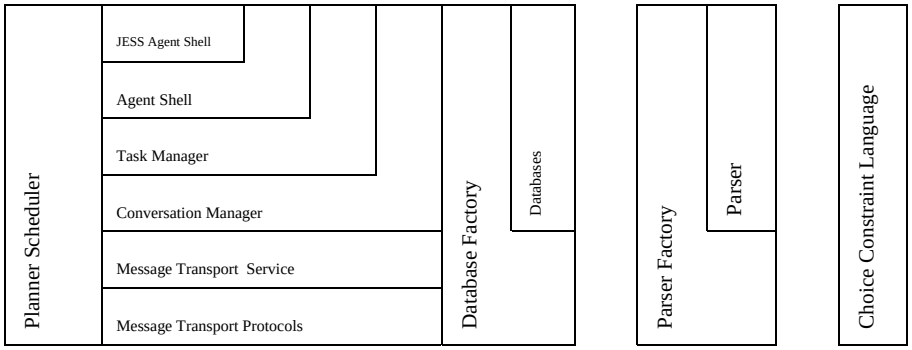


Figure 9: Components within FIPA-OS

The Database Factory, Parser Factory and CCL components are optional and do not have an explicit relationship with the other components within the tool-kit. The Planner Scheduler generally has the ability to interact with all components of an Agent, although

not necessarily vice versa. The switchable implementations included as part of the FIPA-OS distribution for each component include:

- MTP's
    - RMI (proprietary)
    - IIOP (FIPA compliant)
- Database's
    - MemoryDatabase
    - SerializationDatabase
- Parser's
    - SL
    - ACL
    - XML
    - RDF

Multi-Agent Systems (MAS) consists of many agents that can combine their abilities to solve problems. Due to the collaborative nature of MAS, agent standards play an important role for commercialisation of agent technology. FIPA (Foundation for Intelligent Physical Agents), a non-profit organisation for producing standards for open agent interfaces, has produced several specifications tackling different aspects of MAS. These specifications don't try to dictate internal architectures of agents or how they should be implemented, but they specify the interfaces necessary to support interoperability between different MAS. FIPA identifies four areas for standardisation:

2.1.1. Agent Communication Interface

This describes the communication between agents and it supports all interactions between two agents. FIPA has specified an Agent Communication Language (ACL) to support the interface (ACL is based on Knowledge Querying and Communication Language KQML). ACL has five levels of formal semantics:

- *Protocol* – defines the structure of the agent dialogue, like fipa-request-protocol.

- *Communicative Act (CA)* – defines the type of communication currently performed, like “request” when an agent is requesting a service from another agent.
- *Messaging* – defines meta-information of the message, like identity of the sender and receiver.
- *Content Language* – defines the language (i.e. the grammar) of the content message, like XML.
- *Ontology* – defines the meaning of terms and concepts used in content expression like meaning of the XML tags.

2.1.2. Agent Management

This describes facilities necessary to support the creation of agents, communication between agents, as well as security and mobility. *FIPA 97 defines the platform to be an infrastructure in which agents can be deployed and where FIPA agents can enter, advertise their services, locate other agents and communicate with agents of other platforms.* It consists of three agents – often called the platform agents – ACC 5 , AMS and default DF:

- *Directory Facilitator (DF)* – DF is an agent that provides “yellow pages” services to other agents, where agents can register their services and request information of other agents.

- *Agent Management System (AMS)* – AMS is an agent that provides an agent name service, an index of all the agents currently registered in the platform. AMS makes sure that all the agents have unique names, Agent Global Identifiers (GUIDs). AMS has supervisory power on the platform and it can create, delete and de-register agents and it oversees the migration of the agents to and from other platforms.

- *Agent Communication Channel (ACC)* – ACC is an agent that routes messages between agent platforms and it must minimally support Internet Inter-Orb Protocol (IIOP).

2.1.3. Agent/Software Integration Interface

This interface supports the interaction between agents and non-agent software. A concept called "wrapper agent" is introduced where an agent wraps itself on to a piece of software/hardware and acts as an agent representative of that piece.

2.1.4. Agent/Human Interaction Interface

This interface describes how agents can interact with human users providing agents with user models.

2.2 Agent Lifecycle

FIPA agent lifecycle defines that the agent can be in three different states in its lifetime (represented in the AMS): active, waiting and suspended. Mobility support specification defines one more state to support the mobility (transit), and two actions to enter and leave the state (move and execute).

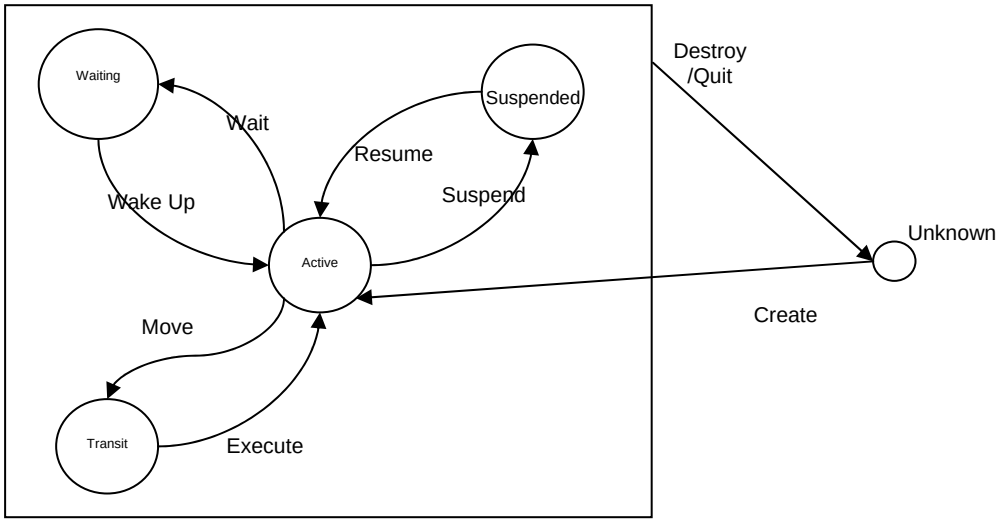


Figure 10: Possible Agent States

2.3.1. Non-Component Core Classes

This section aims to briefly look at the classes that any non-trivial Agent implementation will make use of, but are not necessarily part of any particular component.

2.3.1.1. fipaos.ont.fipa.ACL

This class represents the abstract notion of an ACL message within the FIPA ACL specifications . By default it supports parsing and deparsing of the FIPA standard string encoding for ACL , although this is for historical reasons (ideally the parsing/deparsing of stringified representations of objects should be independent of the objects containing that information – this provides scope for multiple content language representations to be considered for a particular class). In versions of FIPA-OS prior to v1.3.0, the ACLMessage class was used for the same purpose – the ACL class was introduced due to the changes between FIPA97/98 specifications and FIPA2000, and the introduction of better typing (i.e. ACLMessage uses String's to represent GUID's/AID's, whereas the ACL class requires concrete AgentID objects). In order to ensure a degree of backward compatibility, the ACLMessage class is still currently bundled with FIPA-OS, although it simply wraps the ACL class (see Figure 11).

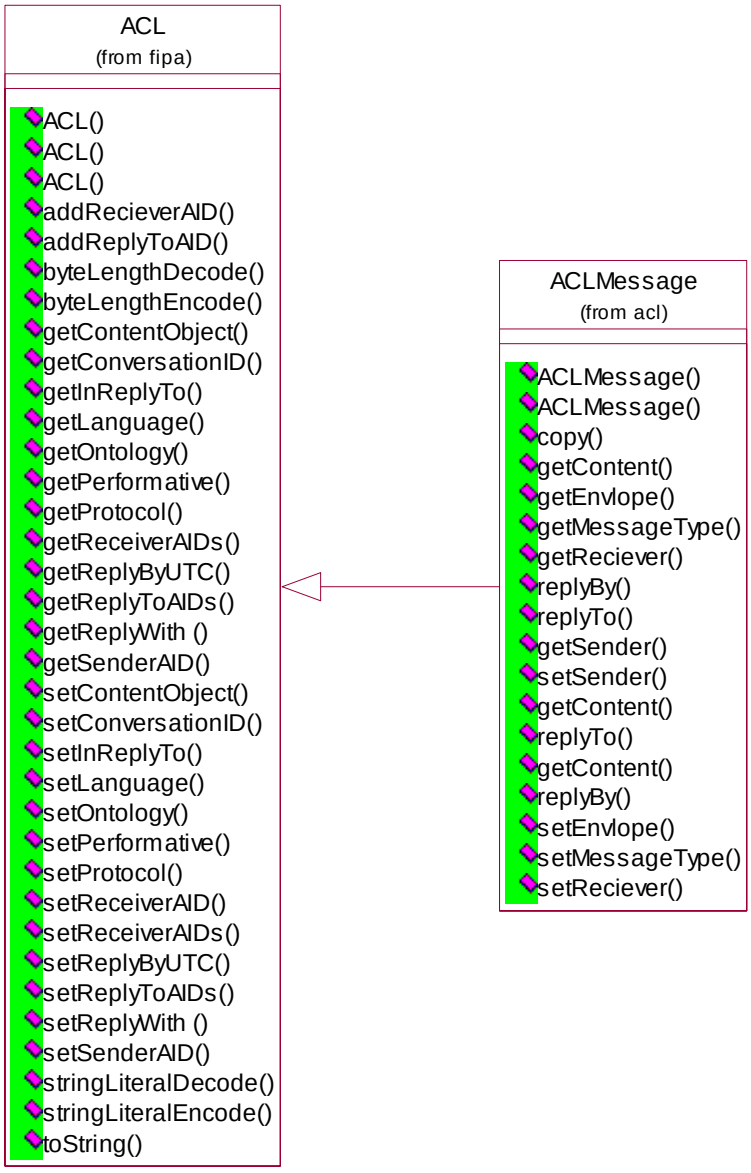


Figure 11: ACL & ACLMessage Objects

2.3.1.2fipaos.ont.fipa.fipaman.Envelope

This class provides an abstract representation of the FIPA defined Envelope from the MTS specification . An Envelope object by default provides access to the last assigned values of eachof its parameters. Changes made to the Envelope by each ACC can be

inspected by using the getSubEnvelopes() method it provides, which returns a List of Envelope objects.

2.3.1.3fipaos.mts.Message

The Message class is a convenience class that contains references to an Envelope and ACL object, the two components that make up a message within the MTS (Message Transport Service).

2.3.1.4fipaos.util.DIAGNOSTICS

This class provides a standardised API for printing debugging messages to screen and to a file, allowing levels to be assigned to each message. This enables the level of detail in debugging messages displayed/recorded to be controlled at runtime. The static println() methods defined by this class are the recommended mechanism for displaying debugging information for the following reasons:

- Controllable level of detail on a per-message basis (as mentioned above)
- All debugging information can be logged to a file, at a different detail level to that displayed on-screen
- Display/writing of debug messages is completely decoupled from code calling println() methods via a buffer, increasing application speed compared to System.out.println(), which blocks until the text is displayed on some operating systems (notably Windows).

2.3.2 Agent Shell (FIPAOSAgent)

The FIPAOSAgent class provides a shell for Agent implementation to use by simply extending this class.

2.3.7. Composition of an Agent

The FIPAOSAgent shell is responsible for loading an Agent's profile, and initialising the other components of which the Agent is composed. It creates these mandatory components in this order initially:

- MTS
- Task Manager
- Conversation Manager

At initialization of the Conversation Manager, references to the MTS and Task Manager are passed to enable them to be dynamically bound to the CM. This is all achieved via the listener interfaces implemented by the various components, so these components are not explicitly dependant on each other. Figure 12 highlights the relationships between the classes of these core components, and the interfaces used to remove inter-component dependence.

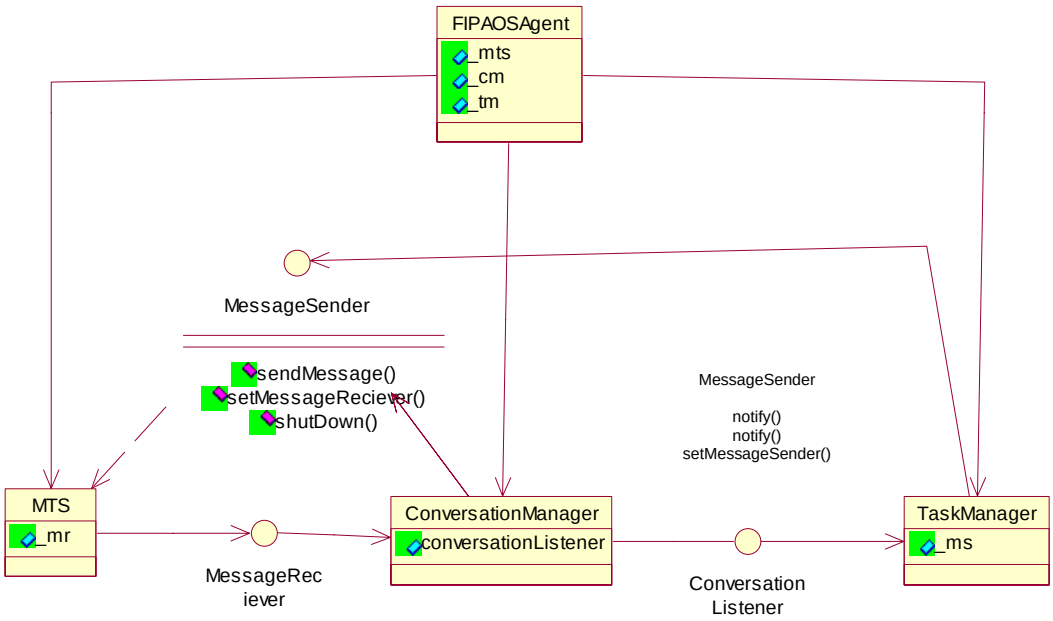


Figure 12: Core Component Relationships within FIPAOSAgent / Agent Shell

New core components could simply be added by implementing the required interfaces and passing references to the new class at construction-time to the existing components. As can be seen, the interfaces defined are also inter-related since they allow registration of other listener interfaces with implementation objects:

- ConversationListener – Implementing classes are generally interested with receiving Conversation object updates from another object. Provides a method to register a MessageSender with the underlying implementation, providing a dynamic mechanism for binding a component that can send messages.
- MessageReceiver – Implementing classes are interesting in receiving "raw" messages one at a time. Provides a method to register a MessageSender with the underlying implementation also.
- MessageSender – Implementing classes provide a direct or indirect (i.e. they pass messages to another MessageSender implementation) mechanism for sending ACL messages. Provides a method to register a MessageReceiver with the underlying implementation, providing a dynamic mechanism for binding a component that should receive incoming messages. This provides a flexible mechanism to allow the core-components to register with one another once they have been constructed, without encountering the "chicken-and-egg" problem of which component should be constructed first when references to it need to be passed to other components and vice versa. Each component has an implicit reference to the FIPAOSAgent class to which they belong.

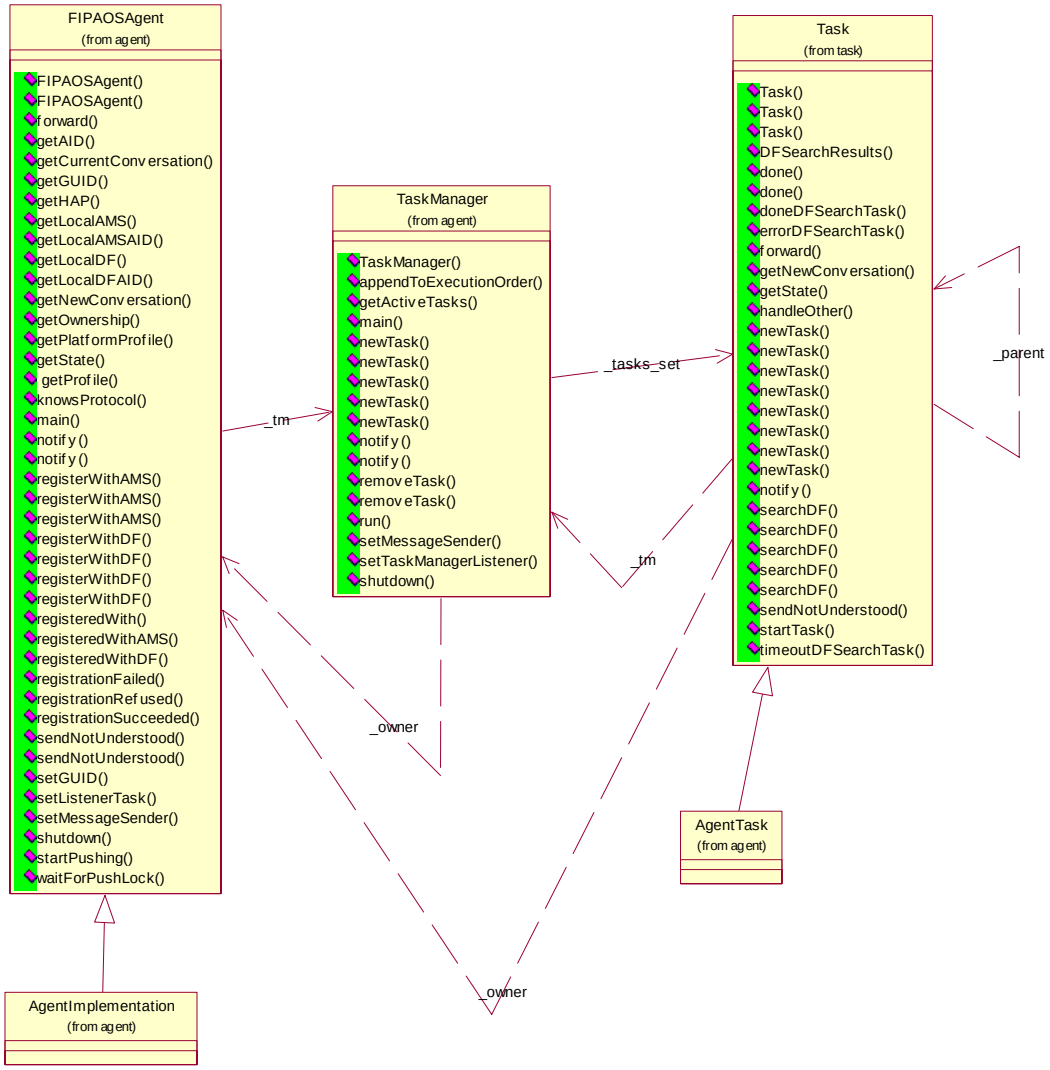


Figure 13: Agent Implementation and Relationship with Agent Shell

Figure 13 highlights how an Agent implementation relates to some of the components of the Agent Shell. Generally an Agent consists of a class that extends the FIPAOSAgent class, and a number of Task implementations that contain the functionality of an Agent.

2.3.2.2Functionality Provided by the Agent Shell

The Agent Shell provides the following functionality:

- Sending messages – This is accomplished by using the forward() method in either the FIPAOSAgent or Task class, depending on where in an Agent implementation the message is being sent from. In the former case, the outgoing message is always passed to the CM via its sendMessage() method. See the Task Manager and Conversation Manager sections for details on how messages are dealt with.
- Retrieving the Agents' properties (Profiles, AID, state) & Locating platform Agents (DF and AMS) – numerous methods are provided to access this information from the FIPAOSAgent class.
- Registration with platform Agents – The FIPAOSAgent class provides registerWithAMS() and registerWithDF() methods, as well as the call-back methods registrationSucceeded(), registrationFailed() and registrationRefused()which should be overridden. This functionality is provided by use of the AMSRegistrationTask and DFRegistrationTask's 1 . Figure 14 highlights how the Agent Shell creates a AMSRegistrationTask to register with the AMS, and a callback is made to indicate the result of that registration (NOTE: this is only a logical representation of interactions, and doesn't reflect the concrete interactions that occur). Reception of incoming messages from the AMS by the TaskManager is implicit. A similar set of interactions occur when registering with the DF.

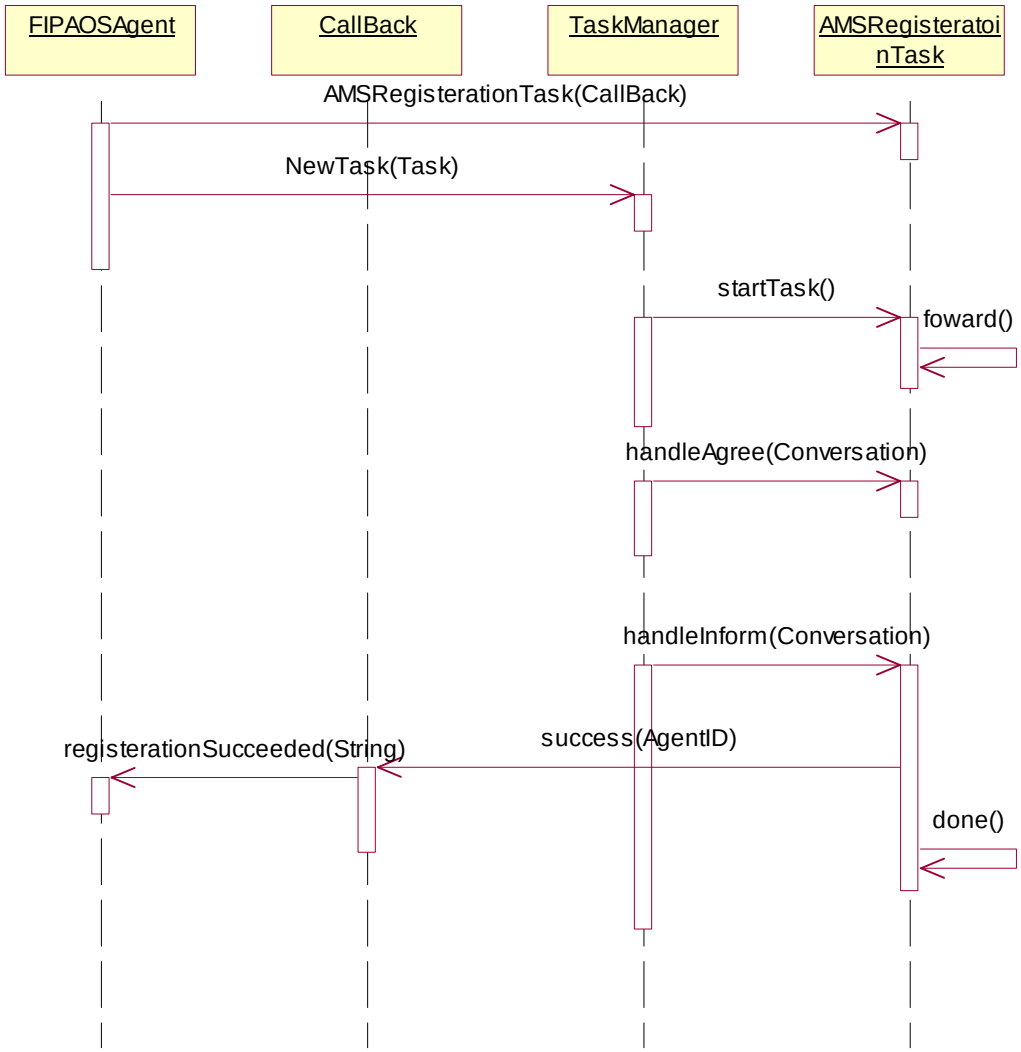


Figure 14: Logical Interactions when Successfully Registering with AMS

- Setting up Task's – The FIPAOSAgent class provides access to the _tm variable, enabling direct access to the TaskManager class & its associated newTask() methods. The Task class provides newTask() methods within its API, which allow access to the same functionality as provided directly via the TaskManager class.
- Shutting down the Agent – The Agent and its components can be cleanly shutdown by invoking the shutdown() method in the FIPAOSAgent class. This in-turn invokes the shutdown() method on all of the components of the Agent.

2.4. TM (Task Manager)

The Task Manager provides the ability to split the functionality of an Agent into smaller, disjoint units of works known as Tasks. The aim is that Task's are self-contained pieces of code that carry out some task and (optionally) return a result, have the ability to send and receive messages, and have little or preferably no dependence on the Agent they are executed within. This provides a number of benefits: These classes are not part of the FIPA-OSv1.3.2 distribution, but are available separately from our SourceForge CVS repository.

- Tasks are highly re-usable - they can be used in many Agents without having to re-write the same code / functionality.
- Easy to debug, since tracking the flow of control is simple (Task's are completely event-based) and useful debugging messages help to indicate when task-interactions fail/are unhandled.
- An Agent can execute multiple Tasks at once – the Task Manager takes care of routing incoming messages and other events to the right Tasks, rather than using a "cludge" of code within the Agent itself to decide what to do with a particular message.
- Conversation state is effectively encapsulated within a Task, reducing the manual tracking of Conversations to a bare minimum.
- Tasks can spawn child-tasks – this enables complex Task's to be created through simply utilising simpler Task within them.

2.4.1. Composition of the TM

The TaskManager itself is composed of several parts, depicted in Figure 15.

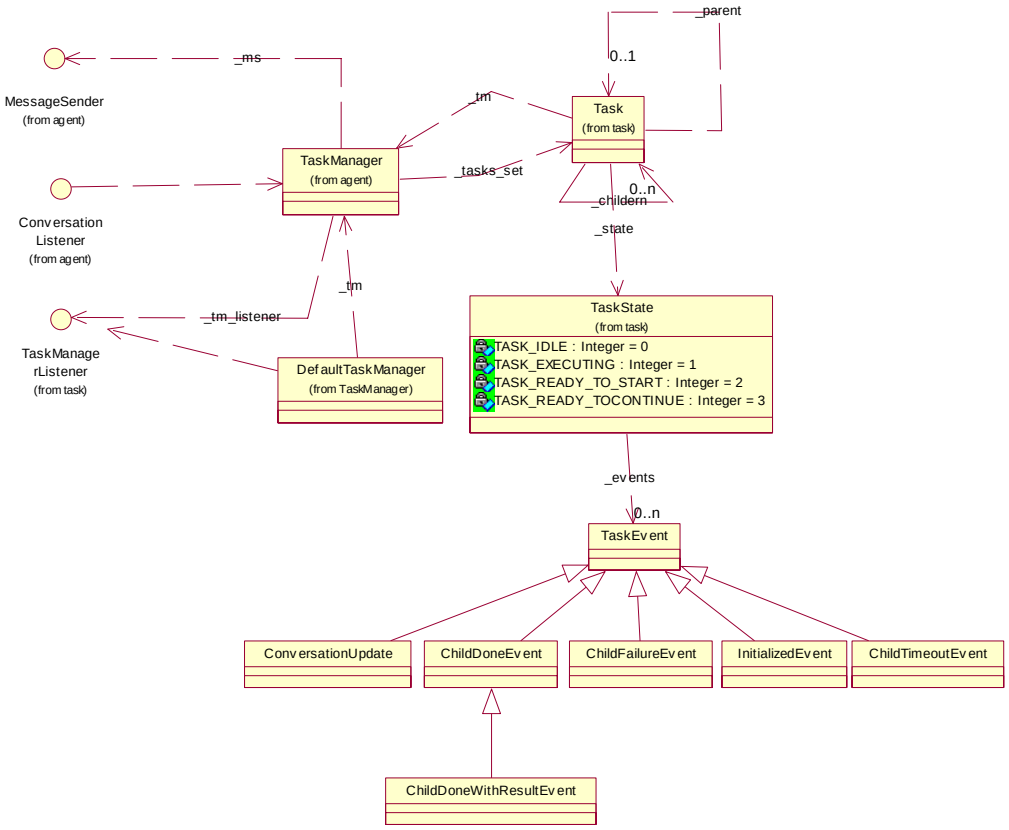


Figure 15: TaskManager Class Relationships

The TaskManager class provides the coordination mechanism for Task‘s within an Agent. All active Task’s are referenced from the _tasks_set of the TaskManager object. The TaskManager also has references to a MessageSender to enable sending of messages from the TaskManager, and implements the ConversationListener interface so that it can be informed of conversation-updates.

2.4.2. Task Events

The entire Task Manager component is built around event-based processing. Every Task within the TM has a queue of pending events of type TaskEvent. The TaskManager generally processes these events in the order they are generated for a particular Task. The TaskEvent’s currently handled are listed in Table 1, along with the listener methods invoked when they are delivered to a Task.

Event	Listener	TaskEvent Listener Method Description
InitialisedEvent	public void startTask()	Indicates that a Task has been initialised and is ready to start (i.e. its startTask()method should be invoked – sub-classes should override this method, which has a default implementation that does nothing).
ConversationUpdate Event	public void handleX(Conve rsation)	Indicates that a new message that is part of a conversation that the Task is involved in has been received, and needs to be dealt with. This will cause a method with the given signature to be invoked, where X is the performative of the last message in the conversation received. If such a method does not exist within the Task, the handleOther() method will be invoked, which has a default implementation that sends a not-understood in response to the last message (implicitly ending the conversation).
ChildDoneEvent	public void doneX(Task)	Indicates when a child-Task completes This will cause a method of the given signature to be invoked on the Task, where X is the name of the child-Task (by default. this is the classname of the

		child-Task). If no such method exists, a warning message will be printed at the maximum DIAGNOSTICS level.
ChildDoneWithResultEvent	public void doneX(Object)	Indicates when a child-Task completes, and has produced a result. This causes a method of the given signature to be invoked in the same manner as for the ChildDoneEvent, except the result object is passed as an argument.
ChildTimeoutEvent	public void timeoutX(Task)	Indicates that a child-Task timed-out before it had a chance to complete. This causes a method of the given signature to be invoked, where X is the name of the child-Task (by default this is the classname of the child- Task). If no such method exists, a warning message will be printed at the maximum DIAGNOSTICS level.
ChildFailureEvent	public void errorX(Task, Throwable)	Indicates that a child-Task failed (i.e. threw an un-caught exception) whilst processing a TaskEvent for it. This causes a method of the given signature to be invoked, where X is the name of the child-Task (by default this is the classname of the child-Task). If no such method exists, a warning message will be printed at the

		maximum DIAGNOSTICS level.

Table 1:TaskManager Event Types

The TaskManager decides when to pass the event to the receiving Task based upon the current state of the Task (encapsulated by the TaskState class), and the order it is instructed to deal with the Task's which have pending events.

2.4.3. Task Manager Listener

In order to support the ability for TaskEvent's (and therefore the execution of Task's) to be scheduled by some external component 2 , the TaskManager doesn't directly decide in which order to deal with Task's. Figure 16 highlights the interactions between a TaskManager and a TaskManagerListener when newTask() is invoked.

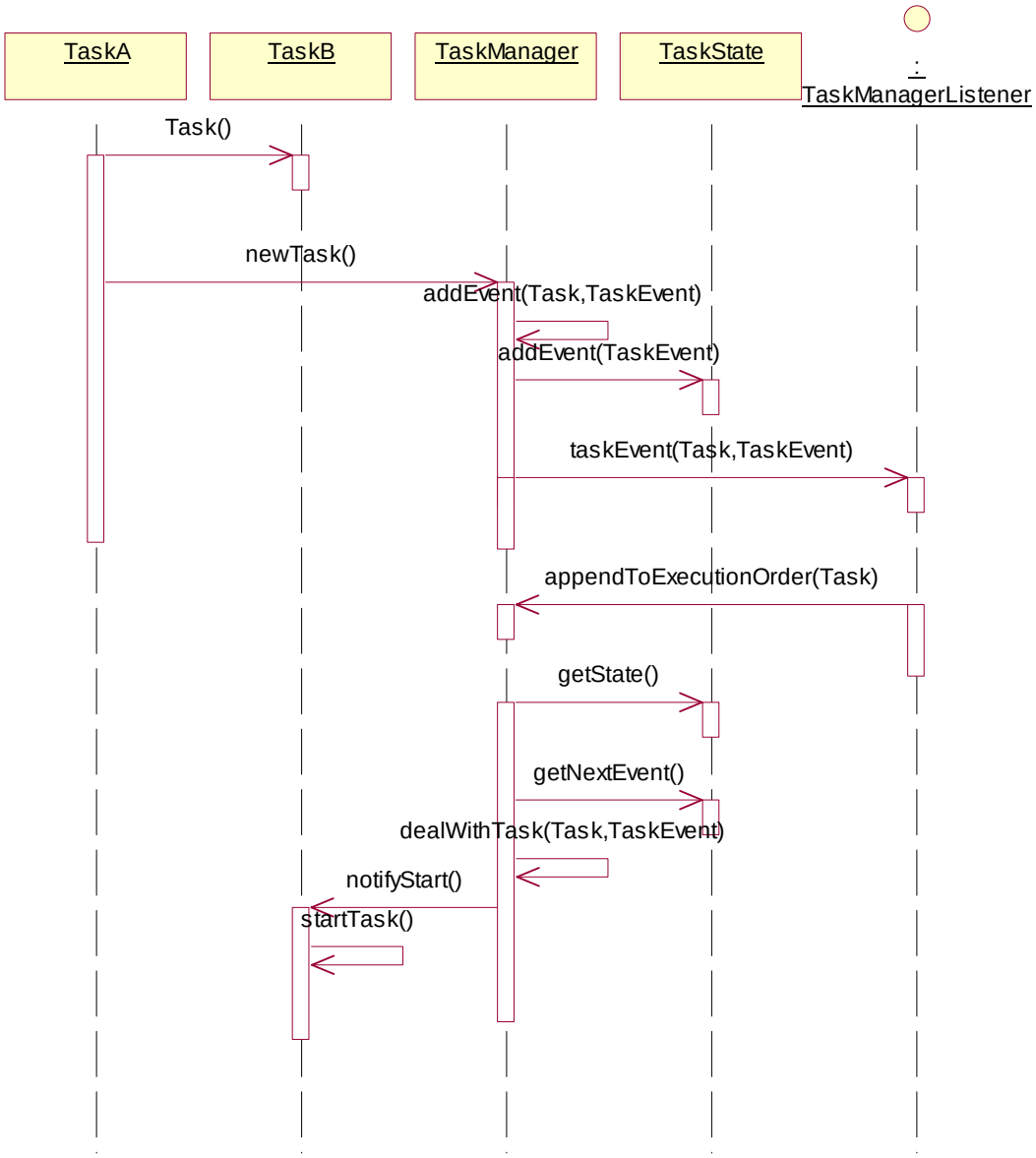


Figure 16: newTask() - TaskManager and TaskManagerListener Concrete Interactions

Whenever a new TaskEvent is generated, it is passed to the registered TaskManagerListener. The particular implementation behind this interface can then instruct the TaskManager in which order to execute the Task's it has with pending events (NOTE: It cannot instruct the TaskManager as to which events to deliver, since we wish to ensure events arrive at a Task in the order they occur 2 For example, the new Planner

Scheduler when it is ready for release. with regard to that Task). A default implementation of the TaskManagerListener interface is provided, the DefaultTaskManagerListener class – this simply allows Task's to be executed in the order events arrive for those Task's.

2.4.4. Starting a Task

Whenever a Task is created, it should be registered as a top-level task via a newTask() method of the TaskManager, or via a newTask() method of another Task. In the later case, the new Task is registered as a child-Task of the other Task, and thus the _parent field of the new Task references the other Task, and the _children Set of the other Task contains a reference to the new Task.

NOTE: The TaskManager initialises a number of instance variables in the Task when newTask() is invoked – the behaviour of methods defined by the Task class are only defined AFTER the newTask() method has returned, hence code in the Task's constructor should NOT utilise any methods from the Task API**.**

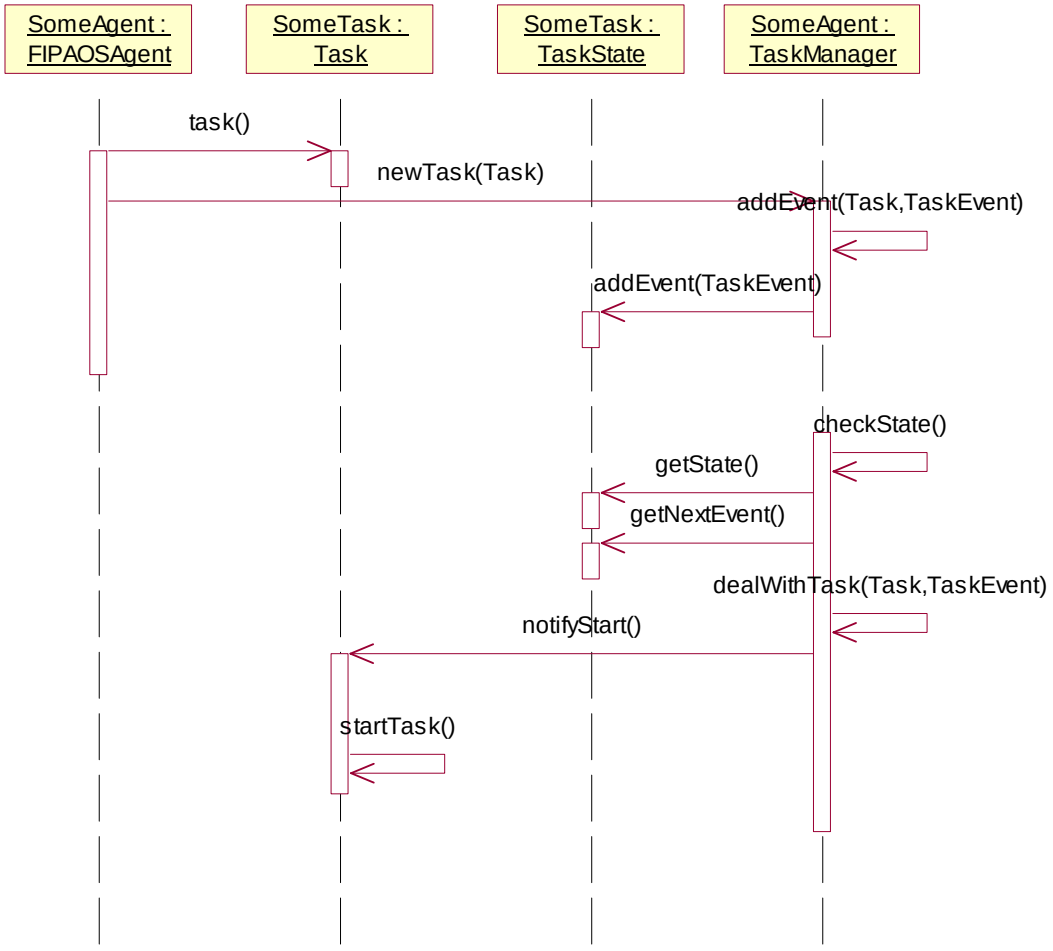


Figure 17: FIPAOSAgent / Task / TaskManager newTask() Concrete Interactions

As shown in Figure 17, once a Task has been initialised, from a Agent developers point of view the startTask() method will be invoked – it is advisable given the above to start any processing for the action to be carried out to occur within this method. To enable this to happen, an InitialisedEvent is generated and added to the queue of TaskEvent's within the Task's TaskState object. When the TaskManager eventually comes to deal with this TaskEvent, the startTask() method is invoked on the Task. There are also a number of alternative ways to use newTask() to start a Task. Other than the ability to relate a conversation with a Task, a time-out can be specified for a Task, at which point its parent-Task will be informed. Figure 18 highlights the interactions between the parent-

Task, TaskManager and child-Task when a Task is started with a timeout, and that timeout is reached.

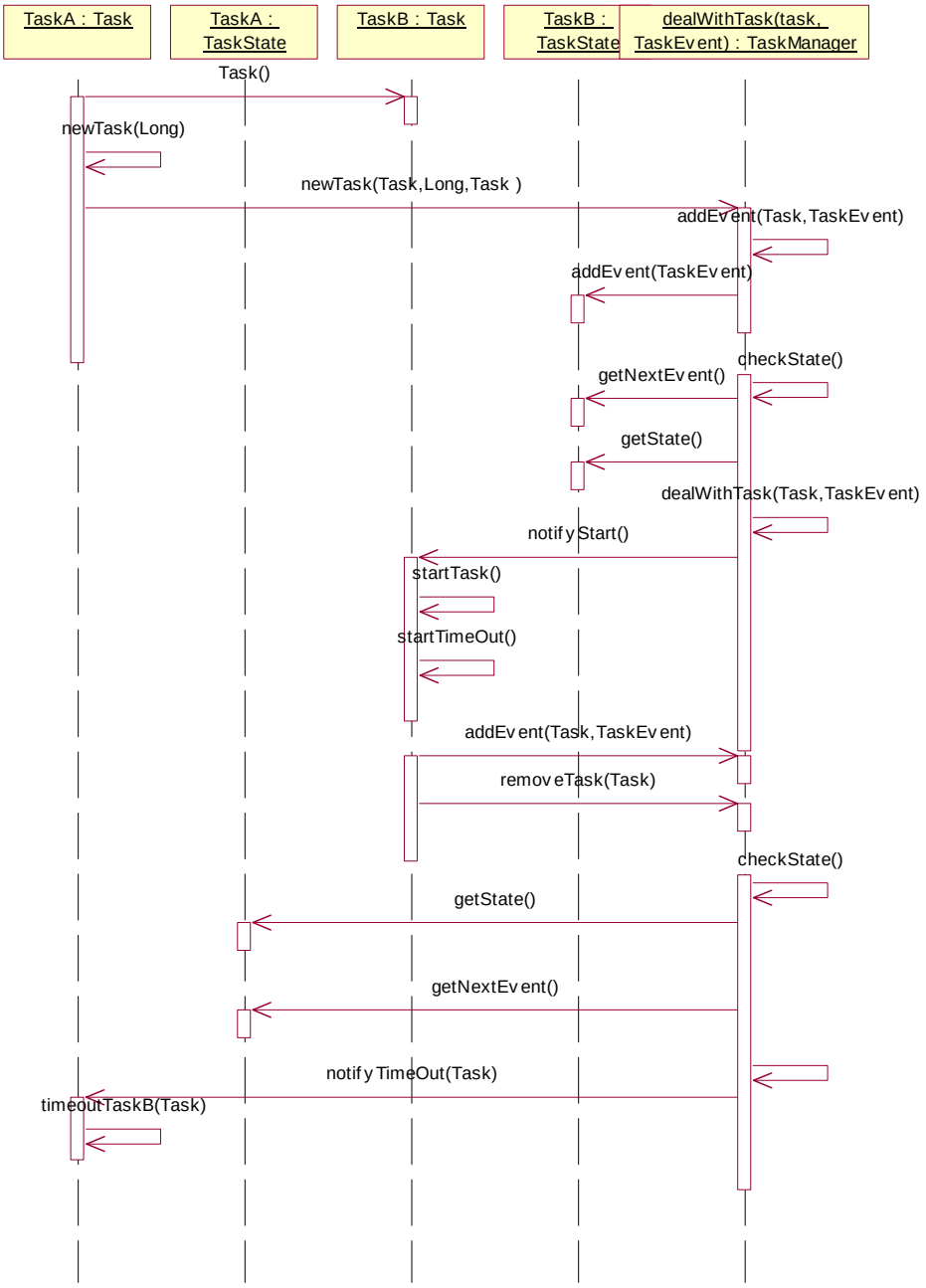


Figure 18: Task / TaskManager Concrete Interactions when Timeouts Occur

2.4.5. Parent-Task and Child-Task Communication

In order for Task's to be able to interact together, a number of simple communication events have been produced for use within the TaskManager. As described previously, these events allow a parent-Task to be informed when one of its child-Tasks completes, times-out or fails. The aim is that these simple events provide the basis for allowing Task's to interact together without creating explicit dependencies between them. Figure 10 highlights the logical interactions between a number of parent/child-Task's. As described previously, startTask() is invoked after newTask() has been invoked on a Task, and doneX() is automatically invoked on the parent after a child-Task invokes either of done() or done(Object).

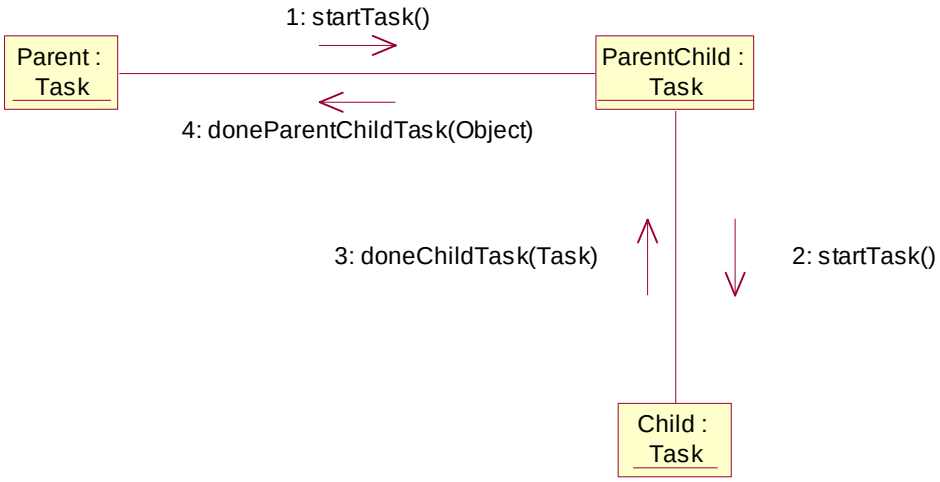


Figure 19: Multiple Nested Parent / Child Completion Logical Interactions

2.4.6. Task Messaging

Task's enable multiple conversations to be conducted simultaneously without an explicit need to track conversation state. As per the FIPAOSAgent class, a forward() method is provided as part of the Task API to enable Task's to send messages, and acts as per Figure 11. The TaskManager has areference to a MessageSender, to which it passes all outgoing messages via its sendMessage() method.

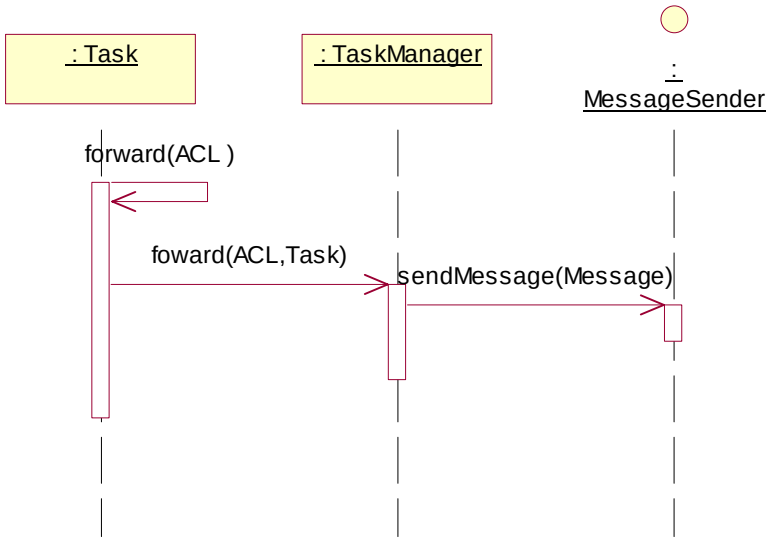


Figure 20: Concrete Interactions when Forwarding a Message from a Task

Whenever a Task sends a message, the conversation the message is part of is automatically bound to that Task (even if no explicit conversation id is provided, the TM ensures one is created) – this ensures that any subsequent messages received which form part of that conversation are passed to that Task. Figure 12 highlights the interactions between the TaskManager and a Task when receiving an incoming message – since the TaskManager implements the ConversationListener interface, it is notified of conversation updates via the notify() method. The binding between Task and Conversation can also change – if one Task starts a conversation, another can continue it by simply sending a message as part of that conversation, or by being initialised using a suitable newTask() method.

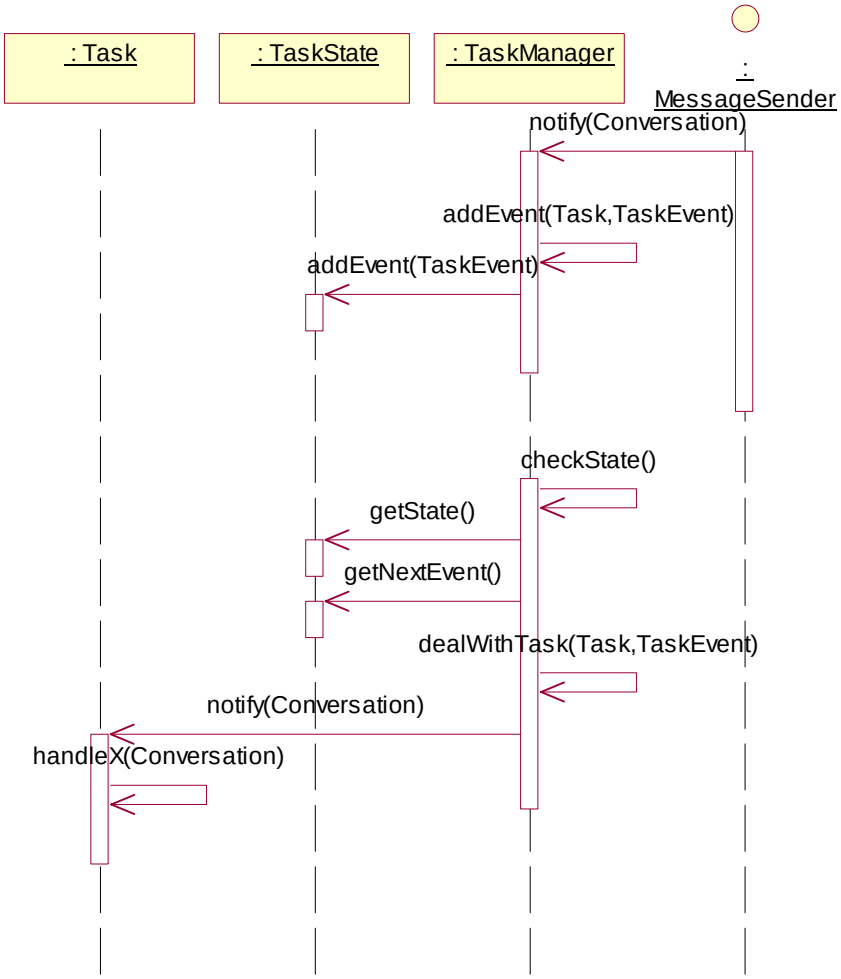


Figure 21: Concrete Interactions when TaskManager Receives a Message

In the event that no Task is bound to the conversation of an incoming message, a default Task should be provided to deal with it (this is achieved by invoking the setListenerTask(Task) method of FIPAOSAgent) – this is generally only the case with new incoming conversations, hence a new Task should be spawned to deal with the interactions with the other Agent.

2.4.7. Other Useful Task API Methods & Fields

The Task class also provides a number of other useful methods for use by sub-classes.

- searchDF() – a variety of searchDF() methods are provided to initiate a search on either a local or remote DF. This is achieved through automated use of the DFSearchTask, which results in the DFSearchResults(DFAgentDescription[]) method being invoked on the initiating Task once the DFSearchTask has completed (see Figure 13). This is achieved through default implementations of doneDFSearchTask() and errorDFSearchTask() methods in the Task super-class, so care should be taken if these methods are overridden.

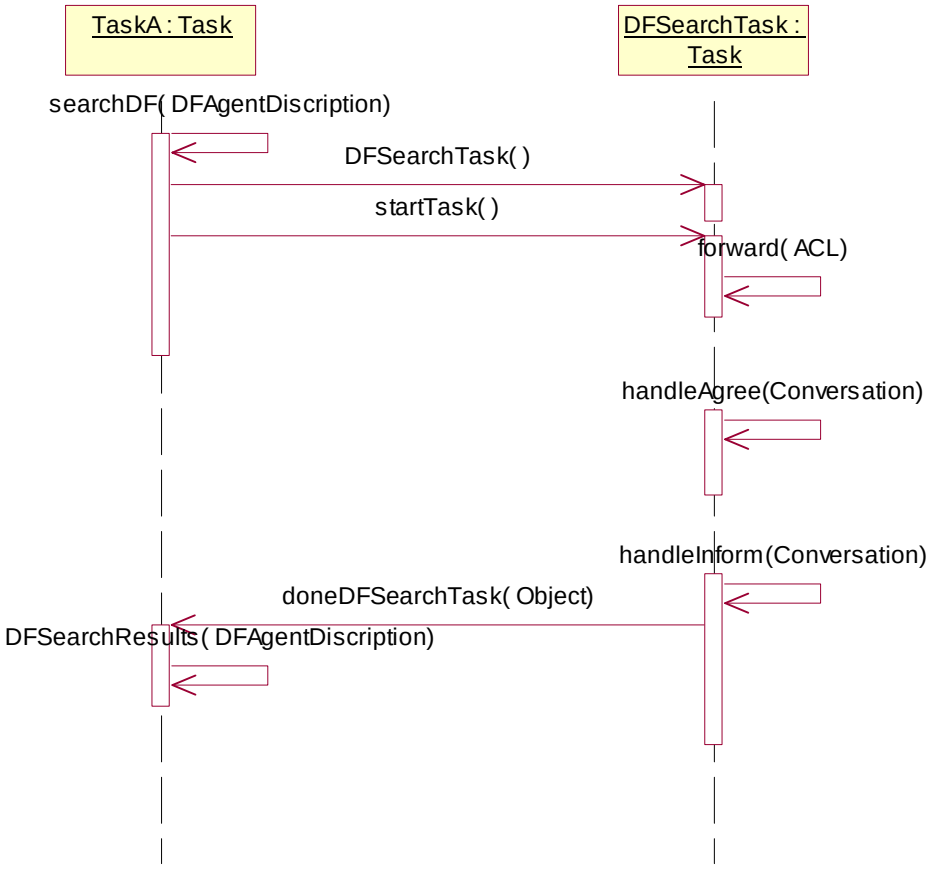


Figure 22: Logical searchDF() Interactions

- sendNotUnderstood() – provides a convenience mechanism for replying to a message with a not-understood.

- getNewConversation() – another convenience method for creating a new conversation which is bound to this Task.
- _tm – reference to the TaskManager that manages the Task.
- _owner – reference to the FIPAOSAgent that owns the Task.

2.5. CM (Conversation Manager)

The CM provides the ability to track conversation state at the performative level, as well as mechanisms for grouping messages of the same conversation together. If a conversation is specified as following a particular protocol, the CM will ensure that the protocol is being followed by both the Agent it is part of, and the other Agent involved in the conversation.

2.6.1. Composition of the CM

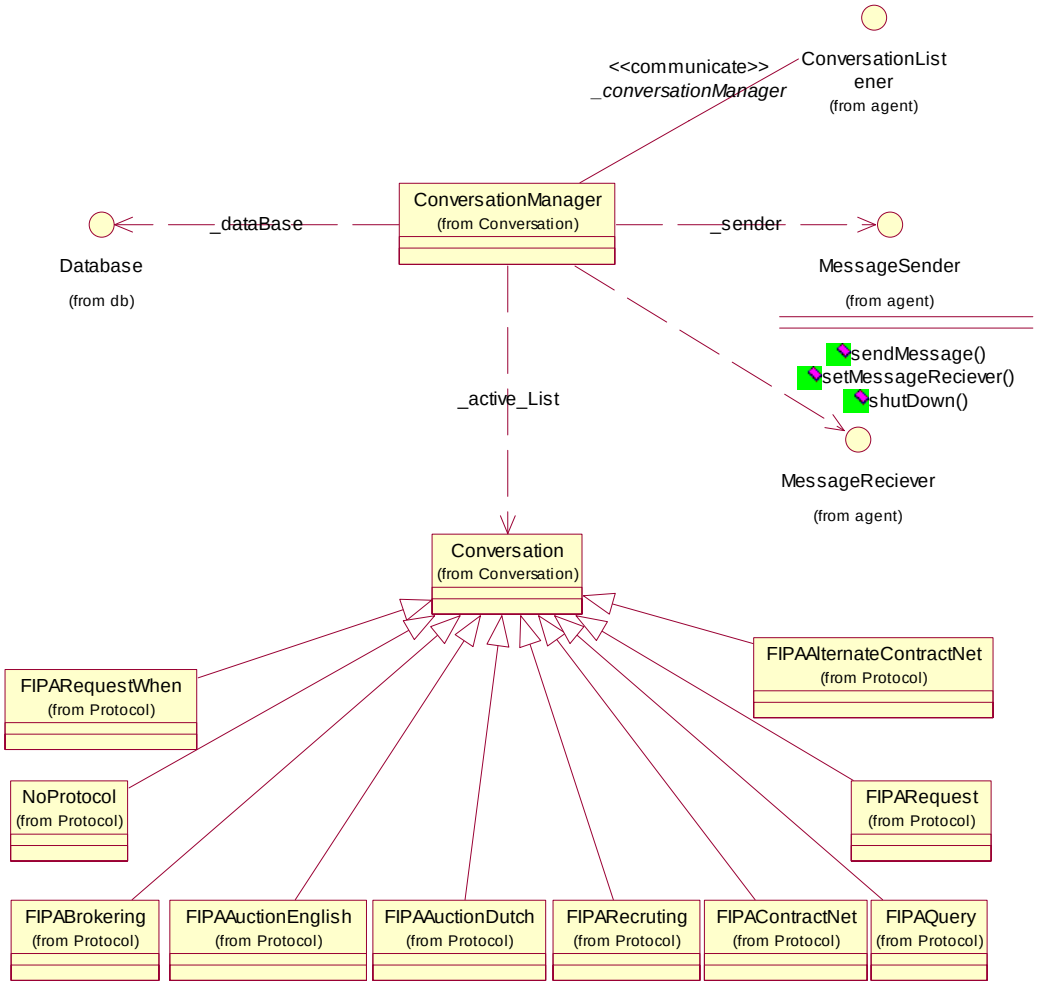


Figure 23: Conversation Manager Composition

Figure 23 highlights the key classes that compose the ConversationManager and their relationships. The ConversationManager implements the MessageReceiver interface so it can deal with incoming messages, the MessageSender interface to enable other components to send messages via it, has a reference to a MessageSender implementation to enable it to send messages, and has a reference to a ConversationListener so that it can pass updated conversations to components that implement this interface.

Conversation objects represent individual conversations, and encapsulate all of the state information and messages sent and received as part of that conversation. Hence they perform the necessary validation of the protocol being used by the conversation, and

provide mechanisms for discovering what messages have been sent/received, and the messages that should be sent next. The ConversationManager also has a reference to a Database implementation to enable Conversation objects to be stored once they are no longer active (i.e. when the conversation they represent has completed). A Map of active Conversation's is kept by the ConversationManager, enabling quick look-up upon receipt of a message. Various specialisations of the Conversation class are provided to enable different protocols to be supported. Each specialisation simply defines the protocol (in terms of performatives) to be followed for a particular conversation of that protocol type.

2.5.2. Protocol Definition

The protocol a particular conversation type follows is defined by specifying a class variable (__protocol) containing a tree-like structure defining the protocol. This is achieved through specifying an Object[] for each node in the tree, with details of what performative is expected next from which Agent in the conversation, what the desired action is (inform the Agent, ignore etc…) and references to its' child-nodes.
The standard form of the Object[] for a node is:
{ <String performative> [,<Integer action>], <Integer participant>, [<Object[] child_node>] }
which can be repeated to represent multiple possibilities at each node, where:

- performative is the performative of the next message to be received.
- action (optional) is the type of currently supported action which should occur when this message arrives. This is one of:
    - AGENT_ACTION_REQUIRED – recipient Agent should be informed of the arrival of this message
    - CONVERSATION_END – this is the end of the conversation (always reported to the recipient Agent). This value is implicit if a following child_node is not defined given the arrival of this message.
    - NO_AGENT_ACTION_REQUIRED – recipient Agent shouldn't be informed of the arrival of this message.

- participant indicates which Agent should send the message (0 is used for the initiator of a conversation, 1 for the recipient of the first message in the conversation).
- child_node is a reference to another Object[] which should become the current node when a message of this type is received.

The protocol definition can contain loops (although these will need to be closed using a static initialiser), and handling of "not-understood" messages is implicit.

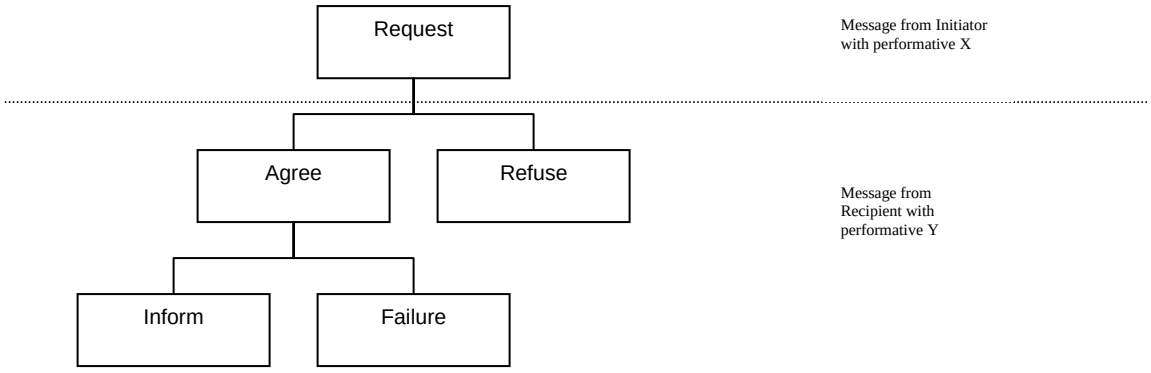


Figure 24: Example Message Protocol

Figure 24 is an example protocol, which could be "encoded" using the following Java code in a Conversation sub-class:

```
public static Object[] __ agree = { "inform", new Integer( 1 ),
                                    "failure", new Integer(1 )};
public static Object[] __request =
                      { "agree", new Integer( AGENT_ACTION_REQUIRED ), new Integer( 1
                      ), __agree, "refuse", new Integer(1 )};
public static Object[] __protocol =
{ "request", new Integer( AGENT_ACTION_REQUIRED ), new Integer( 0 ), __request };
```

Code Listing 1

2.5.3. Messaging

Figure 25 highlights the interactions involved when the Conversation Manager deals with an incoming message. It receives the message via the receiveMessage() method of the MessageReceiver interface it implements, and proceeds to add the message to an existing Conversation object (which encapsulates the state of a particular conversation), or creates a new one if this is the first message of a conversation. The fact that the Conversation has been updated is added to a queue within the Conversation Manager, so that Conversation updates can be dealt with in the order they occur. In the event that a message cannot be added to a Conversation (perhaps because doing so would violate the protocol the conversation is following), a not-understood is automatically generated in response, and the Conversation is brought to an end (the updated Conversation object will be added to the queue of pending Conversation's).

Sometime later, a Monitor pulls the updated Conversation from the queue, and passes it to the Conversation Managers' registered ConversationListener to be dealt with.

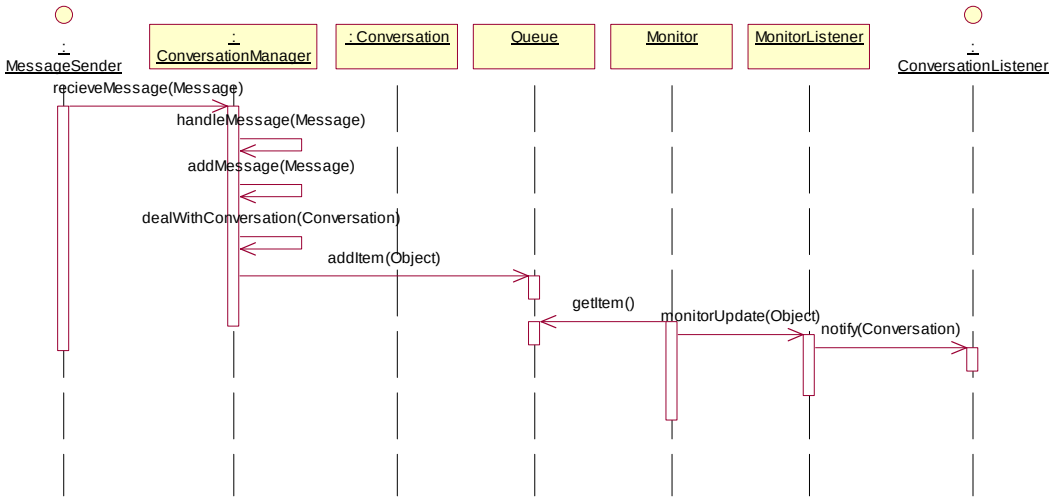


Figure 25: Interactions when Receiving a Message

Figure 26 depicts the interactions that occur when a message is being sent via the Conversation Manager. In this case, the registered ConversationListener invokes the

sendMessage() method (defined in the MessageSender interface, which the CM implements) on the Conversation Manager. The message is then added to the Conversation it belongs to, or a new one is created if the message is the start of a new conversation. Assuming this is successful, the message is sent via the MessageSender implementation registered with the Conversation Manager. In the event that a problem arises, at present a DIAGNOSTICS message is displayed on screen. In the future it is hoped more sophisticated error handling mechanisms will be introduced into the Conversation Manager, such that erroneous messages are passed back to the ConversationListener to be dealt with.

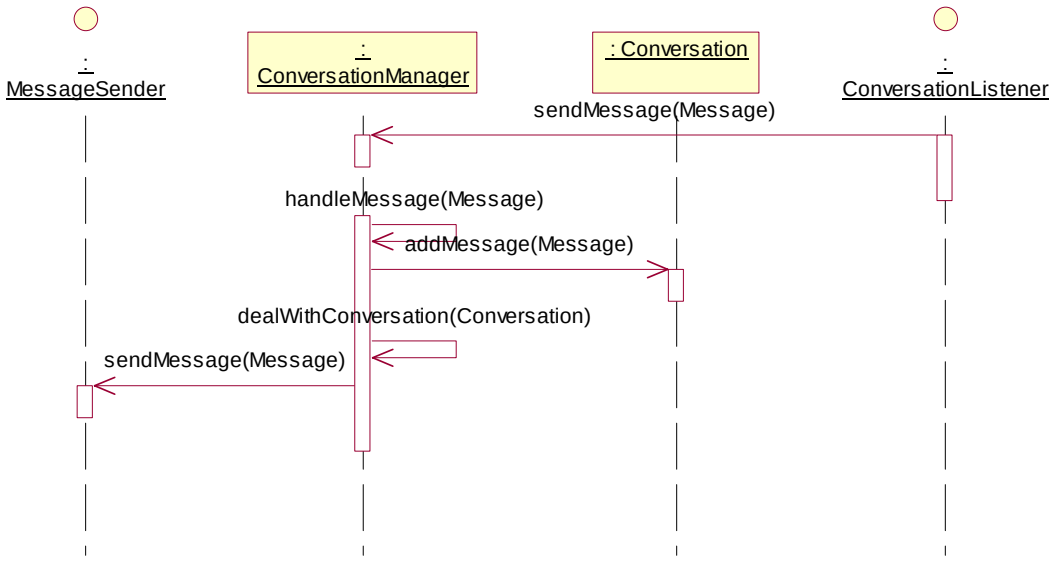


Figure 26: Interactions when Sending a Message

2.6 MTS (Message Transport Service)

The MTS provides the ability to send and receive messages to an Agent implementation.

2.6.1. Composition of the MTS

The MTS within FIPA-OS is logically split such that incoming and outgoing messages pass through a number of services within a "service stack" (see Figure 18). Each service is a stand-alone component that performs some transformation on outgoing messages,

and the inverse transformation on incoming messages. This model is used for the following reasons:

- Ideally each service performs its own function on incoming and outgoing messages – this enables the functionality of the MTS to be split into distinct decoupled components that can be individually tested (e.g. routing of messages to the ACC could be once service, whereas buffering messages could be another). Due to the non-trivial required behaviour of the MTS, it is logical to break the implementation of the requirements into individual components which in conjunction meet the overall requirements of the MTS.
- Addition of functionality to the MTS simply requires a new service to be created.
- Extra services can be slotted into the stack at runtime, due to lack of compile-time bindings between services.

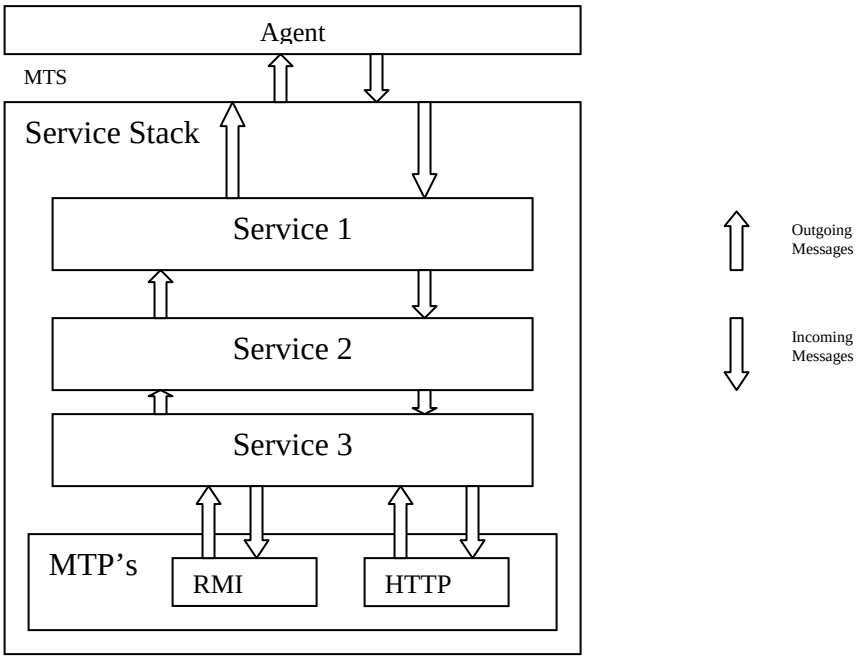


Figure 27: Logical Composition of the MTS

The MTS class implements the MessageSender interface, through which it provides access to the stack in use. Upon receipt of an outgoing message, it is immediately pushed into the stack. Whenever an incoming message is pushed from the top of the stack to the MTS class, it is passed to the MessageReceiver registered with the MTS – hence outgoing/incoming messages are pushed out of/in to the MTS instance in use by an Agent. Figure 19 highlights the class relationships for the MTS class.

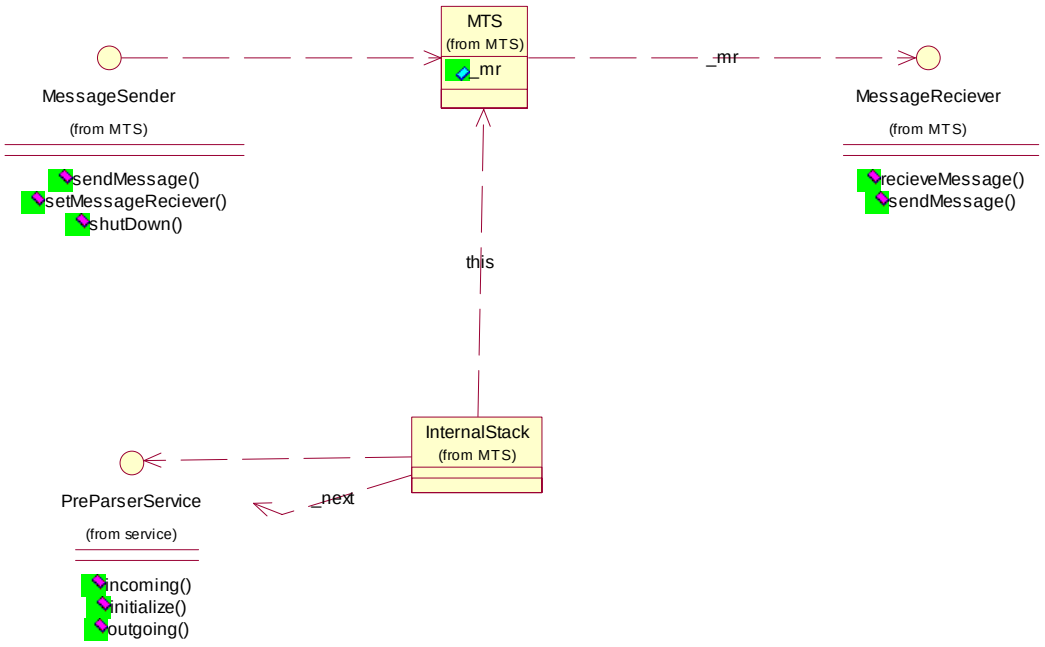


Figure 28: MTS Class Relationships

At present the services used in the MTS stack are hard-coded. In the future this will be dynamically determined based upon the profile of the Agent it belongs to. The MTS stack generally has two forms - one for internal transports, and another for external transports. The internal transports generally deal with a Message object, whereas external transports deal with an Envelope object and a byte[]. In either case, if a message cannot be sent, it will be propagated back up the stack for either another service to deal with, or the Agent implementation. This enables services higher up the stack to deal with error conditions before resorting to passing a message back to the Agent.

2.1.2 Services

Services within the stack implement at bare minimum the Service interface, although in order to bind services together they must implement either the PreParserService or PostParserService interfaces that extend the Service interface (see Figure 29). The ServiceStack class is provided to simplify the process of dynamically binding Service

implementations together using the initialise() method, since it will do this for all services it contains when its initialise() method is invoked.

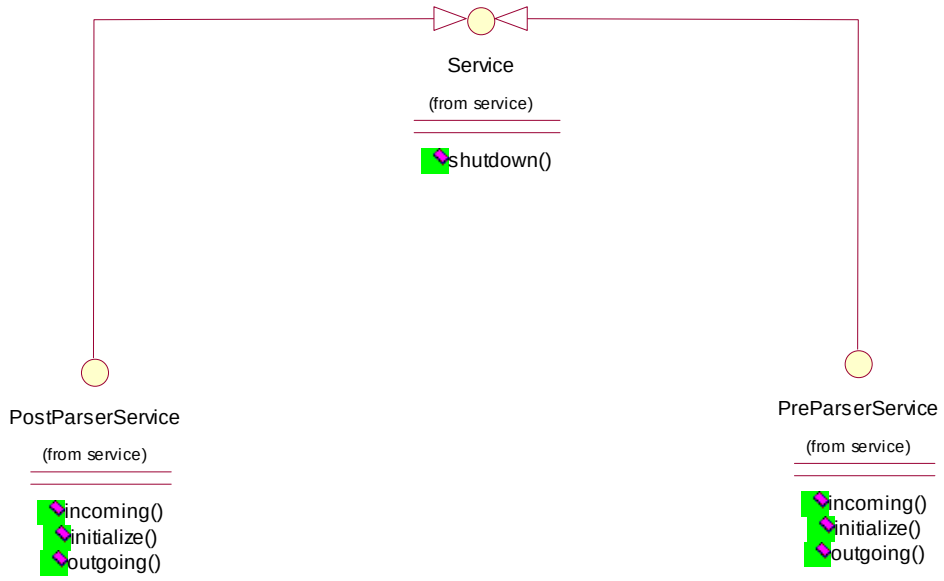


Figure 29: Service Interface Relationships

The Service interface also defines a number of failure reasons to be used with the Envelope getErrorCode()/setErrorCode() methods.

2.6.3. Pre-Parser Services

Services that implement this interface are expected to deal with Message objects, which encapsulate an Evenlope and an ACL object. In abstract terms, Pre-Parser Services deal with Objects.

2.6.4. Post-Parser Services

Services that implement this interface are expected to deal with messages in the form of an Envelope and byte[] tuple, where the byte[] represents the content of the envelope (i.e. the ACL message). In abstract terms, Post-Parser Services deal with “serialised”/”stringified” messages.

2.6.5. Parser Service

The ParserService is a concrete Service implementation – it implements both the PreParserService and PostParserService interfaces, providing a translation mechanism between the Object-Orientated Pre-Parser Services, and the flat byte[] representation of Post-Parser Services (i.e. it takes care of all necessary parsing and de-parsing with regard to incoming and outgoing messages).

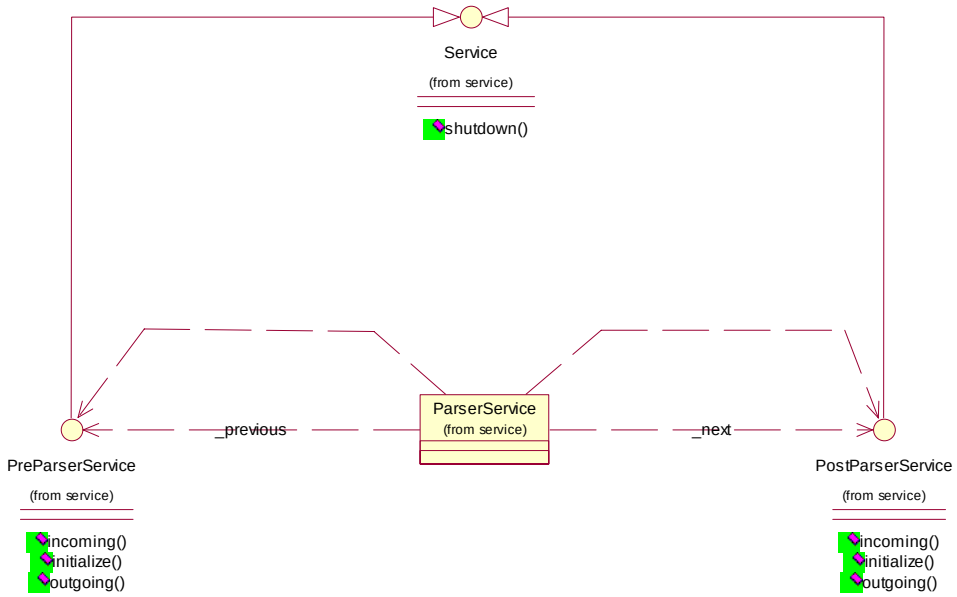


Figure 30: ParserService Class Relationships

2.6.6. Pre-Built Services

Bundled with FIPA-OS you'll find a number of "general-purpose" service implementations – some implement both the PreParserService and PostParserService interfaces since they can be placed anywhere in a stack (although they don't provide a translation mechanism such as the ParserService – services both below and above these services must implement the same interface).

- BufferService – This service implements both PreParserService and PostParserService interfaces (see Figure 22). Its purpose is to decouple services

within a stack by providing a FIFO queue in each direction within the stack between the services.

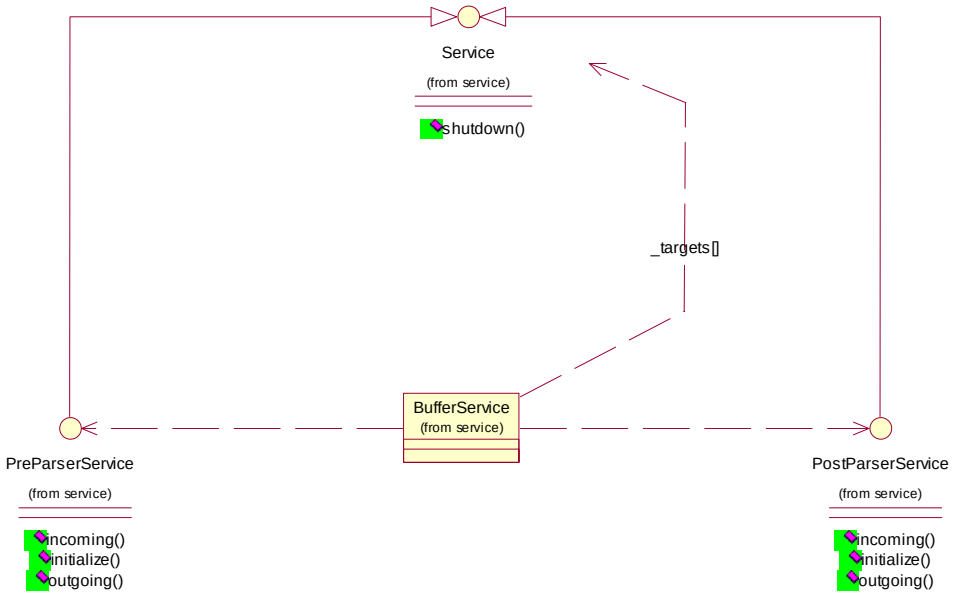


Figure 31: BufferService Class Relationships

- CommMultiplexService – Provides a mechanism for multiple MTP's to be joined to the bottom of a stack, and implements both PreParserService and PostParserService interfaces (see Figure 23), It provides support for the MTP's to be used to send messages as per the FIPA MTS Specification (i.e. attempting to use the MTP's based upon the order of the URL's within the intended-receivers AID addresses field). In the event that none of the available MTP's are able to send the message, it is propagated back up the stack with an appropriate error number.

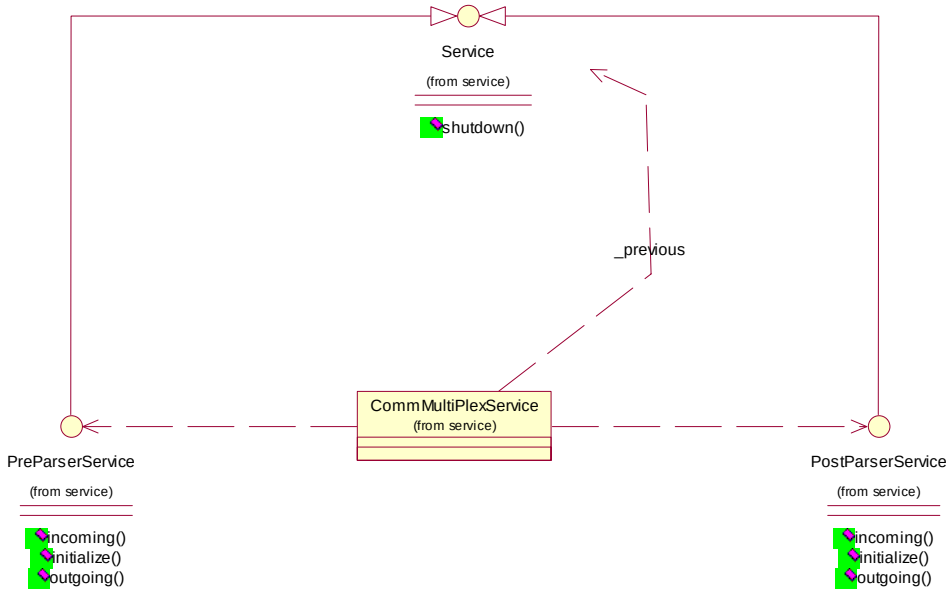


Figure 32: CommMultiplexorService Class Relationships

- ACCRouterService – This service implements the PreParserService interface only. Generally it passes outgoing messages straight through, and only takes notice when it receives an outgoing message that has been bounced back up the stack. In this event (depending on the reason why it has been bounced) it will be pass back down the stack, indicating that the message should be forwarded to the ACC in order to be sent. Hence this service routes messages (where appropriate) to the ACC.

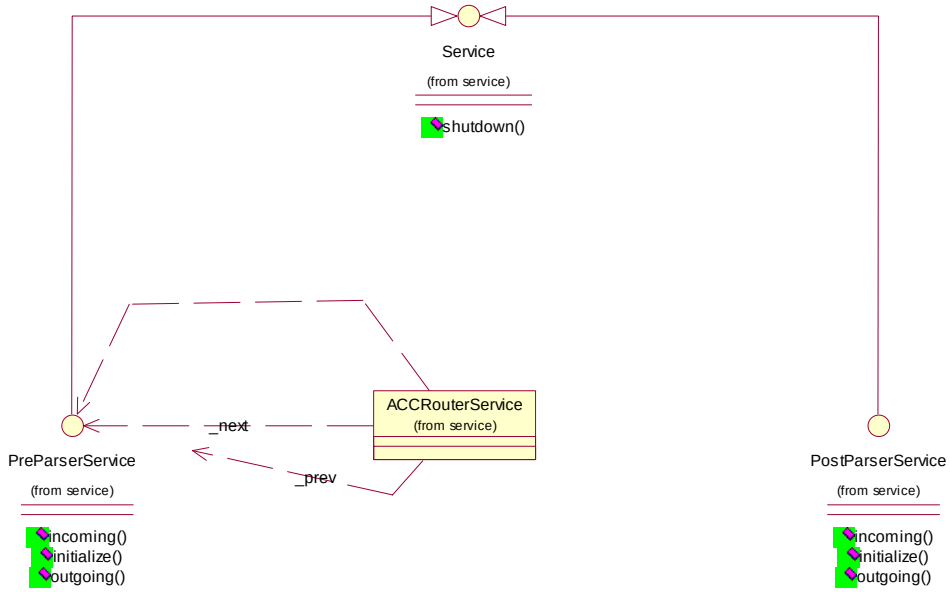


Figure 33: ACCRouterService Class Relationships

2.6.7. MTP's (Message Transport Protocols)

MTP's provide the mechanisms for sending and receiving messages from one Agent to another. Figure 25 highlights the relationships between MTP related classes.

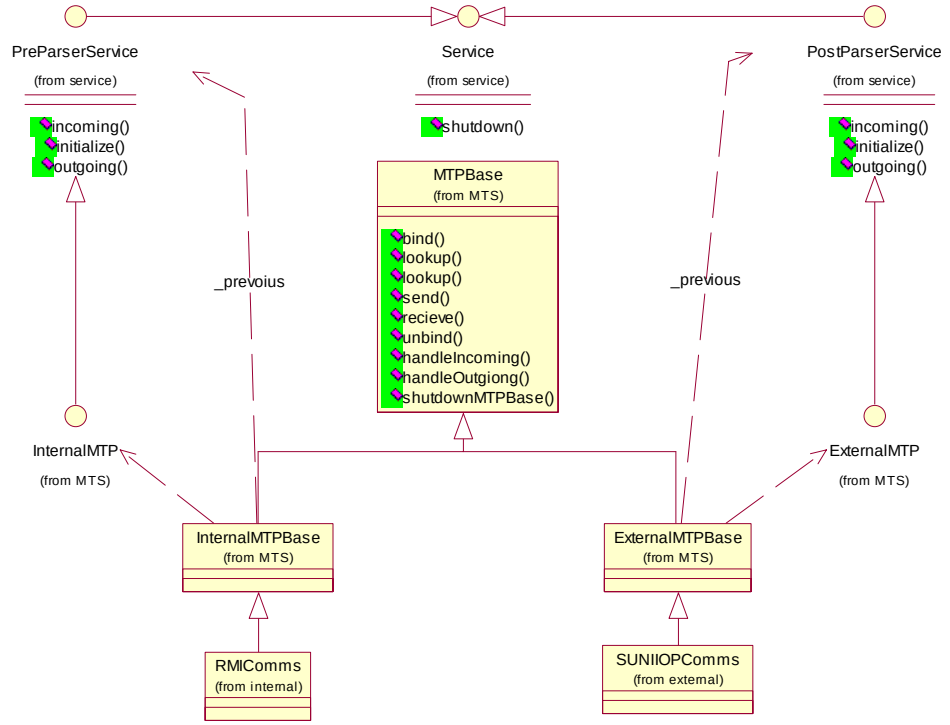


Figure 34: MTP Class Relationships

2.6.8. MTPBase Class

The MTPBase class contains functionality that is common across a number of MTP's. This includes handling incoming and outgoing messages, raising appropriate exceptions and error messages and other general behaviour. The MTPBase class deals with {Envelope, Object} tuples, where theEnvelope determines the behaviour of the MTP, and the Object is the payload of the message.

The InternalMTPBase and ExternalMTPBase classes specialise the MTPBase class to a particular type of MTP – either internal or external – and simply provides a translation mechanism between the InternalMTP and ExternalMTP interfaces and the functionality defined by the MTPBase class (i.e. providing the following translations respectively: Message _ {Envelope, Object} and {Envelope, byte[]} _ {Envelope, Object}). An MTP class which extends either of these classes is required to implement the following methods:

- public fipaos.util.URL getAddress()

A simple mechanism to retrieve the URL's for this MTP

- public java.util.List getProtocols()

A simple mechanisms to retrieve the URL protocol types that an MTP can deal with

- public void shutdown()
  Invoked when the MTP should be permanently shutdown
- protected void bind()
  Invoked when the MTP should startup/bind to a Naming Service
- protected void unbind()
  Invoked when the MTP should shutdown/unbind from a Naming Service (perhaps temporarily)
- protected Object lookup(URL name)
  Looks up a MTP specific reference (in the form of the returned Object) to the given URL
- protected Object lookup(String name)
  Looks up a MTP specific reference (in the form of the returned Object) to the given Agent name (of the form *agen*t@*ha*p).

and depending on whether InternalMTPBase or ExternalMTPBase is being extended, respectively either:

- protected void send(Object target, Message msg) Send the given message to the given target (the target Object will have been obtained from a previous call to lookup(), so it can be type-cast to whatever type the implemented lookup() method returns).

or:

- protected void send(Object target, Envelope env, byte msg[]) Send the given message to the given target (the target Object will have been obtained from a previous call to lookup(), so it can be type-cast to whatever type the implemented lookup() method returns).

An MTP implementation does not have to extend these classes, just implement the InternalMTP or ExternalMTP interfaces in order to be used with FIPA-OS. The

advantage to extending a sub-class of MTPBase class is however that the MTP class simply has to implement a small number of methods that simply deal with matters directly related to the MTP implementation.

2.6.9. Internal MTP's

An MTP generally falls into this category if:

- It provides a proprietary transport mechanism
- Aims to provide efficiency rather than inter-operability
- Does not require the message or its envelope to be prepared for its use (i.e. stringified or serialised in any form)
-

Internal MTP's are the main type of transport used by Agents within a platform, assuming that the majority of communications are intra-platform.

Figure 26 and Figure 27 highlight the interaction involved internally when messages are sent and received within an InternalMTPBase sub-class.

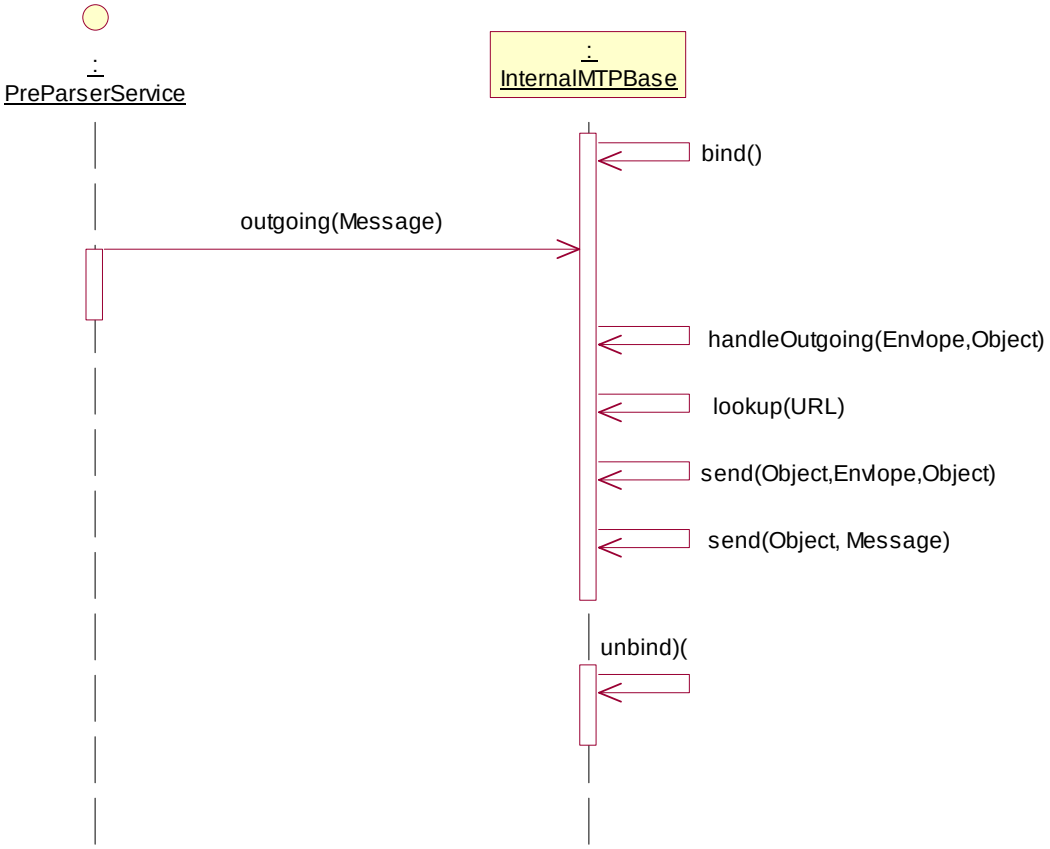


Figure 35: Interactions When Sending a Message

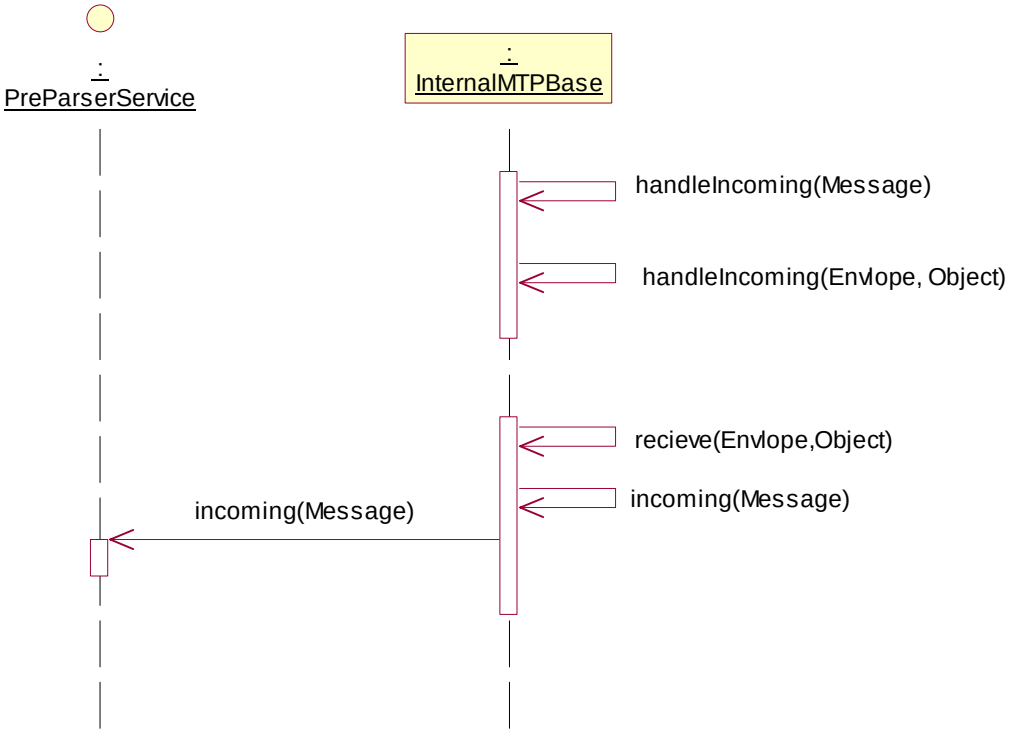


Figure 36: Interactions When Receiving a Message

2.6.10. External MTP's

An MTP generally falls into this category if:

- It provides a standardised transport mechanism (i.e. following a particular FIPA specification)
- Aims to provide inter-operability rather than efficiency
- Requires the message is prepared in some form before it is passed to is (i.e. stringified or serialised in some form).

External MTP's are currently only used by the ACC (although this will change when MTS profiles are introduced, allowing individual Agents to make use of external transports).

Figure 37 and Figure 38 highlight the interaction involved internally when messages are sent and received within an ExternalMTPBase sub-class.

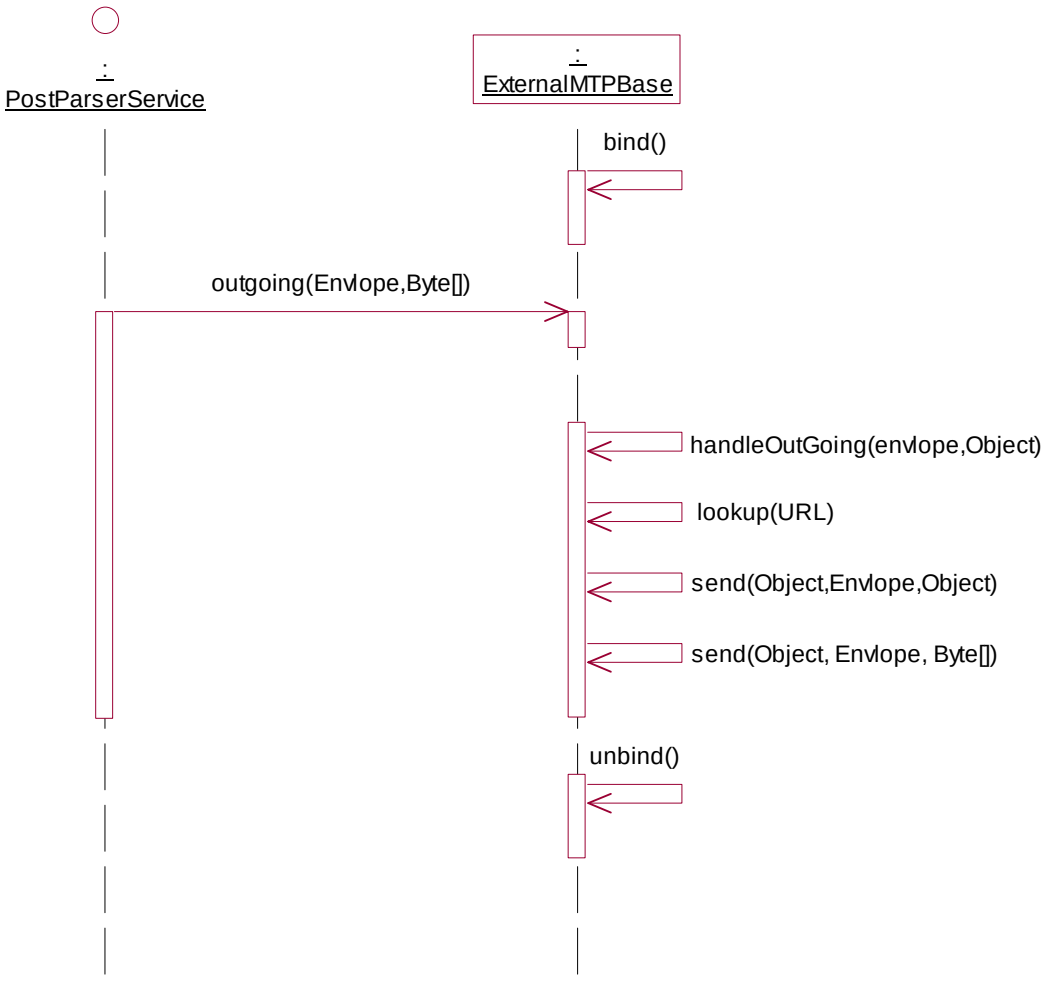


Figure 37: Interactions When Sending a Message

(incl. binding and unbinding of MTP which only normally occurs when an Agent starts/stops)

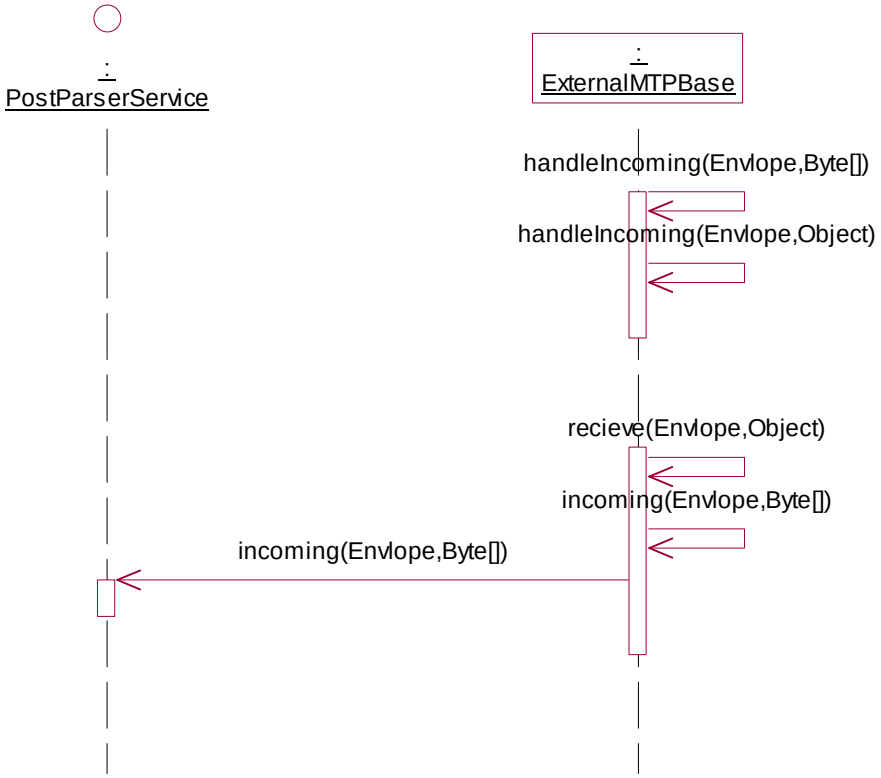


Figure 38: Interactions When Receiving a Message

Chapter 3
IBM Aglets Architecture

IBM AGLETS

The Java aglet extends the model of network-mobile code made famous by Java applets. Like an applet, the class files for an aglet can migrate across a network. But unlike applets, when an aglet migrates it also carries its state. An applet is code that can move across a network from a server to a client. An aglet is a running Java program (code and state) that can move from one host to another on a network. In addition, because an aglet carries its state wherever it goes, it can travel sequentially to many destinations on a network, including eventually returning back to its original host.

A Java aglet is similar to an applet in that it runs as a thread (or multiple threads) inside the context of a host Java application. To run applets, a Web browser fires off a Java application to host any applets it may encounter as the user browses from page to page. That application installs a security manager to enforce restrictions on the activities of any untrusted applets. To download an applet's class files, the application creates class loaders that know how to request class files from an HTTP server.

Likewise, an aglet requires a host Java application, an "aglet host," to be running on a computer before it can visit that computer. When aglets travel across a network, they migrate from one aglet host to another. Each aglet host installs a security manager to enforce restrictions on the activities of untrusted aglets. Hosts upload aglets through class loaders that know how to retrieve the class files and state of an aglet from a remote aglet host.

3.1. The aglet lifestyle

An aglet can experience many events in its life. It can be:

Created: a brand new aglet is born -- its state is initialized, its main thread starts executing

Cloned: a twin aglet is born -- the current state of the original is duplicated in the clone

Dispatched: an aglet travels to a new host -- the state goes with it

Retracted: an aglet, previously dispatched, is brought back from a remote host -- its state comes back with it

Deactivated: an aglet is put to sleep -- its state is stored on a disk somewhere

Activated: a deactivated aglet is brought back to life -- its state is restored from disk

Disposed of: an aglet dies -- its state is lost forever

Note that every activity besides creation and disposal involve either duplication, transmission across a network, or persistent storage of the aglet's state. Each of these activities uses the same process to get the state out of an aglet: serialization.

3.2. Serializing the state...

Aglet hosts use object serialization, available in JDK 1.1 or with the RMI (remote method invocation) add-on to JDK 1.0.2, to export the state of an aglet object to a stream of bytes. Through this process, the aglet object and the tree of serializable objects reachable from it, are written to a stream. An object is serializable if it implements either the Serializable or the Externalizable interface. In a reverse process, the state of the aglet can be reconstructed from the stream of bytes. Serialization allows an image of the heap (the heap's state) to be exported to a byte stream (such as a file) and then reconstructed from that byte stream.

3.3. ...but not all of the state

The state of the execution stacks and program counters of the threads owned by the aglet are *not* serialized. Object serialization touches only data on the heap, not the stacks or the program counters. Thus when an aglet is dispatched, cloned, or deactivated, any relevant state sitting on any stack of a running aglet, as well as the current program counter for any thread, is lost.

In theory, a software agent should be able to migrate with all its state: heap, execution stack, and registers. Some will likely consider the inability of aglets to do this as a flaw in

the aglet's implementation of mobile-agent theory. This feature of aglets arises out of the architecture of the JVM, which doesn't allow a program to directly access and manipulate execution stacks. This is part of the JVM's built-in security model. Unless there is a change to the JVM, aglets and any other mobile Java-based agent will be unable to carry the state of their execution stacks with them as they migrate.

Before it is serialized, an aglet must place on the heap everything it will need to know to be resurrected properly as a newly activated aglet, a freshly dispatched aglet, or a clone. It can't leave any of this information on the stack, because the stacks won't be reproduced in the aglet's new life. As a result, the aglet host informs an aglet that it is about to be serialized so that the aglet can prepare itself. When the aglet is informed of an impending serialization, it must place onto the heap any information it will need to continue its execution properly when it is resurrected.

From a practical standpoint, the inability of an aglet to migrate with its execution stacks is not an unreasonable limitation. It simply forces you to think a certain way when you write aglets. You can look at an aglet as a finite state machine with the heap as the sole repository of the machine's state. If at any point in an aglet's life you can know what state it is in by looking at its heap, then it can be serialized at any time. If not, then you must have a way to record sufficient information on the heap just prior to serialization such that you can continue properly when the aglet is resurrected.

Also, even though the inability to serialize execution stacks necessitates giving aglets a warning prior to serialization, such warnings probably are a good idea anyway. It is difficult to think of a case in which an aglet wouldn't want to know it was about to be serialized and why. It may need to finish some incomplete process before allowing the serialization, or it may want to refuse the action that requires the serialization. For example, if an agent is told it is about to be serialized and dispatched to an aglet host in Silicon Valley, it may refuse and decide instead to dispatch itself to a host on an island in the South Pacific.

3.4. How to write an aglet

The process of writing an aglet is in many ways similar to the process of writing an applet. To create an applet, you subclass class Applet. To initialize an applet, you override the init() method, the starting point for any applet. You can use init() to build the user interface of the applet. If you wish, you can fire off other threads from init(). If you do this, you also may override stop() and start() to stop and restart your threads when the browser leaves and returns to the Web page. If you don't create any threads in init(), your applet likely will get at least one thread just because class Applet descends from class Panel. The AWT user-interface library of which Panel is a part will provide whatever threads are needed to run the user interface you create in init().

The aglet development and run-time environments provide a library of Java classes that support the creation and running of aglets. To create an aglet, you must subclass class Aglet, which includes several methods you can override to customize the behavior of your aglet. The aglet's counterpart to the init() method of applets is the onCreation() method. To initialize an aglet, you override onCreation(). The onCreation() method is invoked only once in an aglet's lifetime and should be used only for initialization.

The aglet also has a run() method, which represents the entry point for the aglet's main thread. This is similar to the main() method of a Java application, except that run() is invoked each time an aglet arrives at a new aglet host. For example, if you designed a CatAglet that visits nine different aglet hosts looking for MouseAglets, onCreation() would be invoked only once, when the CatAglet was first instantiated at its first host. Once onCreation() completed, run() would be invoked. Each time the CatAglet arrived at a new host, a method called onArrival() would be invoked to perform any initialization. Once onArrival() completed, run() would be invoked to get the aglet started again at the new host.

Starting run() again each time an aglet is brought to life illustrates the inability of aglets to transmit the state of their execution stacks. For example, imagine a HealthyAglet whose run() method periodically invokes a method named walk(). If, as it is walking, the HealthyAglet is serialized and transmitted to another host, it wouldn't by default continue

executing where it left off in walk(). It would start over again at the beginning of run(). Thus, when the aglet is informed that it is about to be serialized, it would need to record on the heap that it is walking -- perhaps in an instance variable of HealthyAglet. That instance variable would be serialized and would migrate with the aglet. When run() is invoked to start the aglet's new life, the run() method would check the instance variable, see it was walking beforehand, and call walk().

3.5. The callback model

Before any major event in an aglet's life, a "callback" method is invoked to allow the aglet to prepare for (or refuse to partake in) the event. This is how an aglet learns that it is about to be serialized. For example, before an aglet is dispatched to a new location, the aglet's onDispatch() is invoked. This method indicates to an aglet that it is about to be sent to a new host, the URL of which is specified as a parameter to onDispatch(). In the body of onDispatch(), the aglet must decide whether or not to go. If the aglet decides it doesn't want to go, it throws an exception. If it decides to go, it must complete any unfinished business and prepare its state for serialization. When it returns from onDispatch(), its state will be serialized and all its threads terminated. The class files and serialized state will then be sent to the new host, where the aglet will be resurrected.

The method onDispatch() is a "callback" method because the aglet host invokes it some time after another method, dispatch(), is invoked. An aglet can invoke dispatch() on itself or on another aglet. This callback model for aglets is similar to that of windowing user interfaces. To repaint an AWT component, for example, you invoke the component's repaint() method. At some point later, the system calls back the component's update() method, which in turn calls paint().

The Aglet class defines these five callback methods, which you can override to customize the behavior of your aglet:

onCloning() -- called before a clone operation
onDispatch() -- called before a dispatch
onReverting() -- called before a retraction

onDeactivating() -- called before a deactivation
onDisposing() -- called before a dispose operation (Unlike real life, an aglet can throw an exception if it doesn't want to die.)

For each of these processes, the Aglet class has a corresponding method that triggers the action: clone(), dispatch(), retract(), deactivate(), and dispose(). Some time after these are called, the aglet host will invoke the appropriate callback method.

Each time an aglet begins execution at a host, the host invokes an initialization method on the aglet. When the initialization method returns, the host invokes run(). Depending on the event that precipitated the aglet's new life, the aglet host will choose to invoke one of these four initialization methods:

onCreation() -- called the first time an aglet springs to life
onClone() -- called on a clone after a clone operation
onArrival() -- called after a dispatch or a retraction
onActivation() -- called after an activation

3.6. Interaction between aglet and host

An aglet interacts with its environment (its aglet host) through an AgletContext object. An aglet can obtain a handle to its context by invoking getAgletContext(), a method it inherits from base class Aglet. The aglet context has methods such as createAglet() and retractAglet(), which allow an aglet to add new aglets (or get an old aglet back) to its local host.

3.7. Interaction between aglets

To interact with each other, aglets do not normally invoke each other's methods directly. Instead they go through AgletProxy objects, which serve as aglet representatives. For example, if a BossAglet wishes to make a request of an EmployeeAglet, the BossAglet obtains a handle to a proxy object that "represents" the EmployeeAglet. The BossAglet then makes a request by invoking a method in the EmployeeAglet's proxy, which in turn forwards the request to the actual EmployeeAglet.

The AgletProxy class contains methods that allow aglets to request other aglets to take actions, such as dispatch(), clone(), deactivate(), and dispose(). The aglet that has been requested to take an action can comply, refuse to comply, or decide to comply later.

The proxy also allows an aglet to send a message, either synchronously or asynchronously, to another aglet. A Message object is supplied for this purpose; it carries a String to indicate the kind of message plus one other optional piece of data, either a String or one of Java's primitive types. To send a message you create a Message object and pass it as a parameter to the sendMessage() or sendAsynchMessage() method of the proxy object.

An aglet must go through a proxy object to interact with an aglet, even if both aglets are in the same aglet host. The reason aglets aren't allowed to directly interact with one another is that the aglet's callback and initialization methods are public. These methods should be invoked only by the aglet host, but if an aglet could get a handle to another aglet, it could invoke that aglet's callback or initialization methods. An aglet could become very confused if another aglet inadvertently or maliciously invoked these methods directly.

The aglet being represented by a proxy might be local or remote, but the proxy object is always local. For example, if a BossAglet in Silicon Valley wants to communicate with an EmployeeAglet on a South Pacific island, the BossAglet gets a local AgletProxy object, which represents the remote EmployeeAglet. The BossAglet merely invokes methods in the local proxy, which in turn communicates across the network to the EmployeeAglet. Only aglets, not proxies, migrate across the network. A proxy communicates with a remote aglet that it represents by sending data across the network.

You get a proxy to an aglet in one of three ways, each of which involves invoking a method in the context object:

1. By creating the aglet in the first place with createAglet(). (This returns a proxy object.)

2. By searching through an enumeration of local proxies returned by getAgletProxies().
3. By supplying an "aglet identifier" and, if remote, an aglet location as parameters to getAgletProxy(). (Every aglet, upon creation or cloning, is assigned a globally unique aglet identifier.)

3.8. Security

Mobile-agent systems, such as aglets, require high levels of security, because they represent yet another way to transmit a malicious program. Before aglets can be used in practice, there must be an infrastructure of aglet hosts that prevent untrusted aglets from doing damage but provide trusted aglets with useful access to the host's resources. Security is amply provided for in Java's intrinsic architecture and in the extra security features of JDK 1.1, but as with applets, some attacks (such as denial of service by allocating memory until the host crashes) are still possible. Currently, the aglet hosts from IBM (named Tahiti and Fiji) place very severe security restrictions on the activities of any aglet that didn't originate locally.

Aglets are IBM's version of mobile agents. They are *"…Java objects that can move from one host on the Internet to another. That is, an aglet that executes on one host can suddenly halt execution, dispatch to a remote host, and resume execution there. When the aglet moves, it brings along its program code as well as its state (data). A build-in security mechanism makes it safe to host untrusted aglets."* Aglets' roots are in applets and servlets 6 : when aglet moves, it brings with it everything it needs to execute – including the program code and possible runtime data. IBM has intended the aglets to be the industry standard in mobile agents, they should be secure, easy to program and be platform independent (meaning Java Aglet Application Programming Interface platform independent).

Chapter 4

The e-mart Application

e-mart

4.1. Project Specification

This project describes the implementation of a sample marketplace application called e-mart. The objective is to enable the market participants to electronically trade tasty fruit over a network via their agents.

For simplicity, we shall assume only certain types of fruit will be traded: apples, oranges, pears, bananas and melons. Furthermore, we shall assume that each type of fruit is a commodity and traded in uniform size boxes, i.e. there is no inherent difference between two different boxes of apples.

To keep the project small-scale, we shall assume there are 3 participants that are each capable of buying and selling, these are:

- The first participant will be called **OrchardBot**, the representative of an orchard that is attempting to sell some of its produce. We shall assume the produce has just been picked, and so it possesses a stock of 100 boxes of apples, 80 boxes of oranges and 60 boxes pears, but no cash. It will attempt to sell its goods with a minimum profit margin of 10%.
- The second participant is **SupplyBot**, a representative of several fruit producers that begins with a small stock of 30 boxes of apples, 10 boxes of pears, 30 boxes of oranges, 20 boxes of melons and 20 boxes of bananas. As the supplier needs to remain well stocked it has a budget of Rs. 100 to fund purchases. This agent's minimum profit margin is 5%.
- The third participant is **ShopBot**, a representative of a supermarket chain. Its existing stock is down to 10 boxes of bananas, 5 boxes of pears and 10 boxes of melons; but it has a budget of Rs. 500 to purchase additional stocks. This agent's minimum profit margin is also 5%.

We shall assume all transactions are conducted in Rupees, although as only one currency unit is involved the particular type used has little significance in this application. Furthermore to provide a baseline for prices we shall assume that:

- boxes of apples, oranges and pears each cost Rs 5
- boxes of bananas cost Rs 8
- boxes of melons cost Rs 10

We shall also assume that the market is open, (i.e. agents can join or leave at any time), and agents are not initially aware of their counterparts.

4.2. Application Analysis

Once we have a specification, the desired system can be situated within a domain that contains similar features and challenges. As the agents will exhibit trading behaviour it is obvious that this application is best situated within the *Multi-Agent Trading* domain.

4.2.1. Role Modelling

4.2.1.1. The Components of a Role Model

The role models are described from several perspectives. Each role model entry begins with a description summarizing its main features and general applicability. The model's constituent roles are then shown in a Role Model diagram, like that shown in Figure 39

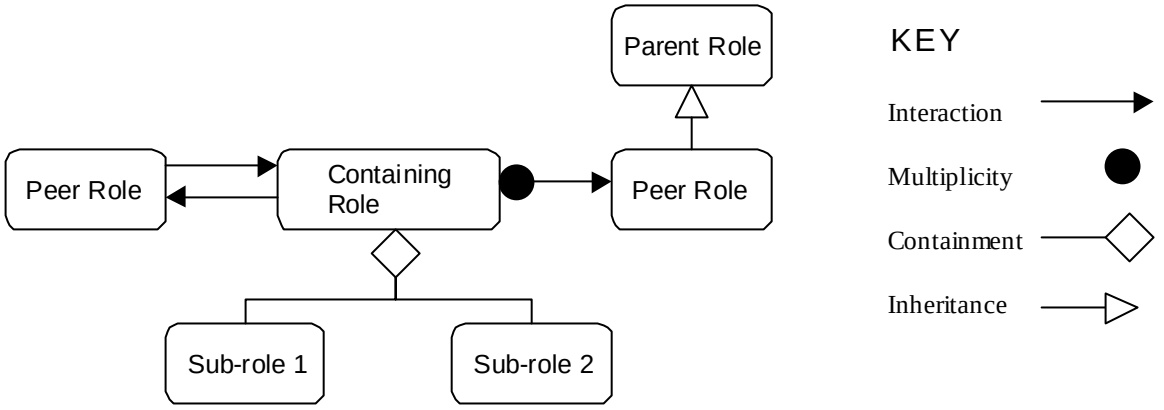


Figure 39: The Notation used within a typical Role Diagram

Role model notation originates from the UML (Unified Modelling Language) class diagram notation, . ***In role diagrams the key concepts are roles rather than classes***, (represented by rectangles), whilst containment and inheritance are depicted in the same fashion as in class diagrams, with diamond and triangle headed lines respectively.

The only difference in notation arises from the key difference between class diagrams and role diagrams. Whereas class diagrams describe the static relationships between classes, role models describe the dynamic interactions between roles. Hence the UML class diagram has been augmented with additional notation to depict interactions: the arrowhead line. Where an arrowhead line is shown with a filled circle, this means more than one simultaneous interaction can occur between entities playing these roles.

After the role diagram, the next component of each role model entry is the *collaboration diagram*. This abstracts away from the specialization and containment relationships between roles and instead concentrates on how they interact; an example is shown in Figure 40.

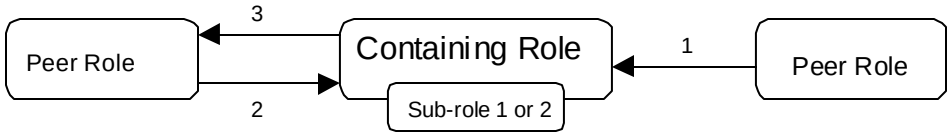


Figure 40: The Notation used within a typical Role Diagram

The main difference between collaboration and role diagrams is that only interactions are shown. Each interaction is annotated with a number, which refers to an explanation of the interaction found later in the role model. Where the collaboration diagram is made easier to understand by the inclusion of sub-roles, these are shown inside their containing role.

The next part of a role model entry is the role description section; this consists of tables like that shown in table 2. The purpose of this section is to describe each role in terms of its social obligations and application-specific functionality. Hence each of the interactions shown in the collaboration diagram will appear in the corresponding role description.

As well as interactions with other roles, each entry also describes the interactions between the role and its external interfaces. The 'external interfaces' represent the means through which the role performs its application-specific activities, such as accessing databases, or reading information from a user interface. Some entries may also describe the social abilities necessary for the role to fulfill its social responsibilities.

ROLE NAME	
Role Model: the model that contains this role	
Relationships to other roles: roles that this contains or specializes	
Description: A written description of this role's characteristics, abilities, responsibilities etc.	
Responsibilities:	**Collaborators:**
[1] An outgoing interaction	⇒ Source Role
[2, 3] An incoming and outgoing interaction	⇐ ⇒ Source and Destination Role
External Interfaces:	

An application specific activity
Necessary Social Protocols: The abilities necessary to fulfill its responsibilities

Table 2: The component fields of a Role Description

Role models are patterns, representing the simplest possible solution for a particular application. As the role model is effectively the lowest common denominator solution the final part of each role model describes common variations that extend its functionality.

We will now model the archetypal decentralized marketplace. Such a market is characterized by an absence of institutional rules and centralized arbiters. Instead trading occurs directly between buyers and sellers using a protocol agreed between the parties concerned. It is assumed that the roles of buyer and seller are not mutually exclusive, i.e. agents may be buyers and sellers simultaneously.

The decentralized nature of this type of marketplace means potential buyers require a means of finding vendors, and prospective vendors require a means of advertising. In static marketplaces, (i.e. where membership does not change), it may be sufficient to equip each agent a priori with knowledge of its peers. A more flexible alternative, and one that is used by dynamic marketplaces, is to create at least one agent to play the role of a broker. The broker maintains an up-to-date registry of prospective buyers and sellers, providing an efficient means for agents to locate appropriate transaction partners.

It is also worth noting that this role model does not assume the presence of a trusted third party that mediates transactions and transfers ownership of the traded goods.

4.2.2. Similarities and Differences

Implicit in this role model is the presence of a name server, as described in the FIPA Architecture; although this is not shown in Figure 41 for clarity. The Distributed Marketplace role model expands the role of the Application's Facilitator role, replacing it with a role termed 'Broker'.

Distributed Marketplace Role Model Diagram

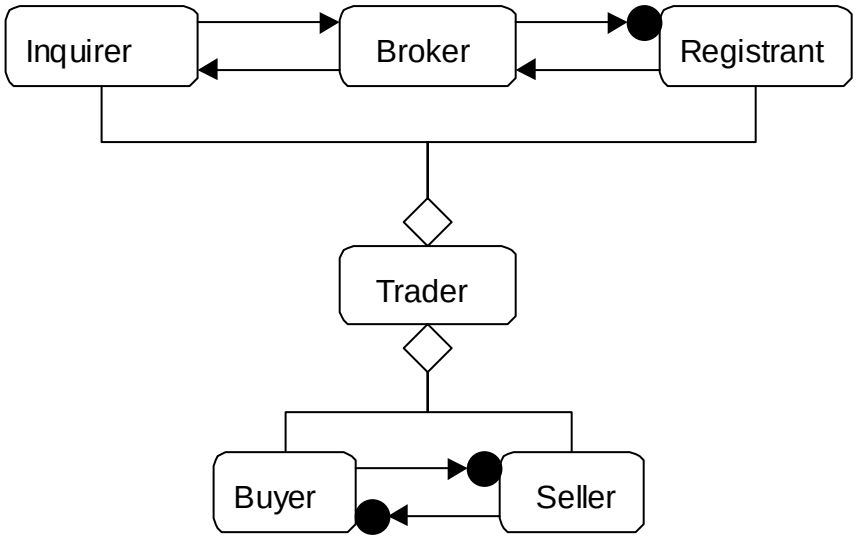


Figure 41: The roles present in a typical distributed marketplace application.

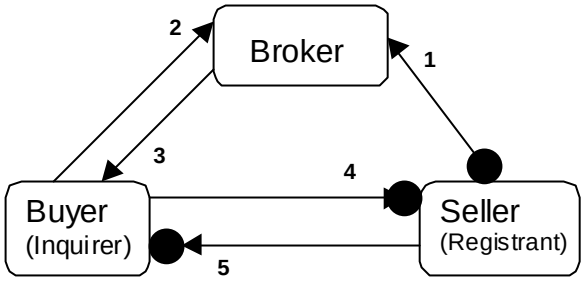


Figure 42: The interactions between the roles of a typical distributed marketplace. Distributed Marketplace Collaboration Diagram

4.2.3. Interaction Summary:

	Collaboration	Explanation
1	Registration	Sellers register or de-register their items for sale
2	Find Request	Asks for agents selling items of interest
3	Find Response	A list of agents matching the desired criteria
4	Buyer Offer	Message containing buyer's proposal
5	Seller Response	Seller's reply to a previously submitted offer

Table 3: Interaction Summary

(The initial interactions between agents as they register with a Name Server are not shown)

4.2.4. Distributed Marketplace Role Descriptions

BUYER	
Role Model: Distributed Marketplace	
Relationships to other roles: Contained by the Trader role	
Description: This is the role played by potential purchasers. No assumptions are made as to the expertise of the buyer, which may be a simple proxy or an intelligent trader with its own buying strategies. Note how the Buyer is not required to register itself with a Broker. The knowledge held by a Buyer will vary. They must be aware of the application ontology (the set of tradable concepts and their attributes), and will probably possess expertise on pricing and trading strategies.	
Responsibilities:	**Collaborators:**

[2, 3] To request information on known sellers	⇐ ⇒ Brokers
[4, 5] To communicate bids to potential vendors	⇐ ⇒ Sellers
External Interfaces:	
To facilitate the entry of user transactions	
To interpret vendor responses	
To facilitate payment and ownership transfer	
Prerequisites:	
At least one trading or auction protocol	

Table 4: Distributed Marketplace Role Descriptions- Buyer

SELLER	
Role Model: Distributed Marketplace	
Relationships to other Roles: Contained by the Trader role	
Description: This is the role played by potential vendors. No assumptions are made as to the expertise of the seller, which may be a simple proxy or an intelligent trader with its own selling strategies. A key issue for Sellers is advertising; this necessitates registering with a Broker if they are to be found by potential buyers. The knowledge held by a Seller will vary. They must be aware of the application ontology (the set of tradable concepts and their attributes), and will probably possess expertise on pricing and trading strategies.	
Responsibilities:	**Collaborators:**
[1] To advertise new items for sale	⇒ Brokers
[4, 5] To receive and respond to bids	⇐ ⇒ Buyers

External Interfaces:
To facilitate user selling preferences
To interpret bids
To facilitate payment and ownership transfer
Prerequisites:
At least one trading or auction protocol
TRADER
Role Model: Distributed Marketplace
Relationships to other Roles: specializes Task Agent, contains Buyer and Seller
Implementation: Modified Task Agent

Table 5: Distributed Marketplace Role Descriptions- Seller/Trader

BROKER	
Role Model: Distributed Marketplace	
Relationships to other Roles: A variant of Facilitator	
Description: A variation on the standard Application Facilitator role which, instead of maintaining a registry of abilities, stores notices about goods sought and for sale. Brokers may be reactive (where active traders will advertise themselves) or proactive (where the broker will request information on each trader's status). Like the Facilitator the knowledge collected by the Broker is distributed on demand in response to queries.	
Responsibilities:	**Collaborators:**
[2] To receive notifications from participants	⇒ Traders
[3] To respond to queries on market	⇐ Traders

participants	
External Interfaces:	
To store information on market participants	
Prerequisites: None	
Implementation: Modified version of default Facilitator	

Table 6: Distributed Marketplace Role Descriptions- Broker

4.2.5. Distributed Marketplace: Common Variations

The role model of Figure table 3 represents the 'lowest common denominator' of distributed marketplace applications. In practice this role model is likely to be extended through the addition of the following related roles:

4.2.5.1. Marketplace Visualiser

Often a marketplace has a means of visualising transactions. This is typically achieved by employing the Observer pattern, where the Visualiser is the observer and the other agents play the role of Subjects.

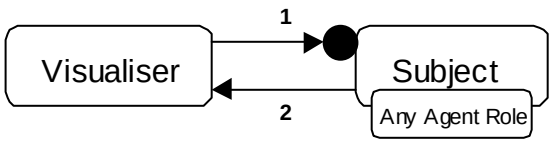


Figure 43: Visualizer

Interaction Summary:

	Collaboration	Explanation
1	Inform Request	Subjects are asked to forward activity reports
2	Activity Notification	Activity reports are dispatched

Table 7: Interaction Summary-Visualiser

4.2.5.2. Transaction Mediator

In cases where the trading parties can not be relied upon to honour agreements a trusted third party is often used to facilitate payment and the transfer of goods. The mediator role is commonly found in conventional non-agent commerce, good examples being clearing organisations like Amex and Visa.

Interaction Summary:

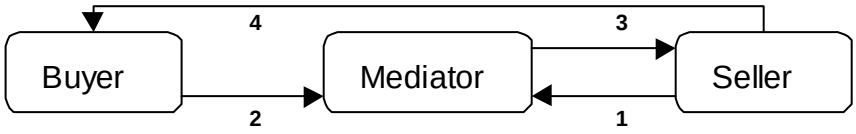


Figure 44: Interaction Summary - Mediator

	Collaboration	Explanation
1	Sale Notification	Terms of sale (e.g. price, item) sent
2	Payment Authorisation	Payment made to Mediator
3	Payment Notification	Payment verified, commission taken (if relevant), remainder transferred to Seller
4	Ownership Notification	Message confirming sale and transfer of goods

Table 8: Interaction Summary - Mediator

4.2.6. Select Role Models

From the initial specification we notice that the agents with trade between themselves rather than through an intermediary. This would tend to suggest that the *Trader* role of the *Distributed Marketplace* role model would better model the participants .From the

Distributed Marketplace role model we can identify the roles with which the *Trader* agents will collaborate. The role model shows each *Trader* can play *Inquirer* and *Registrant* roles relative to a *Broker*. This suggests that a *Broker* agent should be incorporated into our solution, which is not inconsistent with our problem specification.

Next we need to consider the entities of the standard *Application* role model, upon which this role model is based. We will need an agent in the *Agent Name Server* role, but as the *Broker* duplicates the functionality of the *Facilitator* only one needs to be created.

It is possible for each of the identified roles to be played by an individual agent, and which interact together to accomplish a common cause. Alternatively, a single agent could play several roles. In this application the latter arrangement seems more appropriate, i.e. to create one agent for each participant that will encapsulate all 4 roles of the *Trader*.

Having considered all the roles within the basic *Distributed Marketplace* role model we can move on to consider its common variations. One of these, the *Visualiser* variation would seem to satisfy the aspect of the specification that calls for some means of activity analysis.

So, to summarise, after deciding on the domain and considering its constituent role models we have decided to base our solution on the *Distributed Marketplace* role model. Then we have decided to create several agents to fulfil the roles found within the role model:

Agent Name	**Roles Played**
OrchardBot	Trader (Buyer, Seller, Inquirer, Registrant)
SupplyBot	Trader (Buyer, Seller, Inquirer, Registrant)
ShopBot	Trader (Buyer, Seller, Inquirer, Registrant)
Broker	Broker (Facilitator)
Visual	Visualiser
ANS	Agent Name Server

Table 9: Agents (Roles) in the Distributed Market Place

Having identified what roles should exist within the application, we can begin thinking about how agents will realize each role.

4.2.7. List Agent Responsibilities

Each role played by an agent entails some responsibilities, e.g. resources that will need to be produced or consumed, interactions with external systems etc. Hence the next stage is to use the role descriptions to create a list of responsibilities for each agent.

From the descriptions in the *Distributed Marketplace* role model we can obtain the list of responsibilities for the 4 constituent roles of a *Trader* agent. As some responsibilities are shared between roles they only need to be considered once. The responsibilities involved can be categorized as social or domain responsibilities, the former involving interaction with other agents, and the latter involving some local application-specific activity; this results in the following:

TRADER - Social Responsibilities	
Origin	**Responsibility**
Seller-Registrant	To register and de-register presence in marketplace
Buyer-Inquirer	To request information on known sellers
Buyer	To send bids to potential vendors
Seller	To receive and respond to bids

Table 10: Trader - Social Responsibilities

TRADER – Domain Responsibilities	
Origin	**Responsibility**
Buyer	To facilitate the entry of user transactions
Buyer	To interpret vendor responses

Buyer, Seller	To exchange payment and ownership
Seller	To facilitate user selling preferences
Seller	To interpret bids

Table 11: Trader – Domain Responsibilities

The next role to consider is the *Broker*:

BROKER - Social Responsibilities	
Origin	**Responsibility**
Broker	To receive notifications from participants
Broker	To respond to queries on market participants

Table 12: Broker - Social Responsibilities

BROKER – Domain Responsibilities	
Broker	To store information on market participants

Table 13: Broker - Domain Responsibilities

Finally, as the functionality of the *Visualiser* role is already present in the standard Application role model we shall adopt the pre-built Visualiser tool; hence no design is necessary for this role.

4.3 Application Design

The design process is a process of refinement, mapping each of the responsibilities identified in the previous stage to a generalized problem, and then choosing the most appropriate solution.

4.3.1 Problem Design

Having obtained the responsibilities required for each agent, the next stage is to map each responsibility to the problem it attempts to solve. As these problems tend to be variations on common challenges, it is possible to reapply past tried and tested solutions to solve the problems in question. We begin by considering the Broker:

BROKER

SOCIAL RESPONSIBILITIES

Responsibility:	To receive notifications from participants
Origin:	Broker
Problem:	Receive and Store [Sellers, Commodities-For-Sale]
Solution:	Configure the Facilitator Agent
Explanation:	By default the Facilitator will query agents pro-actively, but as this scenario calls for a reactive Broker this can be achieved by setting its cycle time to 0.
Responsibility:	To respond to queries on market participants
Origin:	Broker
Problem:	Match Requirements, Send Message [Buyer, Seller Identity]
Solution:	Automatic – functionality provided by Facilitator Agent

Table 14: Broker - Social Responsibilities

DOMAIN RESPONSIBILITIES

Responsibility:	To store information on market participants
Origin:	Broker
Problem:	Store [Seller – Commodity, Price]
Solution:	Automatic – functionality provided by Facilitator Agent

Table 15: Broker - Domain Responsibilities

How then have these solutions been derived? In the case of the *Broker* it is through the reuse and adaptation of existing solutions from its parent role, the *Facilitator*. This is justifiable because the problems facing the *Broker* are very similar to those tackled by the

Facilitator. This ethos of reusing existing solutions is central to most development toolkits.

The next role to consider is the Trader, which is descended from the Task Agent role. The Task Agent role is quite generic, and consequently there will be much less scope for reuse in the application specific *Trader* role. Instead, solving the *Trader* problems will require deeper knowledge of agent behaviours, protocols and strategies; this is summarised in the following table.

TRADER

SOCIAL RESPONSIBILITIES

Responsibility:	To register and de-register presence in marketplace
Origin:	Seller-Registrant
Problem:	Sending Message [to: Broker, about: Commodities-For-Sale]
Solution:	Equip agent with appropriate co-ordination protocol.
Explanation:	By default Task Agents will reactively respond to Facilitator requests. To advertise proactively the agent will need to be supplied with a co-ordination protocol that sends a message to the Broker notifying it of all commodities available for sale.
Responsibility:	To request information on known sellers
Origin:	Buyer-Inquirer
Problem:	Sending Message [to: Broker, about: Commodities-Wanted]
Solution:	Automatic – part of default Task Agent functionality
Responsibility:	To send bids to potential vendors
Origin:	Buyer
Problem:	Initiate Dialogue [with: Seller, about: Commodity-For-Sale]
Solution:	Equip agent with co-ordination protocol

Explanation:	To initiate and engage in a transaction dialogue these agents will need an appropriate co-ordination protocol. For the buyer the most appropriate is the **Initiator** version of the FIPA defined Multi-Round Contract Net
Responsibility:	To receive and respond to bids
Origin:	Seller
Problem:	Engage in Dialogue [with: Buyer, about: Commodity-For-Sale]
Solution:	Equip agent with co-ordination protocol
Explanation:	To reciprocate in a dialogue the selling agent will need a complementary co-ordination protocol. In this case the most appropriate is the **Respondent** version of the Multi-Round Contract Net.

Table 16: Trader - Social Responsibilities

From the above table we can see that one problem is solved by default by virtue of the functionality of the generic FIPA agent, leaving three other solutions that need to be realised.

Finally, to complete the design of the Trader agent we need to consider its domain responsibilities.

TRADER

DOMAIN RESPONSIBILITIES

Responsibility:	To facilitate the entry of user transactions
Origin:	Buyer
Problem:	Information Entry [Commodity, Price, Number]
Solutions:	➢ Add Buying User Interface
Explanation:	Buying instructions can be issued in the form of goals, which can

	be entered through the agent's standard goal entry user interface.
Responsibility:	To interpret seller responses
Origin:	Buyer
Problem:	Evaluate [Commodity, Price]
Solution:	➢ Equip agent with negotiation strategies
Explanation:	Agents can possess strategies that influence their dealings with others. In this case the **GrowthStrategy** seems the most appropriate, it will begin bidding at a low price and gradually increase it until a vendor accepts or a certain period of time elapses.
Responsibility:	To exchange payment and ownership
Origin:	Buyer, Seller
Problem:	Transfer [Money, Commodities]
Solution:	➢ Automatic (simulated by sending facts) *or* ➢ Implement Domain Function
Explanation:	Once agreed, a transaction can be simulated by both parties exchanging messages that represent payment and the purchased commodity. A more realistic alternative is to perform the through some external program, a so-called 'Domain Function'. This could involve a transaction settlement system like Visa, and a delivery system like ***TCS***.
Responsibility:	To facilitate the entry of user selling preferences
Origin:	Seller
Problem:	Information Entry [Commodity, Price, Number]
Solution:	➢ Add Selling User Interface
Explanation:	As for entry of buying preferences

Responsibility:	To interpret incoming bids
Origin:	Seller
Problem:	Evaluate [Commodity, Price]
Solution:	➢ Equip agent with negotiation strategies
Explanation:	Like buying behaviour, selling behaviour is governed by negotiation strategies. The strategy is likely to be used in conjunction with trading expertise like seasonal demand, pricing histories etc. In this case the **DecayFunction** seems appropriate, this will lower the asking price gradually until it matches an incoming bid.

Table 17: Trader - Domain Responsibilities

By now, we have identified the means to realise all the agents' responsibilities. So we can move onto the next stage of the design process: knowledge modelling.

4.3.2 Knowledge Modelling

The next stage of the design process is to model the declarative knowledge that will be used by the agent roles. This stage should result in the concepts inherent to the application (termed Facts within ZEUS), their attributes and possible values (also known as constraints).

4.3.2.1. Concept Identification

The key concepts in the E-mart application are mentioned in the specification, namely apples, oranges, pears, bananas and melons. As these concepts refer to physical instances rather than abstract ones they will inherit from the **Entity** fact, this is a child of the root **ZeusFact** concept that provides all its children with a cardinality attribute that represents the number of each concept in existence.

All Entity facts also possess an attribute that refers to the concept's inherent value. In our trading application this can be used to store information about the price of each commodity. (Although this is not appropriate for applications where there is no notion of inherent worth and prices are completely determined by supply and demand).

No other attributes are explicitly mentioned in the specification, although as an exercise the scenario could easily be extended with attributes like sell-by-date, colour, country of origin etc.

4.3.2.2. Typing and Constraints

Constraints provide a means of verification, restricting values to a subset of legal values. In this case the cardinality constraint (must be non-negative) already exists and so no additional constraints need to be defined. Also, as the value of each concept is not necessarily universal (different agents may attach different valuations) there is no need to provide a default value for the unit_cost attribute of each fruit. Instead the fruit valuations will be added when the agents are defined.

Hence the resulting facts, their attributes, types and constraints (also known as the *'Problem Ontology'*) looks like this:

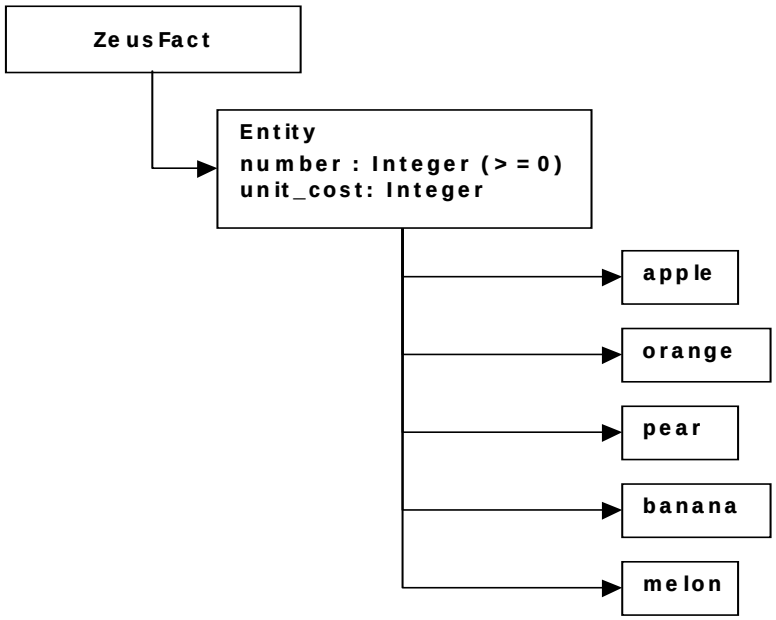


Figure 45: Problem Ontology

Once the ontology is defined we can begin the process of realising the design.

4.4. Application Realisation

Once a design has been produced the next stage is to realise it. The realization process consists of the following activities:

1) Ontology Creation
2) Agent Creation, for each task agent this consists of:
   - Agent Definition
   - Task Description
   - Agent Organisation
   - Agent Co-ordination
3) Utility Agent Configuration
4) Task Agent Configuration
5) Agent Implementation

The purpose of these stages is to translate the design we have derived from the role models into agent descriptions that can be implemented. The remainder of this section will walk-through the stages of the agent development methodology describing how the *E-mart* application was implemented.

4.4.1 Ontology Creation

An ontology is a set of declarative knowledge representing every significant concept within a particular application domain. The significance of a concept is easily assessed, if meaningful interaction can not occur between agents without both parties being aware of it, then the concept *is* significant and must be modelled. Note that for convenience we use the term 'fact' throughout ZEUS to describe an individual domain concept.

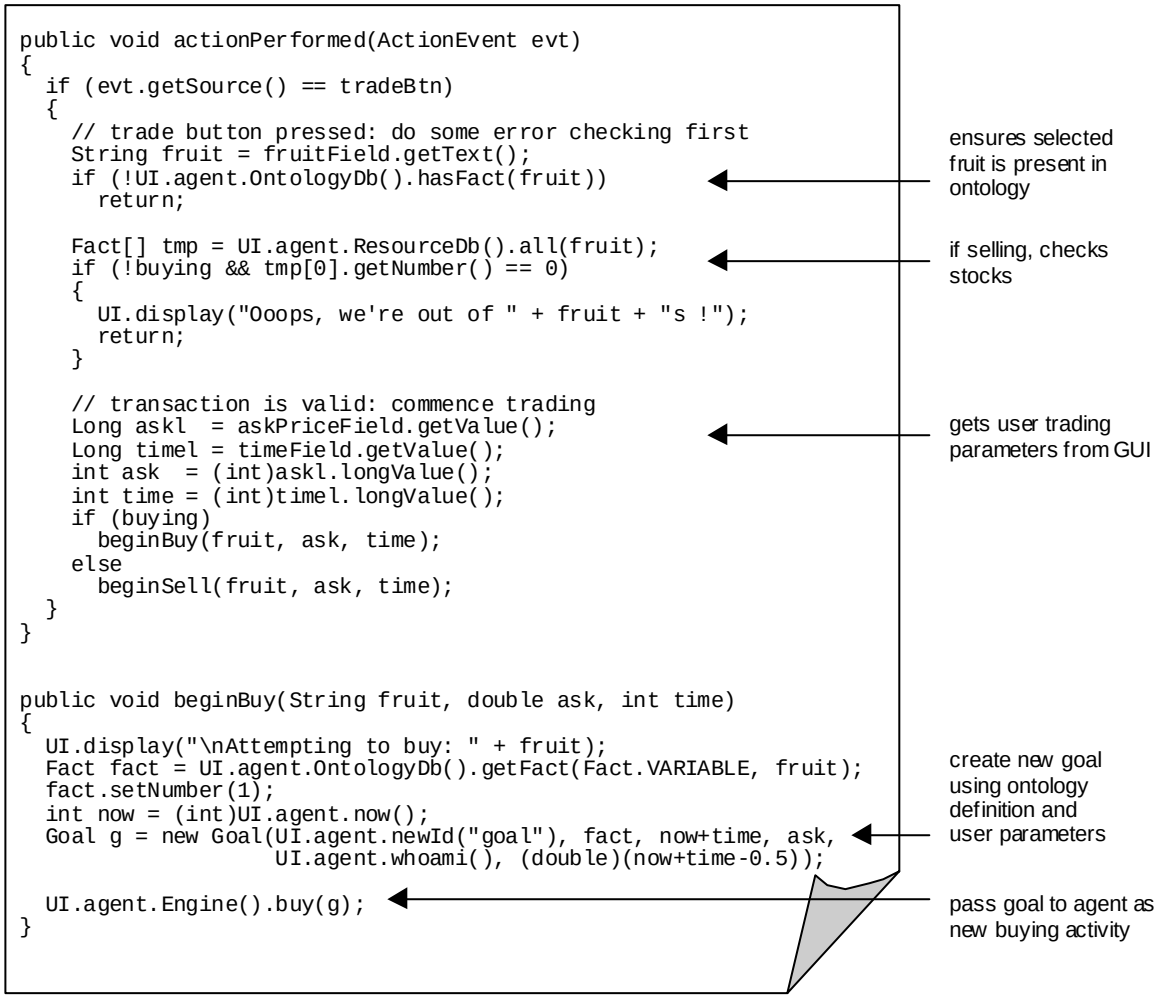
Prior to attempting this stage you should have already identified the following:

- the key concepts within the problem domain
- the significant attributes of each concept
- the types of each attribute

- any constraints on the attributes

4.4.1.1. Acting on User Instructions

If you look at the file TraderFrontEnd.java, most of the user interface should be self-explanatory. One aspect that deserves some explanation is the code that obtains the user instructions from the interface and uses them to change the behaviour of the agent. For instance, when the Trade button on the Buying panel is pressed the following code is called:



Code Listing 2

From this we can see the process of getting an agent to bid for an item of fruit is relatively simple. After checking the desired fruit exists in the ontology, a new fact representing the desired item is created and then used together with the user's buying preferences to create a new goal. Finally the goal is sent to the agent's Co-ordination Engine as a new buying activity, this will cause the agent to begin a conversation using the 'Fipa-Contract-Net-Manager' protocol that we equipped the agent with earlier.

The procedure for triggering selling behaviour is very similar, with a new goal being created to represent the commodity being sold, and which is passed to the sell() method of the agent's Co-ordination Engine.

4.4.1.2. Monitoring Agent Events

Another useful feature of the TraderFrontEnd is to keep the user informed of ongoing trading activities, like the arrival of new bids and the responses made by sellers. This is achieved by implementing the ConversationListener interface; the method that listens for the beginning of a conversation is shown below:

The other method of the interface, conversationContinuedEvent(), is practically identical, although it will be triggered under slightly different circumstances (when a message is sent during an existing conversation).

```
public void conversationInitiatedEvent(ConversationEvent event)
{
  String correspondent = event.getReceiver();
  String mode = " to ";
  if (correspondent.equals(agent.whoami()))
  {
    correspondent = event.getSender();
    mode = " from ";
  }
  Goal g = (Goal)(event.getData().elementAt(0));
  double cost = g.getCost();
  String f = g.getFactType();
  display("Conversation started...");
  display("[" + event.getMessageType() + "]" + mode +
          correspondent + " >> " + f + " @ " + cost);
}


public void conversationContinuedEvent(ConversationEvent event)
{
  .
  .
}
```

determines the conversation event's origin

obtains the subject of the conversation

displays selected information about the current conversation

Code Listing 3

4.4.1.3. Monitoring Changes to Resources

As well as receiving and acting on instructions, the TraderFrontEnd needs to present accurate information on the state of its agent's resources. In this example the changes in question involve stock arriving and being dispatched, and money being credited or debited. To cause a change, press one of the "Supply" buttons that appear beside the stock level of each commodity, this simulates the delivery of one item of that commodity by adding a new resource to the agent's own Resource Database (ResourceDb). This will trigger a *factAddedEvent*, which can be listened for by the TraderFrontEnd; in addition the *factDeletedEvent* will notify listeners that a particular resource has been removed and no longer exists locally. The following line registers the TraderFrontEnd's interest in these events:

```
UI.agent.ResourceDb().addFactMonitor(this,
  FactEvent.ADD_MASK|FactEvent.DELETE_MASK, true);
```

The excerpt of code illustrating the factAddedEvent is relatively simple, it retrieves the fact reference from the trigger event and passes it to a method that updates the appropriate GUI component. Changes to Money resources are slightly more complex, as credit transactions involve the creation of new separate Money resources these must be merged with the agent's existing Money resource to create a true figure for the amount of money owned.

4.5 Running the E-mart Application

Assuming that all the agents will be running on the same machine, launching the application will involve the following process:

- Enter the command **'run1'**, this will execute the run1 script and start the Agent Name Server (ANS). No errors should be reported, and a new Java interpreter process should be running in the background. If a problem has occurred, check your system's configuration (e.g. CLASSPATH setting, install directory in the .zeus.prp file etc.)
- Enter the command **'run2'**, this will start the OrchardBot, SupplyBot and ShopBot agents, as a result three more Java processes should now be running in the background. Whilst on screen an Agent Viewer window and a Trading GUI window will appear for each agent.
- Enter the command **'run3'**, this will start the Visualiser and Facilitator agents. Two additional Java processes should be started, and a Visualiser Hub window should appear on screen.

Although the agents can run independently of the Visualiser, you may find it useful to launch the Society Viewer window to observe how the agents interact with each other. The Statistics tool may also be useful to plot price fluctuations over time, whereas for example the Report Tool has no relevance as no tasks are actually performed by the agents.

Once launched the E-mart agents will display their front-end GUIs and await user instructions; they will do nothing autonomously apart from replying to register themselves with the Name Server, (and the Visualiser if one exists). Now users have two options, they can offer an item for sale or formulate a new bid.

4.5.1. Offering an Item for Sale

The procedure for tendering goods is relatively simple:

- Go to the “Sell Fruit” tab pane.

- Now choose the commodity to be sold either by typing its name into the **Sell** textfield or clicking on the **Choose** button and selecting it from the current ontology hierarchy.
- Next enter the reserve price of item (this is the lowest price you are willing to accept) in the **Reserve Price** field. This value will be kept secret and used by the negotiation strategy to create an asking price, which will be the price quoted to potential buyers.
- The elapsed time before the sale should be completed can be entered in the **Deadline** field, this sets the period of time that potential buyers have to submit bids and negotiate.
- Finally once the selling parameters have been entered click on the **Trade** button: this will notify the broker of the item for sale, and thus any interested parties.

Once offered for sale the agent waits for incoming bids until the deadline period expires, whereupon the item will be withdrawn from sale. How the agent responds to incoming bids is described later in this section.

4.5.2. Bidding for an Item

The procedure for bidding for goods is quite similar to that for selling:

- Go to the “Buy Fruit” tab pane.
- Now choose the commodity to purchase either by typing its name into the **Buy** textfield or clicking on the **Choose** button and selecting it from the current ontology hierarchy.
- Next enter the highest price you would be willing to pay for this item in the **Maximum Price** field. This value will be kept secret and used by the negotiation strategy to create a bid price, which will offered to the seller.
- The elapsed time before the purchase should be completed can be entered in the **Deadline** field, this sets the period of time by which negotiation should have concluded.
- Finally once the bidding parameters have been entered click on the **Trade** button: the agent will then consult the broker for the presence of matching items for sale.

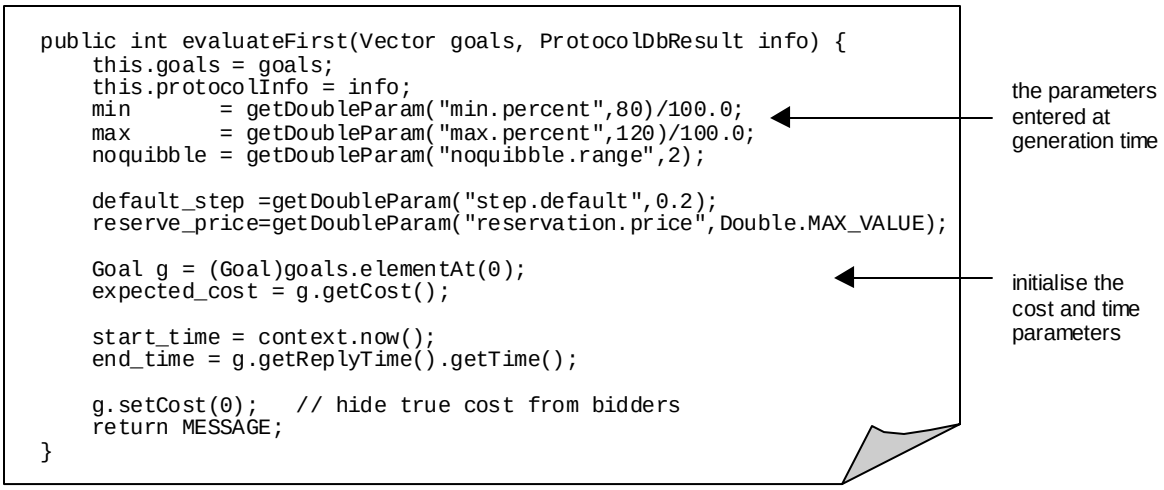
If the commodity to buy is currently on the market the broker will send the interested bidder the identity of the seller, which it can contact directly and begin negotiation. Otherwise the bidding agent will ask to be informed if any appropriate items are placed on the market. Hence the order in which items are placed on the market is not significant – bidders can bid before sellers offer and vice versa, providing they do so before each other's deadline period expires.

4.5.3 How Negotiation Works

In the E-mart application sellers to determine their replies to incoming bids using the LinearInitiatorEvaluator strategy, whilst LinearRespondentEvaluator is used by potential purchasers to formulate their bids. These strategies are implemented in the eponymous files found in the zeus.actors.graphs package.

These strategies are linked to the agent's co-ordination engine by their implementation of the *StrategyEvaluator* interface class, and two significant methods: evaluateFirst() and evaluateNext(). The former method is used to calculate the initial negotiating position, whilst the latter uses the parameters set when the agent was created (as described in the COORD-2 section of the Realisation Guide), or runtime parameters (like those entered through the agent GUI). An implementation of these methods (from the LinearInitiatorEvaluator.java file) are shown:

The evaluateFirst() method is intended to initialise negotiation rather than implement it: this is the function of the evaluateNext() method, as shown below:



Code Listing 4

Typically the evaluateNext() method modifies the negotiation position in the light of new information, either in the form of new bids, or the passage of time bringing the deadline closer. It functions by changing the attributes of the current goal, and determines the next state of the agent's co-ordination engine by returning one of the following three types of message:

- OK - i.e. the bid or proposal was acceptable, agents now move into the settlement stage
- MESSAGE - i.e. the bid was not acceptable but close enough to warrant a counter-proposal
- FAIL - this ends the negotiation, typically because the bid was too low and the time deadline has expired

If the negotiation results in a price acceptable to both parties the co-ordination engine of each agent will move into the settlement stage. Here the resource in question is removed from the seller's ResourceDb and transferred to the buyer, whilst simultaneously the money resource of the buyer is debited and the amount credited to the seller

```
public int evaluateFirst(Vector goals, ProtocolDbResult info) {
    this.goals = ds.goals;
    if ( !ds.msg_type.equals("propose") )
      return FAIL;

    Goal g = (Goal)ds.goals.elementAt(0);
    offer = g.getCost();
    double now = context.now();

    if ( first_response ) {
       first_response = false;
       dt = now - start_time;
       end_time = end_time - dt;

       expected_cost = Math.max(offer,expected_cost);
       min_price = min*Math.min(reserve_price,expected_cost);
       max_price = Math.min(max*expected_cost,reserve_price);

       price = min_price;
       step = (max_price-min_price)*dt/(end_time-start_time);
       step = Math.max(step,default_step);
    }

    if ( offer < price + noquibble ) return OK;
    if ( !first_response )           price += step;

    if ( price < min_price )
      return FAIL;
    else if ( offer < price + noquibble )
      return OK;

    if ( now + dt >= end_time ) {
      if ( offer < max_price )
        return OK;
      else
        return FAIL;
    }

    g.setCost(price);
    return MESSAGE;
}
```

determine asking price and increment value

revise asking price slightly

running out of time, attempt to settle

Code Listing 5

Chapter 5

Future Work

5.1. Effective File Replication using Mobile Agents

The task at hand with the deadline of December this year is to provide complete mobility to the FIPA Agent and conform this to the JINI environment. This then shall help in the transfer of heavy data across a distributed network environment. As an example, lets assume that as a result of a processing job at the Tier0 Regional Center, a data base file, which may interest other regional center, is produced. Using the distributed Station Servers a remote notification event is sent to all the Regional Centers. Each remote Regional Center interested to have a local copy of this file will create a Mobile Agent. The aim of the each such Mobile Agent is to "bring home" the interesting file in an effective and cooperative way with others agents representing interested Regional Centers. The aim is to effectively use the internet connection (e.g. transfer the file(s) only once on trans-Atlantic lines), limit the number of concurrent ftp transfers from the same center and transfer the file(s) to all interested Regional Centers as fast as it can be done at a certain network availability. Each created agent is transferred (using a transaction manager to insure integrity) to the place where the interesting file was created. The Agent contains a set of specific constrains and can get additional information about the current connectivity of the center it represents. For a predefined time window agents interact at this Regional Center and try to create clusters. The clustering shall be done based on available bandwidth and additional constrains (e.g. maximum number of concurrent transfers allowed by the host center). A schematic view of this approach is shown in Figure 46.

The clusterisation will practically allow an agent to include other ones. As an example if the bandwidth from Tier0 to Tier1-A is higher, the agents originating from Tier2-A and Tier2-B will be included in the Tier1-A agent. Additional constrains may be used in this clustering process.

At the end of this time window (which allows interested agents to move and interact) the clustering scheme is "published "on JavaSpace to allow eventually interested parties to join later (in case of local network problems at the notification time) and in this way "late agents" and can also be included. Once the clusterisation is done the host center will

decide how transfers may may be done concurrently, will schedule them and will allow them to start the execution.

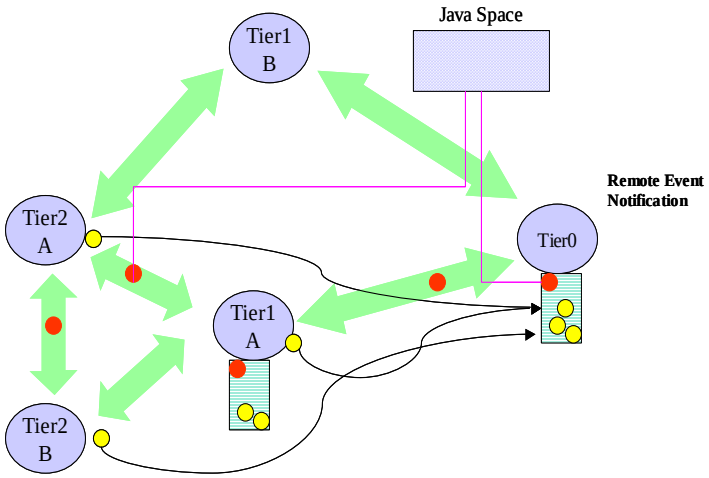


Figure 46: Performing effective file replication using cooperative Mobile Agents

Each agent's task is to "bring home" the interested file. The execution will practically start a FTP transfer (in push mode) to its center and the agent will monitor and control the ftp transfer. When the transfer is finished the agent will move back to its base center following the file transfer. If it contains other agents from the previous clusterisation the process is repeated. In the example considered, the transfers to the two Trie2 centers may be done in parallel or sequentially from the Tier1-A center, or again as a sequential process Tier2-A(B) -> Tier2-B(A) depending on the network conditions/loads and the Tier1- A constrains. In this scheme the data transfer is done with dedicated tools for efficient transfer (parallel streams ftp) and the control of these transfers is done trying to optimize it at the entire system level and adapt it to the particular state of the available connectivity between Regional Centers.

References

1 FIPA *FIPA 97 Specifications Part 1 & 2* Foundation for Intelligent Physical Agents Version 1.2 2000

2 William R. Cockayne. *Mobile Agents,* Manning Publications, 1998.

3 Jeffery M. Bradshaw. *Software Agents,* The MIT Press, 1997.

4 Joseph P. Bigus, Jennifer Bigus, *Constructing Intelligent Agents with Java,* Wiley Computer Publishing, 1998.

5 Bill Venners, *The Architecture of Aglets,* available at http://www.artima.com/underthehood/aglets.html

6 Iosif Legrand, *A Distributed Server Architecture for Dynamic Services,* Draft 2.0

22nd May 2001 cil

7 Jaron Collins, Zeus *Technical Manual,* British Telecommunications plc.

8 Jaron Collins, Zeus *Role Modeling Guide,* British Telecommunications plc.

9 Natalya F. Noy and Deborah L.McGuinness, *Ontology Development 101: A guide to creating your first ontology,* Stanford University, Stanford.

10 Pattie Maes, *Agents that reduce work and information overload,* available at http://pattie.www.media.mit.edu/people/pattie/CACM-94/CACM-94.pl.html

11 H. Nwana et al., "*ZEUS: A Toolkit for Building Distributed Multi-Agent Systems*

" Artificial Intelligence, vol. 13, no. 1,1999,pp. 129-186.

12 Bradley James Rhodes, *Just-In-Time Information Retrieval,* Massachusetts Institute of Technology Cambridge, MA, 1996

13 H. Lieberman , N. van Dyke, A. Vivacqua, *Let's browse: a collaborative browsing agent,* Media Laboratory, Massachusetts Institute of Technology, 1997

14 Milla Mäkeläinen, *Agent Mobility in FIPA-OS,* Nottingham Trent University, 2000

15 Katia Sycara, Kieth Decker, Anandeep Pannu, Mike Williamson, *Distributed Intelligent Agents,* The Robotics Institute, Carnegie Mellon University

16 Jay Budzik and Kristian J. Hammond *User Interactionswith Everyday Applications as Context for just-in-time information Access* , Intelligent Information Lab, Northwestern University.

17 Hyacinth S. Nwana and Divine T. Ndumu, *A perspective on Software Agents Research,* applied research and technology dept, BT Labs, UK

18 Sycara, Anandeep Pannu, Mike Williamson, *Designing behaviours for*

Information Agents, The Robotics Institute, Carnegie Mellon University

19 Yanyan Yang, Omar F. Rana, David W. Walker, Christ Georgousopoulos,Roy Williams, *CACR Technical Report*, CACR – 186, March 2000

20 Federico Bergenti and Agostino Poggi, *Exploiting UML in the Design of Multi Agent Systems*, Diparti di Ingegneria dell' Informazione, Universita degli Studi di Parma Parco Area delle Scienze 181A, Parma, Italy

21 Hyacinth S. Nwana and Divine T. Ndumu , *A Prespective on Software Agents Research,* Applied Research and Technology Dept. British Telecommunications Laboratories , Suffolk, IP5 3RE, UK

22 Robert A. Flores-Mendez , *Towards a standardization of multi-agent system frameworks.*

23 Von der Fakultät Informatik der Universität Stuttgart, zur Erlangung der Würde eines, Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigten Abhandlung, *Control Algorithms for Mobile Agents*