FIPA-OS Developers Guide

Publication History

18 October 2000
First release.

2 February 2001
Updated to include details of the XML-specific databinding code

Table of Contents

About this document

What is this document?

This document attempts to explain the architecture of the FIPA-OS platform to enable FIPA-OS developers to update and expand the functionality of FIPA-OS by providing an understanding of its design. This document is based upon the FIPA-OS v1.3.2 distribution, available via our website [9].

Intended Audience

This document is intended for anyone attempting to incorporate new functionality or modify existing functionality into the FIPA-OS platform.

Reading Guide

It is strongly recommended that the reader should look at the FIPA-OS web site at http://fipa-os.sourceforge.net/ to understand the rationale behind this platform and for information on future updates.

Developers using FIPA-OS are encouraged to provide extensions, bug fixes and feedback to help improve the planned future releases. All such input should be contributed to the Open Source project via the SourceForge site at http://sourceforge.net/projects/fipa-os/. You are required to register as a developer to access some of the services at the SourceForge site. General issues and thoughts can be discussed via the FIPA-OS mailing list on fipa-os-developers@lists.sourceforge.net although you must register at http://lists.sourceforge.net/mailman/listinfo/fipa-os-developers on this list before you can send and receive messages. An archive of the messages sent to this list can also be viewed from http://www.geocrawler.com/redir-sf.php3?list=fipa-os-developers. Should you experience difficulties using this list, then please contact the FIPA-OS co-ordinators at fipaos@emorphia.com. Please consult the *FIPA_OS_Public_Licence.txt* file for further details on the requirements for using, extending and evolving FIPA-OS.

Conventions used

Within the text filenames appear in *italics*. In examples where users should enter data, the suggested data appears in **bold**. For examples of entering data at the command prompt, variables are encapsulated in < and > and optional data is encapsulated in **[** and **]**, e.g. [<comms-transport>] is an optional parameter which can be specified at the command prompt.

Terminology

ACL	Agent Communication Language [3]
AID	Agent Identifier [1]
API	Application Programming Interface
CCL	Choice Constraint Language [7]
FIPA	Foundation for Intelligent Physical Agents [11]
HAP	Home Agent Platform [1]
MTP	Message Transport Protocol [4]
MTS	Message Transport Service [4]
RDF	Resource Definition Framework [13]
SL	Semantic Language [2]
XML	extensible Markup Language [12]

Chapter 1
FIPA-OS Overview

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 1 highlights the available components and there relationship with each other (NOTE: The Planner Scheduler is not currently available).

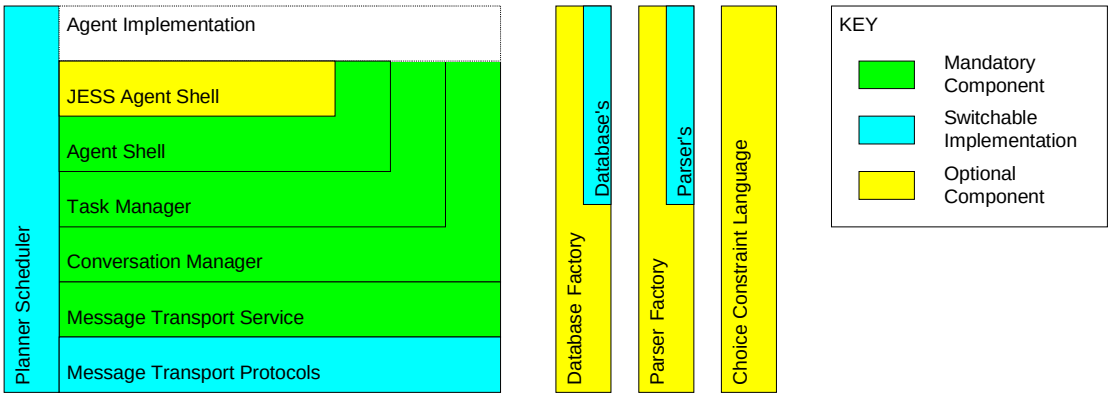

Figure 1 - Components within FIPA-OS

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 [5])
- Database's
    - MemoryDatabase
    - SerializationDatabase
- Parser's
    - SL
    - ACL
    - XML
    - RDF

Chapter 2 details the mandatory components of the FIPA-OS platform.

Chapter 3 details the optional components of the FIPA-OS platform.

Chapter 2
Core Components

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.

fipaos.ont.fipa.ACL

This class represents the abstract notion of an ACL message within the FIPA ACL specifications [3]. By default it supports parsing and deparsing of the FIPA standard string encoding for ACL [6], 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 2).

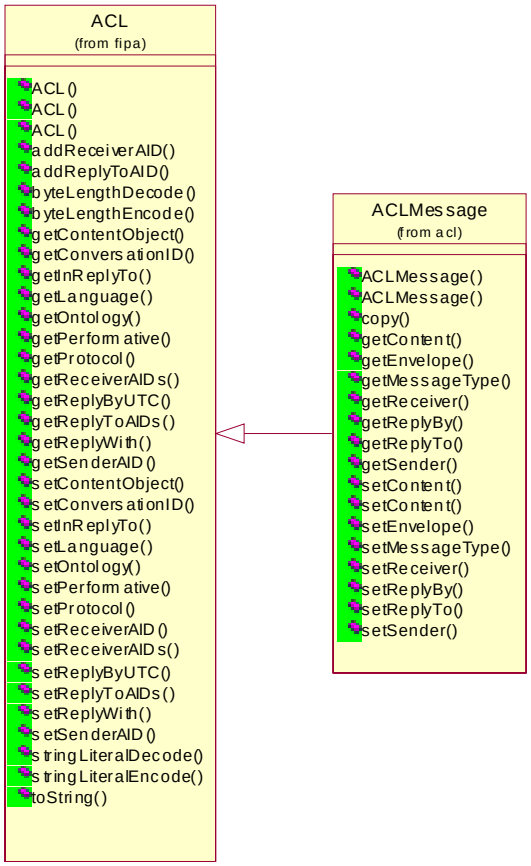

Figure 2 - ACL & ACLMessage Objects

fipaos.ont.fipa.fipaman.Envelope

This class provides an abstract representation of the FIPA defined Envelope from the MTS specification [4]. An `Envelope` object by default provides access to the last assigned values of each

of 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.

fipaos.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).

fipaos.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).

Agent Shell (FIPAOSAgent)

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

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 initialisation 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 3 highlights the relationships between the classes of these core components, and the interfaces used to remove inter-component dependence.

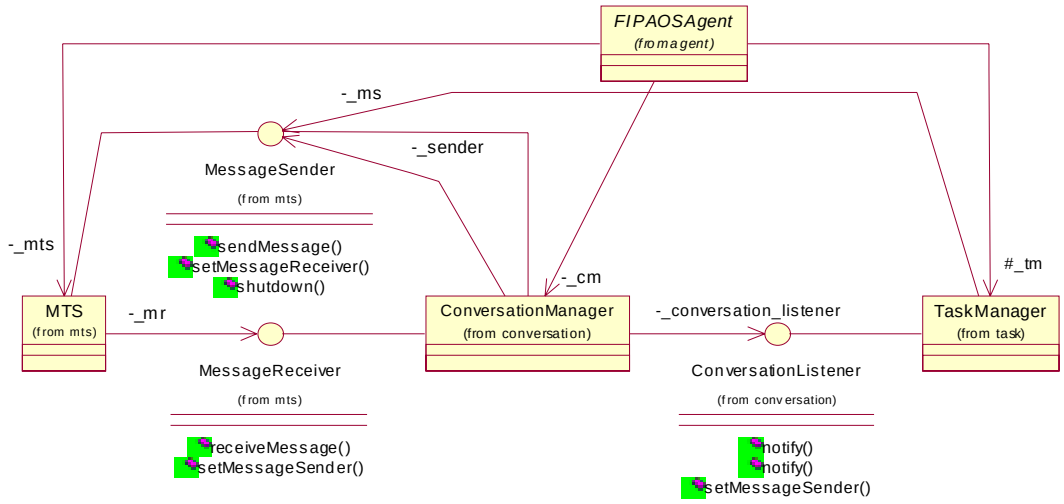

Figure 3 - Core Component Relationships within FIPAOSAgent / Agent Shell

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.

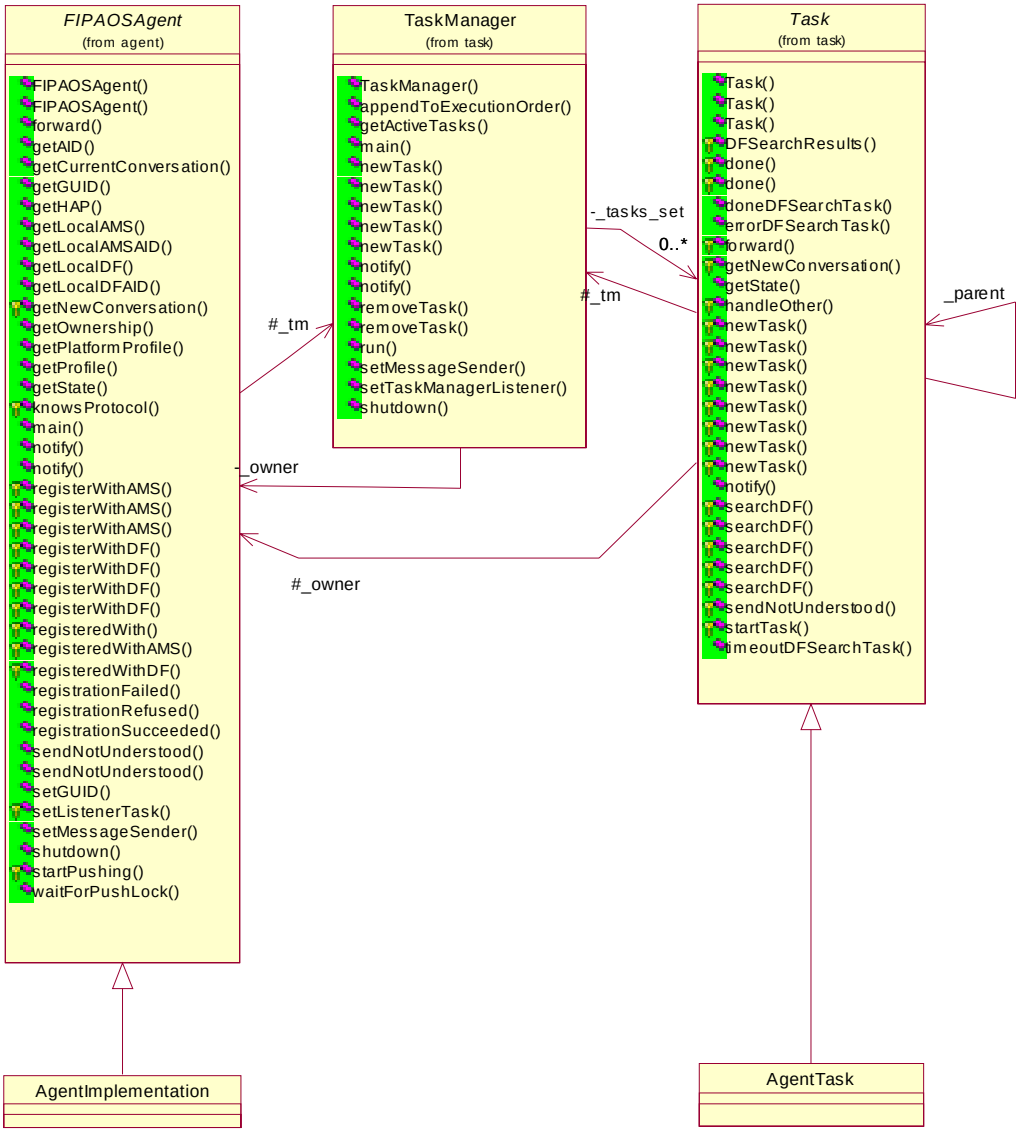

Figure 4 - Agent Implementation and Relationship with Agent Shell

Figure 4 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.

Functionality 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 5 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.

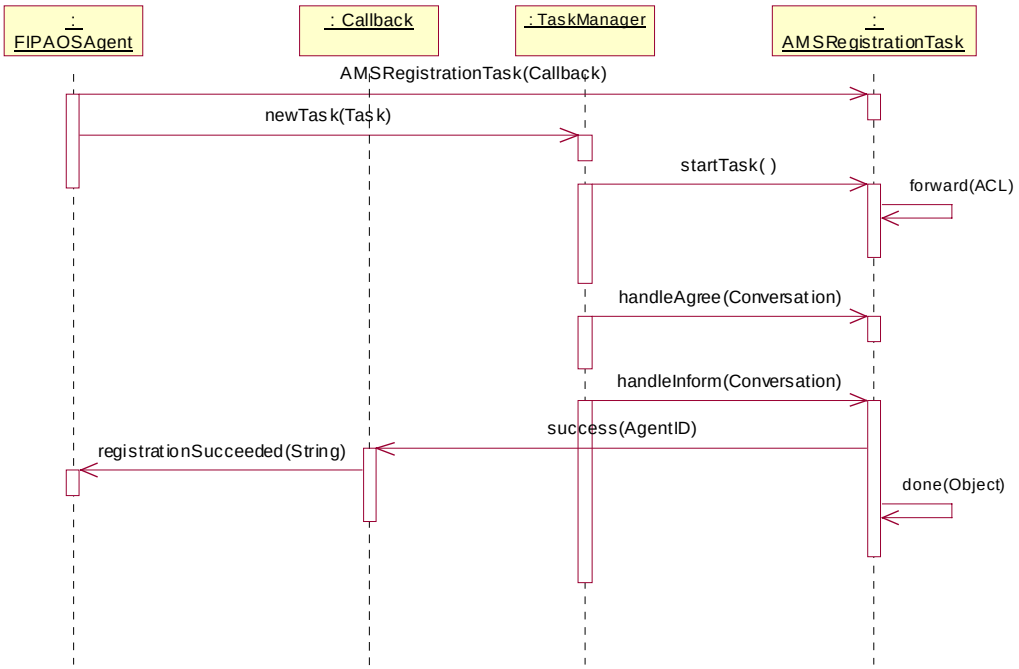

Figure 5 – Logical Interactions when Successfully Registering with AMS

- 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.

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:

[1] These classes are not part of the FIPA-OSv1.3.2 distribution, but are available separately from our SourceForge CVS repository. (They will be/are bundled with later distributions).

- 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.

Composition of the TM

The `TaskManager` itself is composed of several parts, depicted in Figure 6.

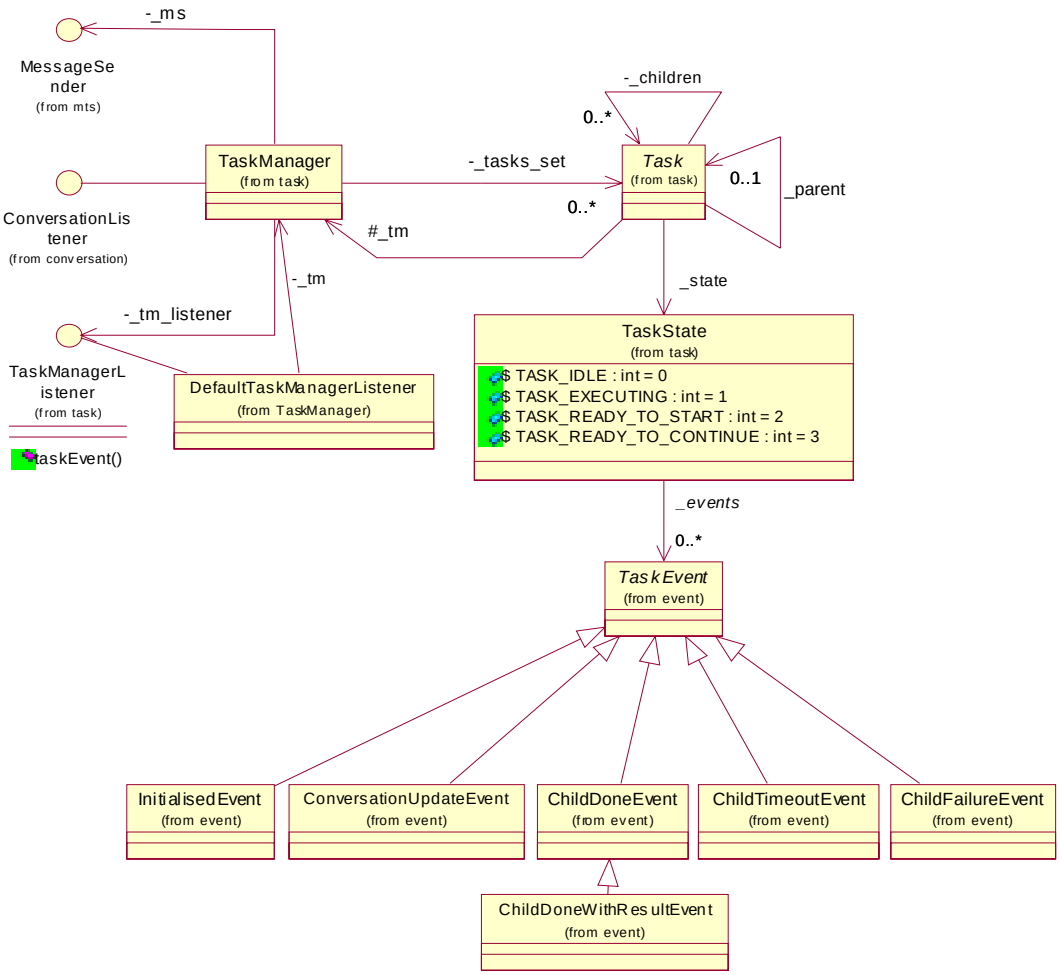

Figure 6 - TaskManager Class Relationships

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.

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.

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).
ConversationUpdateEvent	public void handleX(Conversation)	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

TaskEvent	Listener Method	Description
		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.

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 7 highlights the interactions between a `TaskManager` and a `TaskManagerListener` when `newTask()` is invoked.

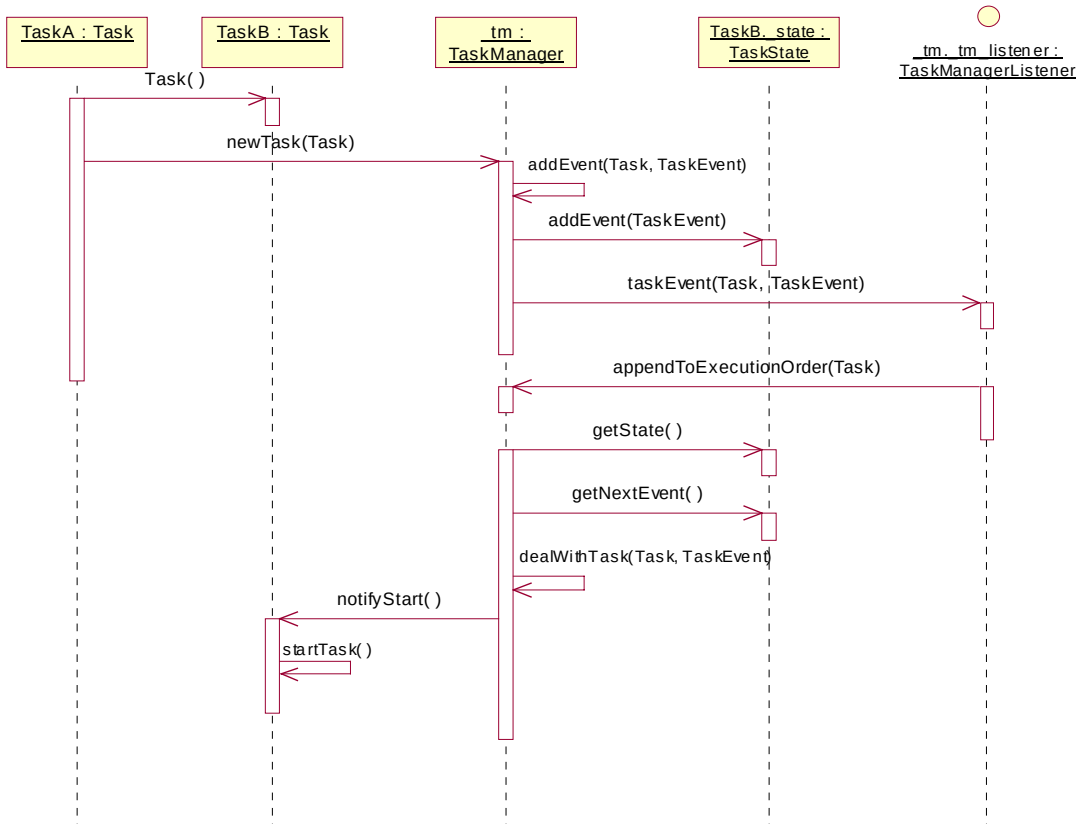

Figure 7 - newTask() - TaskManager and TaskManagerListener Concrete Interactions

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.

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.

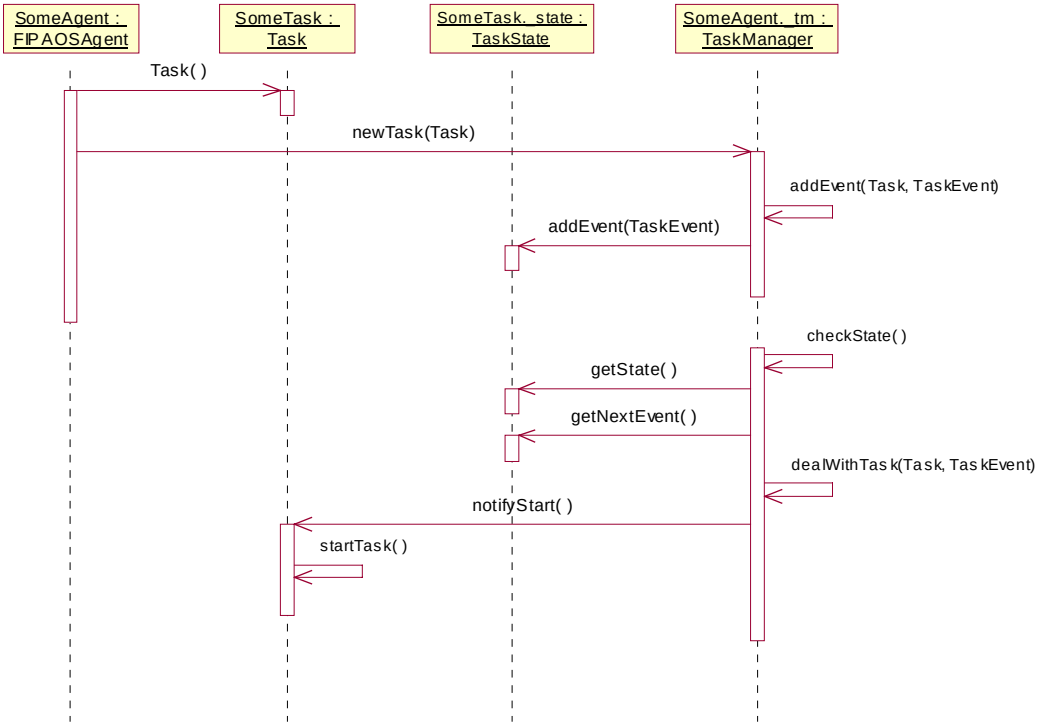

Figure 8 - FIPAOSAgent / Task / TaskManager newTask() Concrete Interactions

As shown in Figure 8, 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 9 highlights the interactions between the parent-Task, TaskManager and child-Task when a Task is started with a timeout, and that timeout is reached.

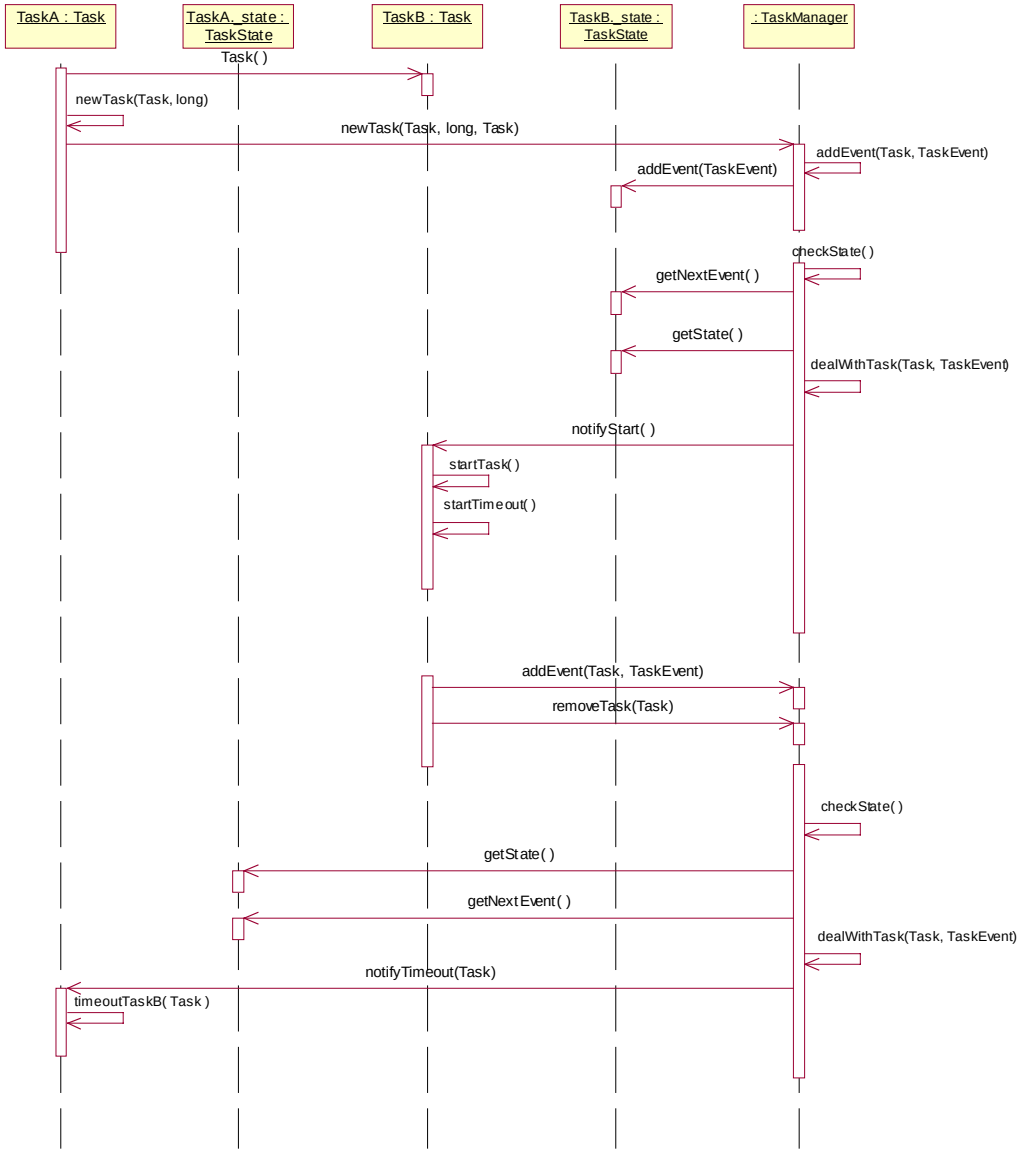

Figure 9 – Task / TaskManager Concrete Interactions when Timeouts Occur

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-`Task`s 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 )`.

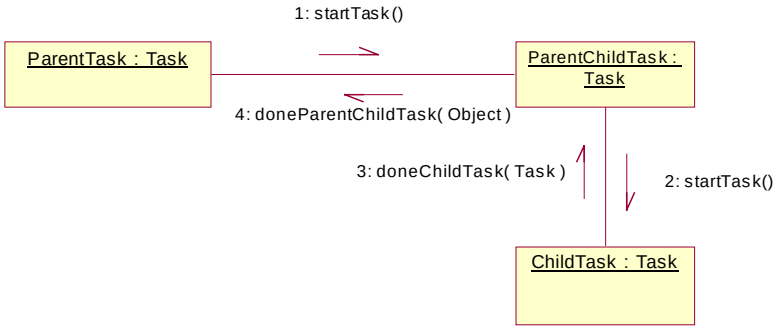

Figure 10- Multiple Nested Parent / Child Completion Logical Interactions

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 a reference to a `MessageSender`, to which it passes all outgoing messages via its `sendMessage()` method.

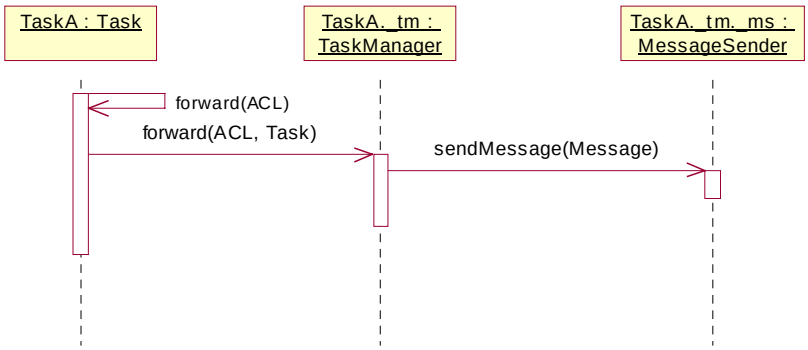

Figure 11 – Concrete Interactions when Forwarding a Message from a Task

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.

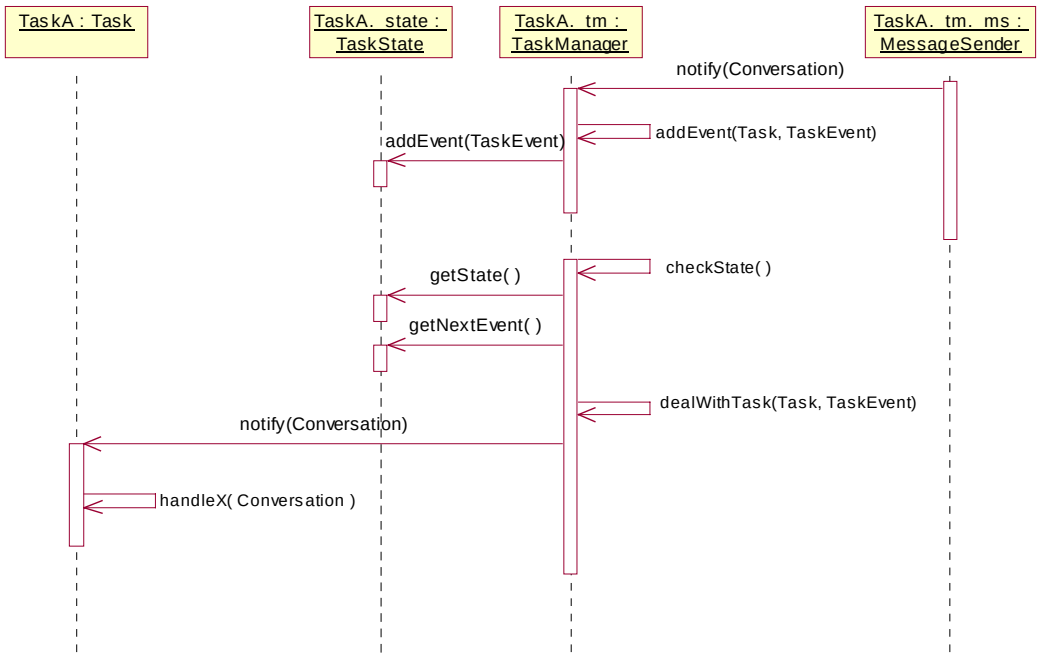

Figure 12 - Concrete Interactions when TaskManager Receives a Message

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.

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.

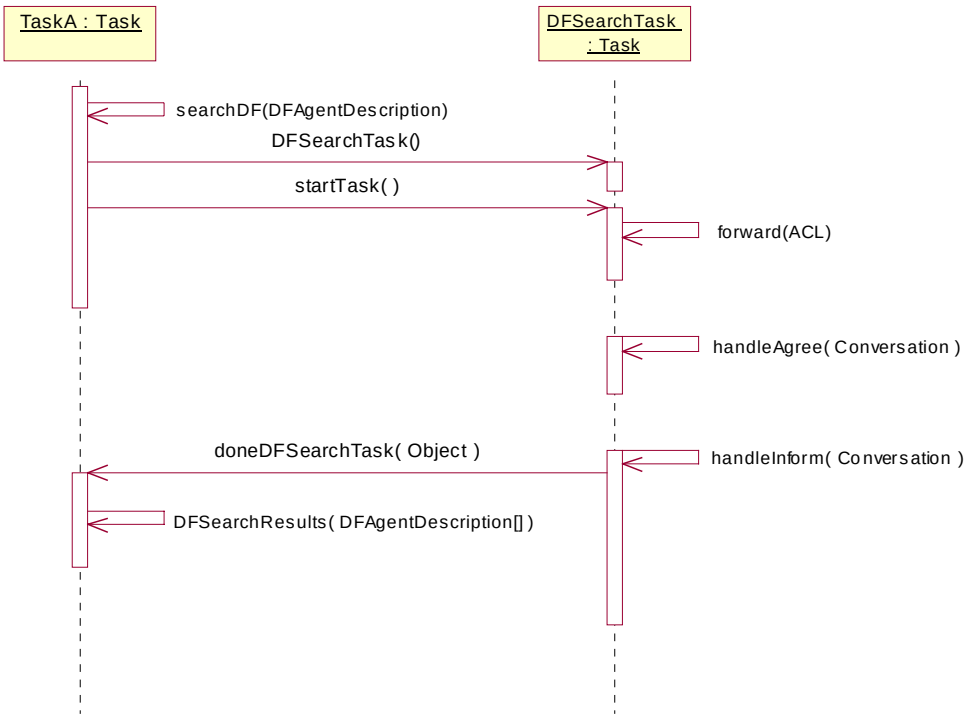

Figure 13 - Logical searchDF() Interactions

- `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`.

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.

Composition of the CM

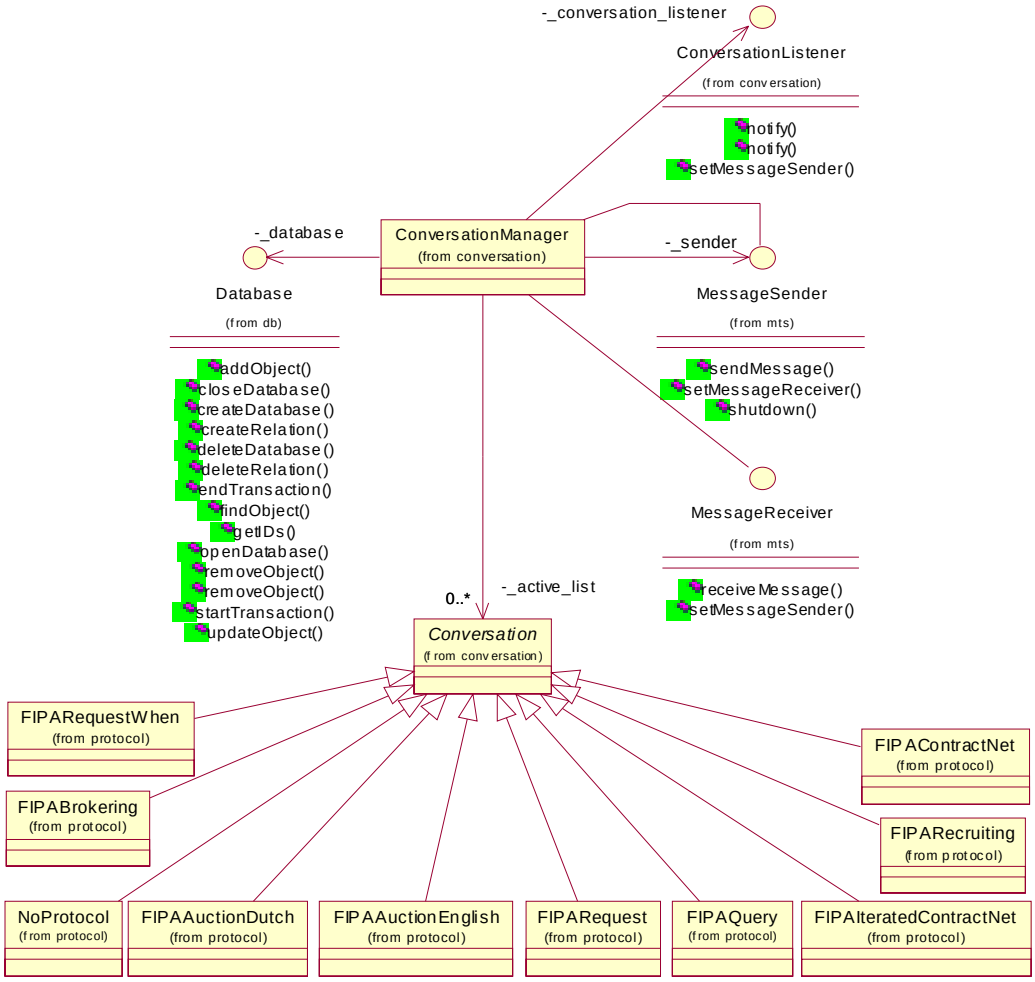

Figure 14 - Conversation Manager Composition

Figure 14 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.

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.

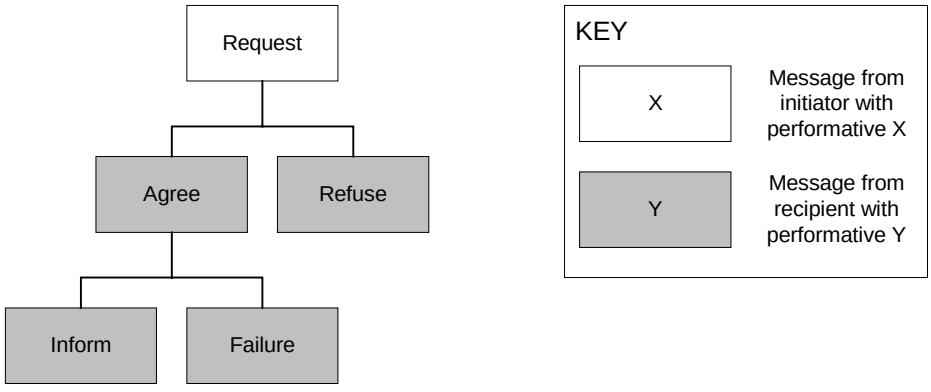

Figure 15 - Example Message Protocol

Figure 15 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 };
```

Messaging

Figure 16 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.

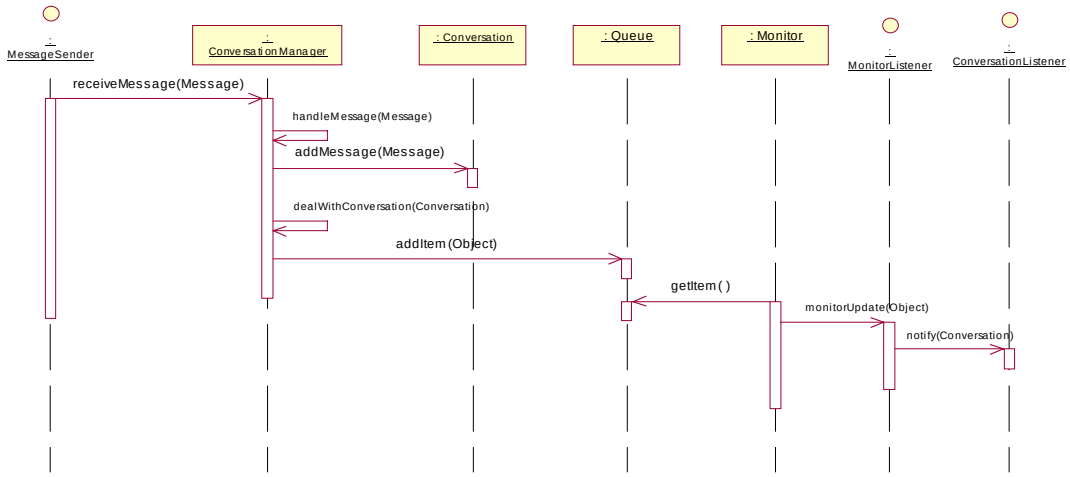

Figure 16 - Interactions when Receiving a Message

Figure 17 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.

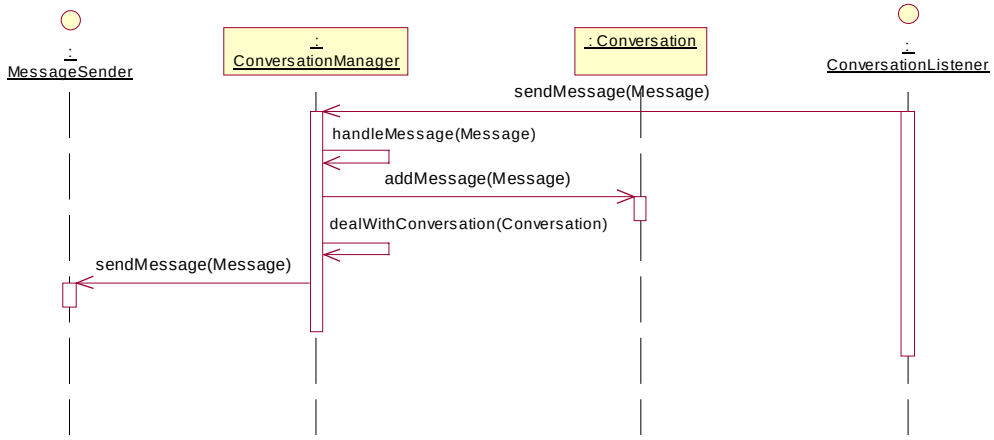

Figure 17 - Interactions when Sending a Message

MTS (Message Transport Service)

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

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.

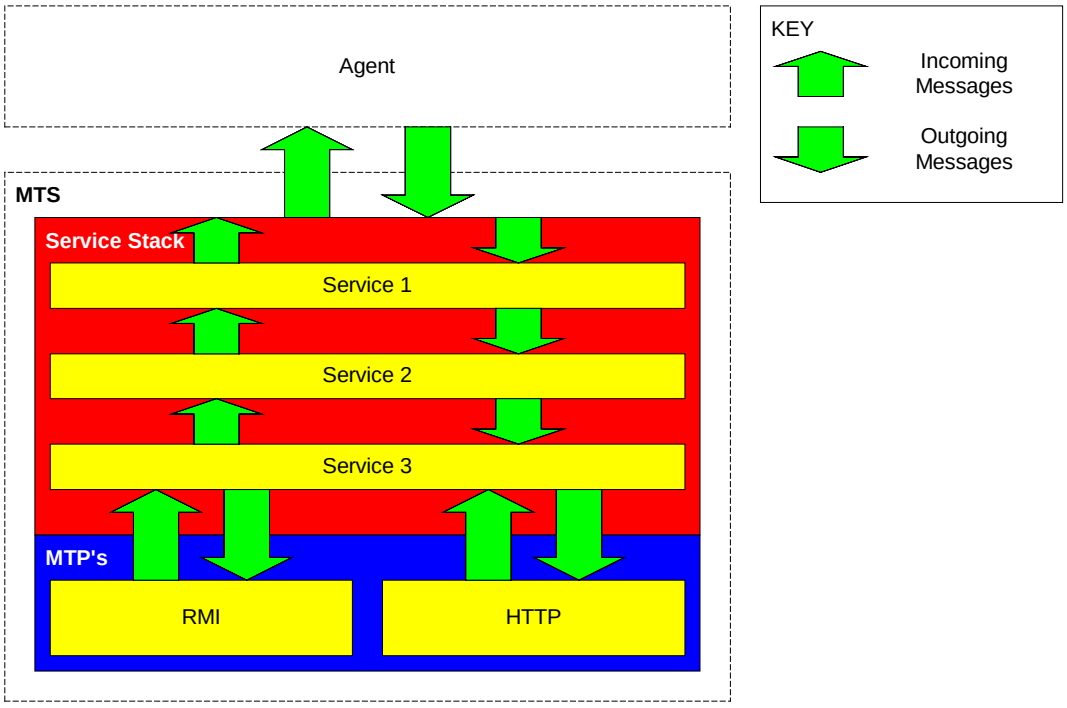

Figure 18- Logical Composition of the MTS

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.

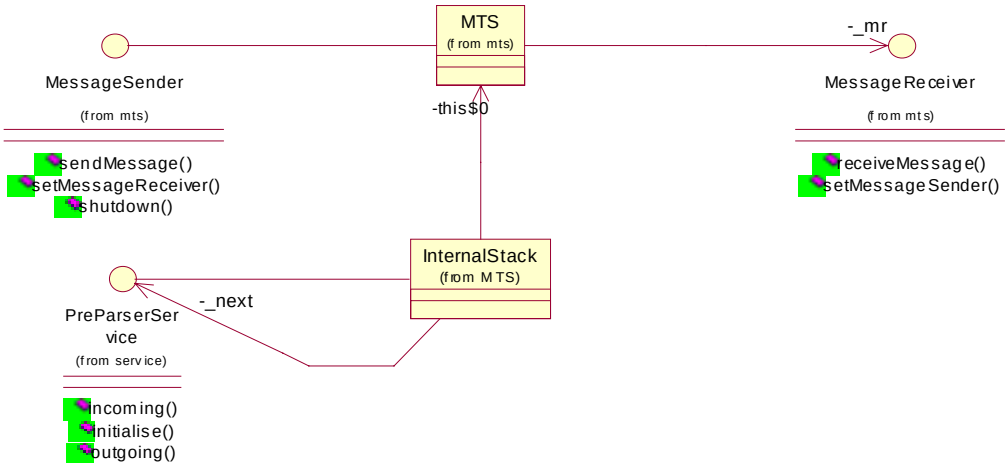

Figure 19- MTS Class Relationships

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.

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 20). 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.

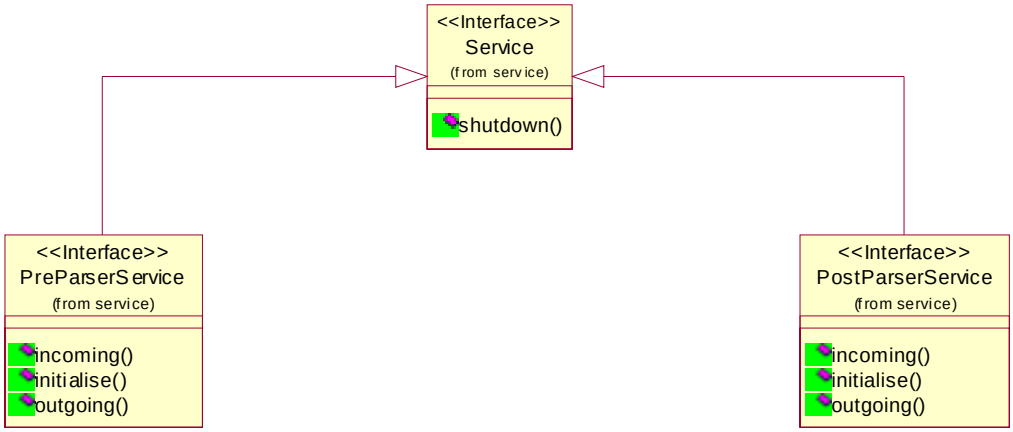

Figure 20 - Service Interface Relationships

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

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.

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.

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).

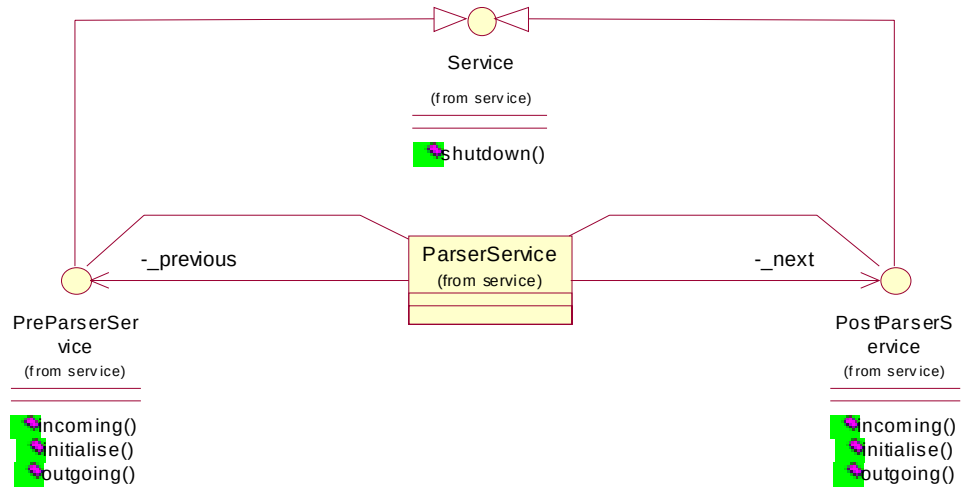

Figure 21 - ParserService Class Relationships

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.

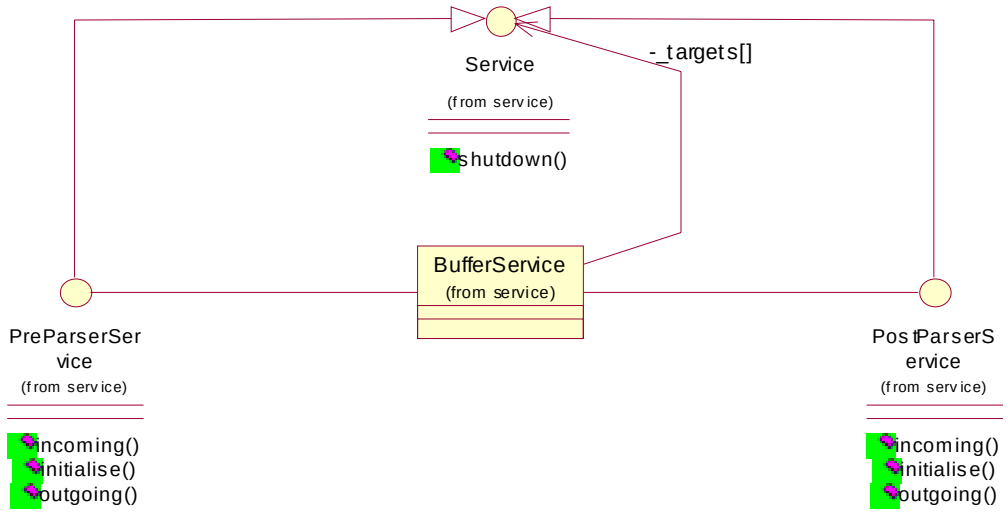

Figure 22 - BufferService Class Relationships

- `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 [4] (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.

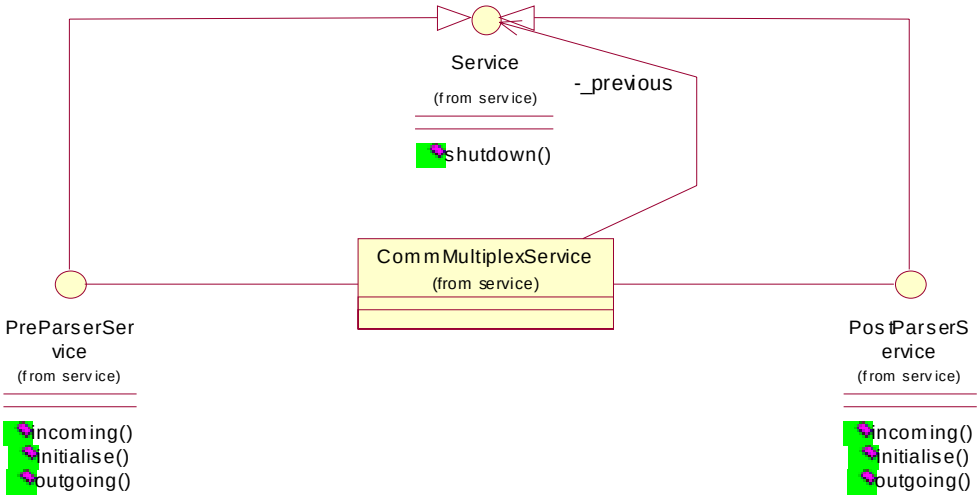

Figure 23 - CommMultiplexorService Class Relationships

- `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.

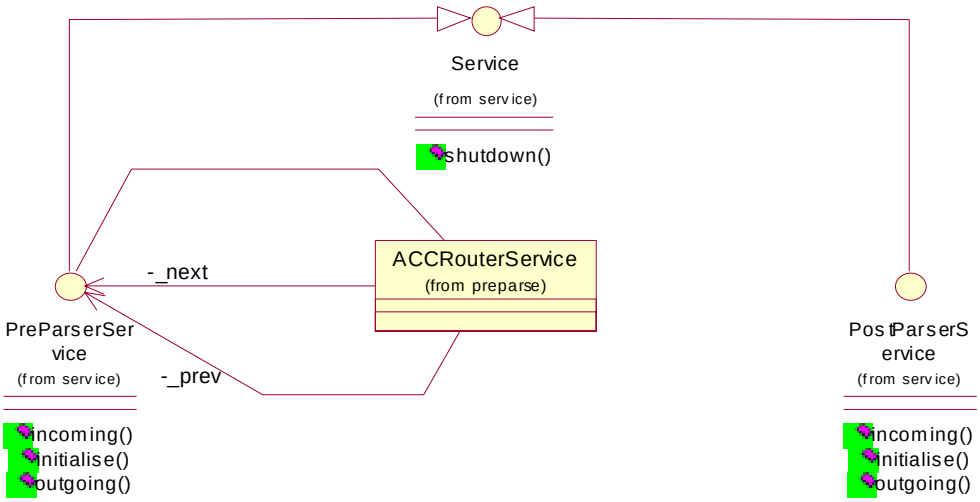

Figure 24 - ACCRouterService Class Relationships

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.

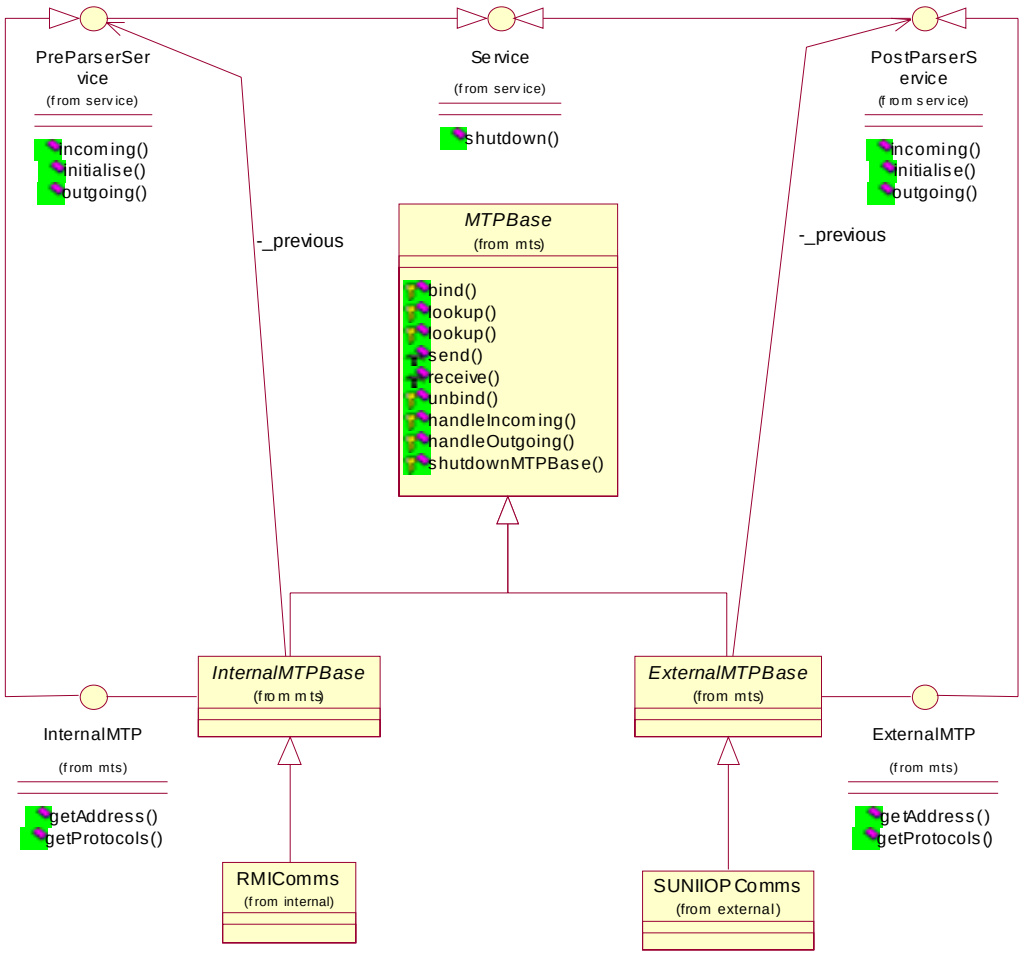

Figure 25 - MTP Class Relationships

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 the `Envelope` 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 *agent@hap*).

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.

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.

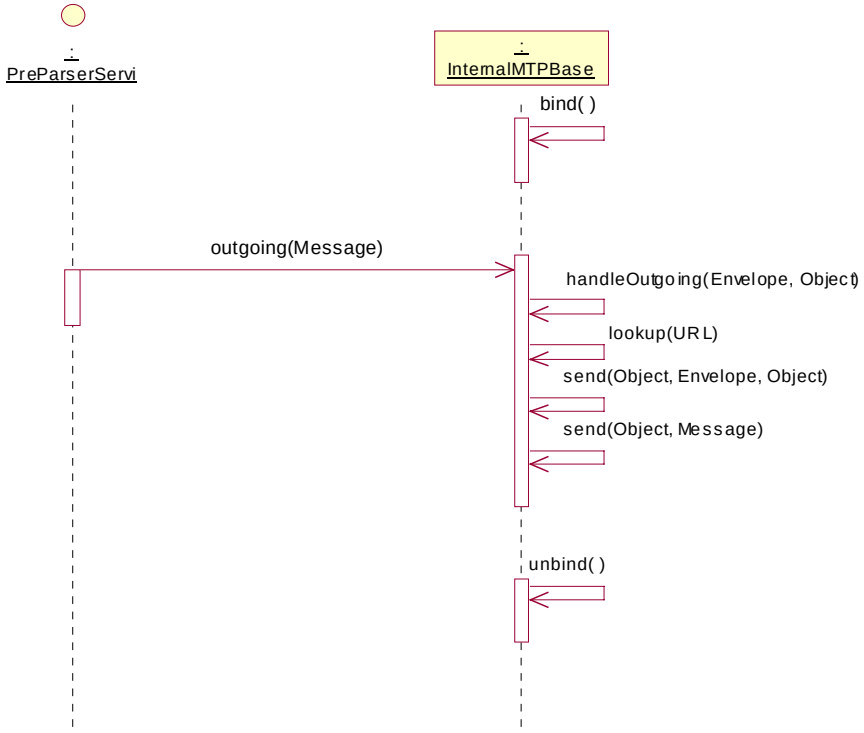

Figure 26 - Interactions When Sending a Message (incl. binding and unbinding of MTP which only normally occurs when an Agent starts/stops)

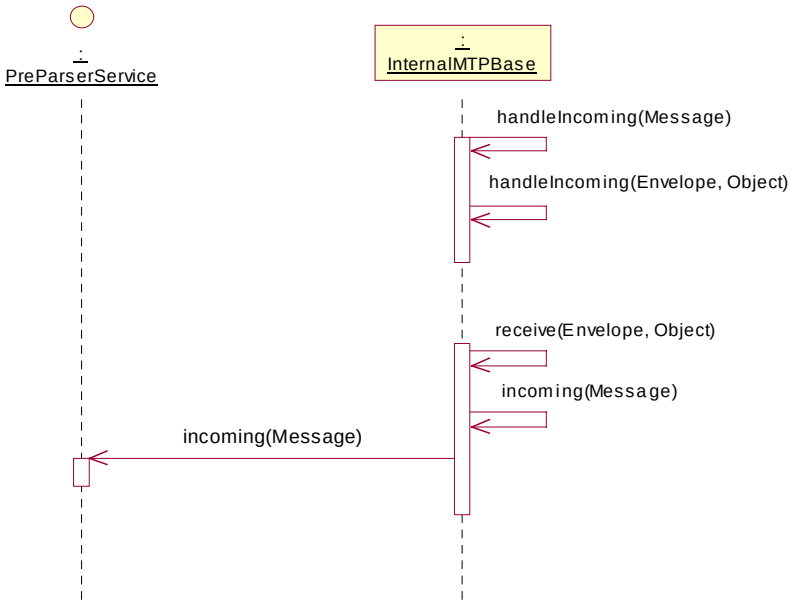

Figure 27 - Interactions When Receiving a Message

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 28 and Figure 29 highlight the interaction involved internally when messages are sent and received within an `ExternalMTPBase` sub-class.

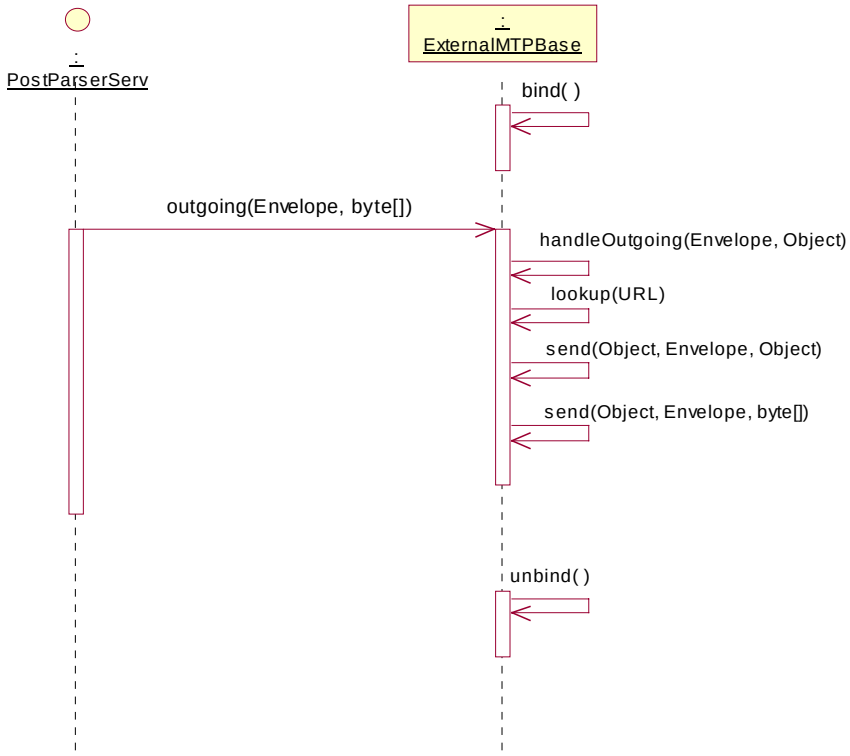

Figure 28 - Interactions When Sending a Message (incl. binding and unbinding of MTP which only normally occurs when an Agent starts/stops)

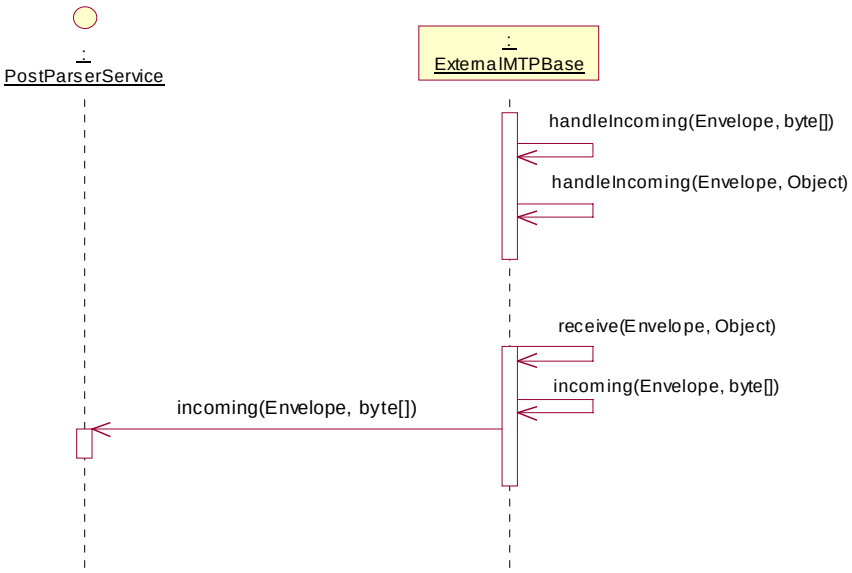

Figure 29 - Interactions When Receiving a Message

Bundled MTP Implementations

FIPA-OS currently comes bundled with the following MTP implementations that are specialisations of the `MTPBase` class.

RMI

The RMI transport is based upon Sun's RMI implementation that is part of the core Java 1.1 and Java 2 Standard Edition API. Due to the fact that this transport relies upon the use of Java Serialisation to encode messages, it is not interoperable with Agents written in languages other than Java. However, this also means that it is much more efficient for communications between Agents written in Java (i.e. currently all FIPA-OS Agents). Thus, this transport is an Internal MTP.

IIOP

The IIOP transport is based upon Sun's CORBA implementation that is part of the Java 2 Standard Edition API (although it is not a core Java component). It is compliant with the FIPA IIOP MTP specification [5], and hence is potentially interoperable with an Agent written in any language that supports CORBA, and this specification. Hence, this transport is an External MTP.

Database Factory

The Database Factory provides a mechanism for instantiating a particular instance of a class that implements the `Database` interface. Figure 30 highlights some of the relationships between classes related to the Database Factory.

Based upon some criteria passed as arguments to the `DatabaseFactory` class (i.e. the class-name at present), the `DatabaseFactory` instantiates an appropriate `Database` implementation and returns the reference. The aim is that different types of database can be used transparently by a particular Agent. The `Database` interface defines the required semantics of any object implementing that interface.

The only requirement of objects that can be placed into a `Database` implementation is that they implement the `DatabaseObject` interface, which implies the objects are serialisable.

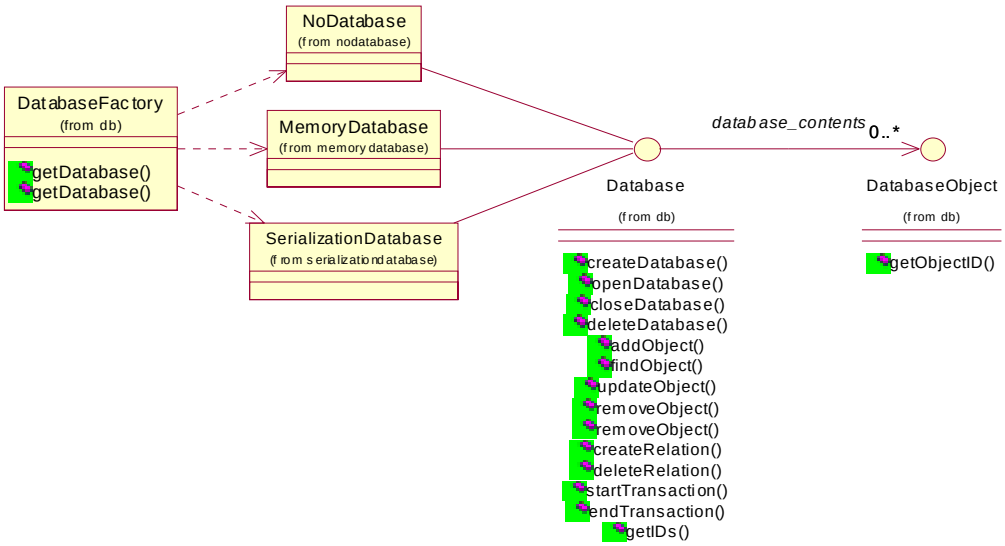

Figure 30 - Database Factory Class Relationships

NoDatabase
This is a dummy `Database` implementation, providing no storage mechanisms whatsoever for `DatabaseObject`'s – all methods either return null or take no action.

MemoryDatabase
This `Database` implementation is a transient database which loses its contents when the JVM the Agent it is contained within closes.

SerialisationDatabase
This `Database` implementation provides a simple persistent database mechanism, which uses Java serialisation to write individual `DatabaseObject`'s to files within the host computers file-system. WARNING: No guarantees about the robustness or scalability of this implementation have been made.

Future Work
This section highlights areas of future work that might be considered.

Improved Profiles
Due to the increasing range of machines FIPA-OS Agents are being deployed onto, it is envisaged that through improved profile support for core-components and optional components Agents can simply be adapted through profile modification.

Planner Scheduler
In order for FIPAOS Agents to become more proactive and goal based, a new and improved Planner Scheduler may be introduced to enable this. The aim is that FIPA-OS Agent development can be simplified through use of profiles to determine the goals of an Agent, and the Agent will draw upon Task instances it knows about to achieve those goals (though cooperation with other Agents or individually).

Agent Component Monitoring
The ability for an Agents state to be monitored might be considered to enable the Planner Scheduler to become aware of what resources an Agent has available, and how to best re-arrange work to meet commitments.

This could involve monitoring of the number of pending messages/conversation updates/tasks to be dealt with, and re-prioritising work based upon the current situation the Agent finds itself in.

Chapter 3
Optional Components

Parser Factory

The aim of the Parser Factory is to provide automated transparent parsing and deparsing mechanisms for multiple languages to a language-neutral semantic-maintaining representation. This component is unfinished, so FIPA-OS does not currently support this functionality at present.

A subset of the required classes for this functionality have been produced however. FIPA-OS includes a number of parsers that conform to a standard interface and use generic objects to represent the semantic information within a message (although they still require language/parser specific information).

Figure 31 highlights some of the key classes within the Parser Factory (note the deliberate absence of a `ParserFactory` class).

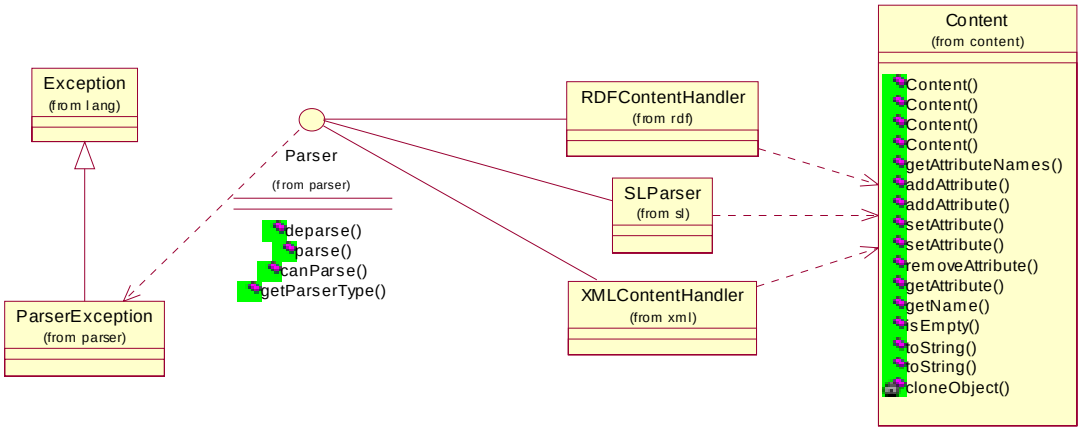

Figure 31 - Parser Factory Class Relationships

Content Object

The Content object provides an abstract one to many attribute-value mapping, enabling the semantics of a document (e.g. a message, or its content) to be represented through an "entity-attribute(s)-value(s)" relationship, which can be of a tree like form due to the ability to nest Content objects within one another.

Parser Interface

The Parser interface provides a number of methods that all parsers are required to implement:

- `public Content parse( String document )`
  Attempts to parse the given "document" into a generalised form (i.e. `Content` objects).

- `public String deparse( Content document )`
  Attempts to de-parse the given `Content` object into a language specific form (determined by the `Parser` implementation).

- `public boolean canParse( String document )`
  Indicates if the `Parser` implementation can deal with the given "document".

- `public String getParserType()`
  Returns the type of the `Parser` implementation (e.g. RDF, XML, SL etc…).

From this interface, it is intended that eventually an Agent could automatically select an appropriate parser automatically to deal with parsing and deparsing of messages.

Data Binding

With the version 1.4.0 release of the platform, FIPA-OS now includes support for a new technology called data binding.

Data binding is the process of converting back and forth between a runtime object and a representation of that object that can be stored persistently or sent as part of a message. It is not tied to any programming language or to any constraint and instance document formats.

In the 1.4.0 release of FIPA-OS, support has been added for converting between Java objects and their representation in XML, the eXtensible Markup Language. This work is focused around the agent profile system. Agent profiles are stored persistently as XML instance documents and are unmarshalled (converted from document form into objects) into runtime profile objects for system configuration.

Data binding starts with a schema document, in this case, an XML schema. The XML schema defines the form of one or more instance documents. The schema undergoes a process known as schema mapping, which statically generates programming language structures that represent the instance documents, in this case, Java Class and Interface definitions.

The objects that instantiate the generated Class definitions are known as data binding compatible objects and can be marshalled and unmarshalled to and from instance documents. This process relies on the Java reflection mechanism to invoke accessor and mutator methods on the data binding objects. The class definitions for objects that will be created through unmarshalling must be in the JVM classpath before the unmarshalling is attempted.

In the example below, a fragment of an XML schema that defines a RemoteAgentPlatformProfile structure is shown along with the Java Interface and Class definitions that are created during the schema mapping process.

```
<!--
****************************************************************
A Remote Agent Platform Profile
****************************************************************
-->

<complexType name="RemoteAgentPlatformProfile"
enhydra:package="fipaos.agent.profile">
      <complexContent>
            <extension base="enhydra:Profile">
                  <sequence>
                        <element name="addressesLocation"
                              type="string" />
                        <element name="HAPName" type="string" />
                  </sequence>
            </extension>
      </complexContent>
</complexType>
```

The generated Interface definition:

```
package fipaos.agent.profile;

public interface RemoteAgentPlatformProfile extends Profile {
    public void setAddressesLocation (String addressesLocation);
    public String getAddressesLocation ();
    public void setHAPName (String HAPName);
```

```
    public String getHAPName ();
}
```

The generated Class definition:

```
package fipaos.agent.profile;

public class RemoteAgentPlatformProfileImpl extends ProfileImpl
                        implements RemoteAgentPlatformProfile {
    private String addressesLocation;
    private String HAPName;

    public RemoteAgentPlatformProfileImpl () {
        super();
    }

    public void setAddressesLocation (String addressesLocation) {
        this.addressesLocation = addressesLocation;
    }

    public String getAddressesLocation () {
        return addressesLocation;
    }

    public void setHAPName (String HAPName) {
        this.HAPName = HAPName;
    }

    public String getHAPName () {
        return HAPName;
    }
}
```

A fragment from an XML instance document containing this object would resemble:

```
<remoteAgentPlatformProfile hAPName="home.fipa-net.org"
addressesLocation="http://www.fipa-net.org/platform.addresses" />
```

The FIPA-OS configuration tool has been updated to make use of data binding techniques to load, manipulate, and save the new XML versions of the FIPA-OS profiles.

Enhydra Project

FIPA-OS uses prototype data binding code released by the Enhydra project (http://www.enhydra.org) under open-source. The data binding code has now become part of a new Enhydra project called Zeus (http://zeus.enhydra.org). The FIPA-OS development team intends to track the Zeus project and make use of the latest developments in open source data binding.

The FIPA-OS development team gratefully acknowledges the Enhydra project for their data binding work.

JessAgent Shell

The JessAgent shell provides an interface to the Java Expert System Shell (JESS) [14] so that an agent can use a knowledge base to reason. In this way it is hoped that the developers can write their own FIPA-OS Intelligent Agents. JessAgent is not a stand-alone agent – i.e. there will never be an agent running called JessAgent. An agent must extend the `JessAgent` class to get access to the JESS functionality. Since `JessAgent` extends the `FIPAOSAgent` it will give to the extending agent all the normal agent functionalities and additional JESS functionality.

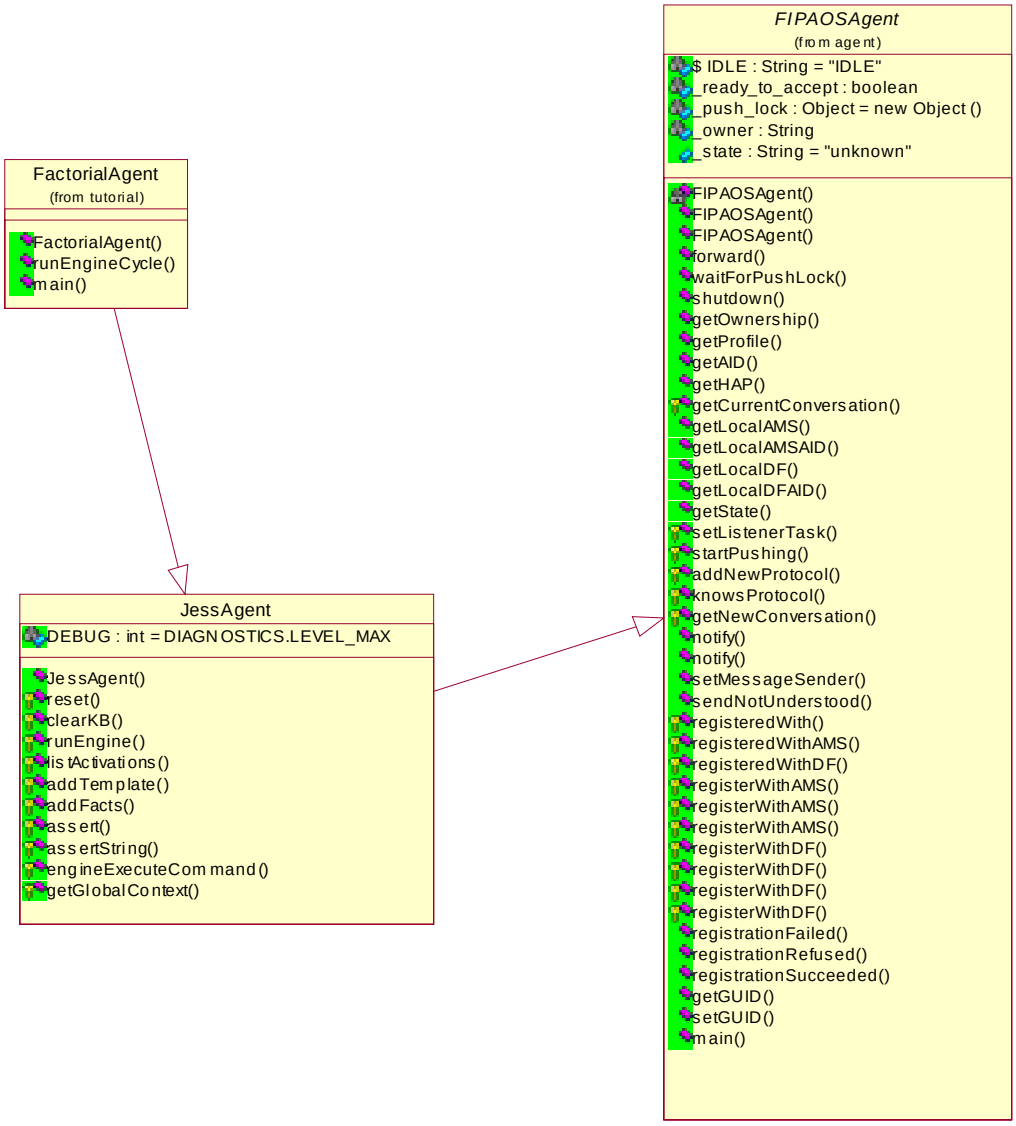

Figure 32 – JessAgent and FactorialAgent Class Relationships

The Figure 32 shows the class relationships between the super class `FIPAOSAgent` and the `JessAgent`, also in the picture the tutorial agent `FactorialAgent` – actual application agent – is shown. More information about the `FactorialAgent` can be found in tutorial document 'Step Five – Extending JessAgent – FactorialAgent'.

JESS a tool for building a type of intelligent software called Expert Systems. An Expert System is a set of rules that can be repeatedly applied to a collection of facts about the world – rules that apply are executed (fired). JESS uses a special algorithm called Rete[3] to match the rules to the facts, which is much faster than simple if-then statements in a loop. Developers wanting to use `JessAgent` should be familiar with JESS language, since the expert system is written using it.

[3] In the Rete algorithm, the inefficiencies of traditional expert system algorithms are alleviated by remembering past test results across iterations of the rule loop. Only new facts are tested against any rule LHSs. Additionally new facts are tested against only the rule LHSs to which they are most likely to be relevant. As a result, the computational complexity per iteration drops.

The `JessAgent` has one instance of the JESS Rete Engine – this means that for each instance of the `JessAgent` there is only one reasoning engine, and therefore all the rules and the facts fed in to the engine will always apply unless the engine is cleared. It is not possible at the moment to have more than one Rete engine in one agent.

JESS rules and other data are typically entered into a separate script file and read it into JESS using the batch command. Jess does offers also commands like 'ppdefrule' and 'save-facts', both of which can be very helpful in interactively building up a system definition and then storing it in a file. See tutorial document 'Step Five – Extending JessAgent – FactorialAgent' or JESS documentation [14] for examples of JESS scripts.

Methods

Constructor of the `JessAgent` sets up the agent functionality, creates a new Rete object, and removes JESS' output router so that we don't get any output printed from JESS. Below is a list of the methods that are provided by the `JessAgent` to the agent extending it.

- `protected void reset()`
  Resets the engine, which means that all the facts and all activations are removed from the engine, and then all the facts found in deffacts are asserted.

- `protected void clearKB()`
  Clears the knowledge base, which means that all the rules, deffacts, defclobals, deftemplates, facts, activations and the like are deleted.

- `protected String runEngine()`
  Runs the JESS knowledge base returning the output obtained from the engine.

- `protected Enumeration listActivations()`
  Returns all the activations currently in the Rete engine.

- `protected synchronized boolean addTemplate(Deftemplate template)`
  Adds a `Deftemplate` object to the Rete engine, returning a boolean indicating whether the operation was successful.

- `protected synchronized boolean addFacts(Deffacts facts)`
  Adds a `Deffacts` object to the Rete engine, returning a boolean indicating whether the operation was successful.

- `protected synchronized boolean assert(Fact fact)`
  Asserts a `Fact` object to the Rete engine, returning a boolean indicating whether the operation was successful.

- `protected synchronized boolean assertString(String fact)`
  Asserts a `Fact` as a string to the Rete engine, returning a boolean indicating whether the operation was successful.

- `protected synchronized Value engineExecuteCommand(String content) throws JessException`
  Executes any JESS command returning a `Value` object containing the output of the execution. If something goes wrong while executing the command, this method throws a `JessException`. Use this method for running most commands to JESS (excluding adding templates and facts).

- `protected synchronized Context getGlobalContext()`
  Returns the Rete engine's global `Context`. Global `Context` is used to determine the type of the value objects returned by engine execution.

Inner class

`JessAgent` has one inner class: `Package` implements the jess.Userpackage interface. This class loads a number of additional packages to the JESS Engine. Presently all packages are loaded, but the class allows for the possibility of including methods and constructors for loading combinations of packages.

FIPA CCL (Choice Constraint Language)

FIPA-CCL is the FIPA approved Content language for communicating Constraint Satisfaction Problems. CCL is designed as a language to be used for agent communication that directly supports the expression of choice and choice problems in the content of agent messages.

The code included in FIPA-OS conforms to version 2.0.1 of the Constraint Choice Language (CCL). This is the version of CCL that has been adopted by FIPA as the standard FIPA compliant content language named FIPA-CCL. This specification may be downloaded from the CCL Website [7].

Constraint Choice Problems and the field of Constraint Satisfaction Problems involve representing problems as combinations of allowed values that may be solved to obtain solutions. Typically these combinations of allowed values will be represented as Constraints saying what values variables may take simultaneously. A solution is found when all variables can be assigned a value without contradicting any of the stated Constraints.

Typical problems that may be modelled and solved using Constraint Satisfaction Techniques are resource allocation problems and tasks involving task scheduling.

Code included

Code is provided in FIPA-OS for supporting a subset of the objects described in the FIPA-CCL specification. Using this code CSP problems can be proposed and solutions returned.

The following Java objects refer to objects in the FIPA-CCL specification that are supported. These are:

```
fipaos.skill.constraint.ccl.constraint.CSPRelation
fipaos.skill.constraint.ccl.object.CSP
fipaos.skill.constraint.ccl.object.CSPSolution
fipaos.skill.constraint.ccl.variable.CSPRange
fipaos.skill.constraint.ccl.variable.CSPValue
fipaos.skill.constraint.ccl.variable.CSPVariable
fipaos.skill.constraint.ccl.variable.CSPVariableAssignment
fipaos.skill.constraint.ccl.variable.IndexPair
```

All the objects in the FIPA-CCL have code to represent them, however only those described above have been fully implemented. These objects have methods that allow access to obtain and set their variables. They also have methods that allow them to be converted to and from Content objects.

The following two classes are used to represent some of the sets of values that may be held in CCL Messages.

```
fipaos.skill.constraint.ccl.variable.List
fipaos.skill.constraint.ccl.variable.Tuple
```

Finally the `CCLMessage` class allows Constraint problems represented using the above classes to be converted to and from text messages.

```
fipaos.skill.constraint.message.CCLMessage
```

Future Work

Parser Factory

There is still much research to be done to enable transparent content language translation.

Bibliography

[1] FIPA Agent Management Specification (XC00023G), August 2000
http://www.fipa.org/specs/fipa00023/XC00023G.doc

[2] FIPA SL Content Language Specification (XC00008F), August 2000
http://www.fipa.org/specs/fipa00008/XC00008F.doc

[3] FIPA ACL Message Structure Specification (XC00061D), August 2000
http://www.fipa.org/specs/fipa00061/XC00061D.doc

[4] FIPA Agent Message Transport Service Specification (XC00067C), July 2000
http://www.fipa.org/specs/fipa00067/XC00067C.doc

[5] FIPA Agent Message Transport Protocol for IIOP Specification (XC00075D), October 2000
http://www.fipa.org/specs/fipa00075/XC00075D.doc

[6] FIPA ACL Message Representation in String Specification (XC00070F), October 2000
http://www.fipa.org/specs/fipa00070/XC00070F.doc

[7] CCL Home Page
http://liawww.epfl.ch/~willmott/CCL/

[8] FIPA-OS Inter-platform Communication Configuration Guide
http://fipa-os.sourceforge.net/docs/Interplatform_Configuration_Guide.pdf

[9] FIPA-OS SourceForge Home Page
http://fipa-os.sourceforge.net/

[10] FIPA-OS SourceForge Development Home Page
http://sourceforge.net/projects/fipa-os/

[11] FIPA Website
http://www.fipa.org/

[12] XML Specifications
http://www.w3.org/XML/

[13] RDF Specifications
http://www.w3.org/RDF/

[14] JESS Home Page
http://herzberg.ca.sandia.gov/jess/