Technical Report

Undergraduate Dissertation: Development of a Message Oriented Interaction Layer for Agent Communication

Craig Paton

Technical Report 2004-11 May 2004
ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Development of a Message Oriented Interaction Layer for Agent Communication

Craig Paton

BSc (Hons) Computer Science

University of Bath

May 2004

Development of a Message Oriented Interaction Layer for Agent Communication

Submitted By Craig Paton

Acknowledgements

Many thanks to my supervisors Julian Padget and Owen Cliffe for their constant help and support throughout the project.

Abstract

Agent infrastructure that provides developmental and operational support for agents is required to fully utilise the abilities of agent technologies. Existing agent platforms tend to require agents to run "inside" them and force the developer to use one particular agent architecture and provide little or no provision for incorporating legacy software into the system. This report details the design and implementation of an agent platform that consists of a simple message oriented interaction layer for structured agent communication and mechanisms for agent discovery based on names and services on offer. A small open application programming interface is provided that does not impose the use of a single programming language or a particular agent paradigm. Such a solution attempts to facilitate the use of agents and legacy software within the same system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Increasingly the type of systems being developed are becoming more open and distributed, with the environment in which they must operate more dynamic. This move away from the traditional approach of focusing on stand alone systems brings with it many new complexities and exciting opportunities. Traditional software engineering approaches have been found wanting when applied to the characteristics of these open heterogenous systems. Agent technologies which can be seen as the next layer of abstraction up from object oriented approaches have proven to be much better suited for managing the complexity of such systems and are one of the most important developments in Computer Science during the 90's. The topic has started to gain momentum and there is currently a large amount of active research in the area, predominantly by universities but there also some global projects like Agentcities (Agentcities 2004) and there have been some successful industrial applications. There is however much work still to be done before Agent technologies become mature and there are numerous key challenges that have to be faced before there are fully embraced by industry. The move to agent technologies, as with any new technology, will therefore not happen overnight (if ever at all) rather it will be a gradual process. During this transition there will be a vast amount of legacy software that will offer services required by these new agent systems and therefore there need to be mechanisms to enable the interaction between agents and the legacy software. It is the responsibility of the agent infrastructure to provide the developmental and operational support for this interaction.

There have been numerous agent platforms developed in the past that have enabled research to be conducted, but they tend to focus on the development of completely new agent systems and do not provide mechanisms to enable the incorporation of legacy software. They also tend to force the agents to be developed using a certain agent paradigm, force a process model on the agent by requiring them to run inside the platform and require the agent developer to learn how to use a large monolithic API to manage the agents behaviour. The focus of this project is to address this problem by developing a new agent platform that simply acts as agent middle-ware allowing agents to communicate with one another with an application programming interface (API) that allows easy access to these services. The API should facilitate both agents and legacy software co-existing in a single system allowing them to communicate with one another.

1.1 Aim

The overall aim of this dissertation is to develop an agent platform that provides the basic functionalities for structured agent communication and an application programming interface (API) that exposes the services offered by the platform.

1.2 Objectives

- The interaction layer developed should provide the facilities to enable agents running on the platform to communicate with one another using structured messages and contents.
- Support services must be made available on the platform to enable agent discovery based on names and services offered.
- An API should be developed to ease the development of agents that are capable of running on the agent platform with access provided for all of the services.

- The API should be small and open. That is to say that it should not impose a process model or particular agent paradigm on the agent developer and should be accessible from a variety of common programming languages.
- The API should enable agent wrappers to be written for legacy software to facilitate its incorporation into the agent systems utilising the platform

1.3 Report Structure

The rest of this dissertation is structured as follows. Section 2 provides a brief discussion of research that was conducted into agent technologies and other topics related to this dissertation and serves to provide background information that is used throughout the rest of the document. Section 3 discusses the techniques used to analyse the requirements for the platform and contains a specification of the functionality required. The initial discussion of the design of the proposed solution to the problem is presented in section 4 and it provides a high level description of the internal design of the agent platform with more detailed implementation information given in section 5. Finally conclusions are drawn from the project work along with highlights of the achievements and suggestions for future work and improvement are given in section 7

Chapter 2

Literature Survey

2.1 What are agents?

Before being able to fully understand the requirements for the development of an agent platform it is necessary to investigate exactly what an agent is. Unfortunately this is not as easy a task as it may sound as there is no common agreement in the agent community about exactly what properties an agent should have. Different researchers in the field have taken different viewpoints on the agent problem and as a result have different focuses in their definitions. Wooldridge and Jennings (Wooldridge & Jennings 1995) said that for a piece of software to be considered to be an agent then it should exhibit the following characteristics:

- Autonomy
- Social Ability
- Reactivity
- Pro-activeness

In contrast Stoham (Stoham 1997) focuses his definition of an agent around the formalism of an agents mental state

> [an agent] is an entity whose state is viewed as consisting of mental components such as beliefs, capabilities choices and commitments.

In contrast to both of the above definitions Genesereth and Ketchpel (Genesereth & Ketchpel 1994) take their definition of what an agent is from the field of agent communication.

> A software entity is only an agent if it communicates correctly in an agent communication language.

Even though these short definitions appear to be in conflict with one another there is some common ground - they have just emphasised different aspects. For example Stoham does not completely dismiss the importance of agent communication and neither would the proponents of agent communication dismiss the need for the agent to have and maintain an internal representation of mental state. There are common underlying themes in these definitions. An article written by Gressener and Franklin (Graesser & Franklin 1996) attempted to survey the agent community by examining the different views of what an agent is and collate the results to form their own definition:

> An autonomous agent is a system situated within and part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.

Based on these definitions, and in the context of this dissertation the author believes that an agent has the following properties:

- An agent will have some mechanism for representing its internal state
- An agent must be able to successfully communicate with other agents and possibly humans, using some structured agent communication language
- An agent must be autonomous and capable of flexible action within its environment

2.2 Foundation for Intelligent Physical Agents

Early Agent systems were developed on an ad-hoc basis to solve a specific problem within an organisation or in order to conduct research in a specific area. While these solutions were useful for their intended goal they used proprietary protocols and mechanisms for interaction and communication that were created and specified by the system developer. The use of such protocols and languages made it very difficult if not impossible for systems developed by different organizations at different times to inter-operate with one another and research work carried out by multiple institutions could not be pulled together and built on. As is common with any new technology as take-up and recognition grew and the technologies involved matured there was increasing need for standards to promote compatibility. Without such compatibility the advancement of agent technologies would be impeded as time and effort would be spent solving unnecessary problems. This is especially the case for multi-agent systems that operate in the open heterogeneous environments for which agency was originally developed. There was obviously a clear need for a body that would produce recommendations and standards that should be followed by agent and infrastructure developers alike. Such standards should be used to promote inter-operability without hindering research. The Foundation for Intelligent Physical Agents (FIPA) (FIPA 2004*a*) is an organisation set up for this exact purpose as shown by its official mission statement:

> The promotion of technologies and interoperability specifications that facilitate the end-to-end interworking of intelligent agent systems in modern commercial and industrial settings.

FIPA is a not-for-profit organisation founded in 1996 whose membership consists of both academic and commercial institutions who provide guidance and input into the standards that are released as well as informing them of what the agent community wants. Since its conception it has developed many specifications and recommendations all of which can be found online in the FIPA repository (FIPA 1996).

At the heart of FIPA's model for agent systems is the need for agent communication where agents can pass semantically meaningful messages to one another in order to accomplish the tasks required for the application. They provide specifications for the interfaces between agents that can be used for this communication without concerning themselves with the internals of the agent. The idea is that if these interfaces are significantly well defined then two agent systems developed using different technologies by different people will able to successfully communicate with one another. In this respect the FIPA standards remain as independent as possible of the underlying technologies that are used to implement the agents and only attempt to influence the externally facing communication mechanisms so that information can be interpreted in a standard way so that the meaning of the message is kept.

There are a number of specifications developed by FIPA that had an important impact on this dissertation. Firstly and arguably the most important specification produced by FIPA of its own agent communication language called FIPA-ACL (FIPA0061 2002). This document states the syntax and semantics for agent communication using this language and its use required by FIPA. More details of the influence of this specification can be found in the next section. The second specification is the Abstract Architecture that comprises of the following three documents; Abstract Architecture (FIPA0001 2000), Agent Management Specification (FIPA0023 2000) and the Message Transport Service Specification (FIPA0067 2000). These documents specify in an abstract way the common requirements and features of an agent platform whilst remaining independent of the actual implementation. These details are obviously relevant to this dissertation and will be used to inform the requirements analysis and specification phase.

2.3 Agent Communication

As discussed in section 2.1 one of the fundamental properties of an agent is its ability to communicate with other agents in a structured way. This is particularly important in agency as demonstrated by the popular saying in the agent community that "there is no such thing as a single agent system". That is to say any system that is of practical use will contain multiple agents that must be able to interact with one another. As such agent communication is a central problem for this dissertation, mirrored by the fact that this is stated as a core objective in section 1.2.

Inter-agent communication is very different from other forms of interprocess communication (IPC) found in standard distributed systems. This is due to a number of reasons two of which will be discussed here. Firstly it is generally considered that all of processes running in a distributed system were written by the same company with a common overall goal, therefore there is no need for argumentation or negotiation as the processes will have been *designed* to be helpful at all times. Secondly agents have greater autonomy than traditional object oriented systems as they have control over both their behaviour and their data. Therefore agent communication cannot be made through standard remote procedure calls as it is up to the receiving process to decide whether or not to perform some action and not the invoker of the message. It is for this reason that agent communication needs to be semantically richer and more structured than previous attempts at IPC. Instead of just invoking actions in the receiver, agents must be able to convey knowledge and beliefs in the messages in order to argue their position and negotiate. It is these fundamental differences in agent communication that forms the core of the reason as to why agent technologies are different from object oriented approaches.

As a result there has been a vast amount of research into how best to perform this agent communication. The problems can be split into two camps based on the two situations that must be present in order for communication to be successful. Firstly the agents must be able to hear one another - the realm of syntax - and secondly agents must be able to understand one another - the realm of semantics. This project is more concerned with the syntax of agent communication, the semantics of the messages are rooted in the application domain and are therefore the responsibility of the agent developer rather than the infrastructure. For the syntax of messages their have been two major attempts at creating an Agent Communication Language (ACL) namely Knowledge Query and Manipulation Language (KQML) and FIPA-ACL which can be seen as the first and second generation respectively, details of each of these are discussed in the following two sections. Both of these languages have their theory rooted in Speech Act theory that was developed initially by Austin (Austin 1962) and later expanded by Searle (Searle 1969). The core idea is that there are certain things that we say which have the same characteristics as actions in that they attempt to alter the state of the world in some way. A common example given is "I now pronounce you husband and wife". Agent communication is then seen as utterances that are a furtherance of the agents intentions as they will try and influence their environment in some way.

2.3.1 KQML

The Knowledge Query and Manipulation Language (KQML) was developed as part of the DARPA Knowledge Sharing Effort (Patil, Fikes & Patel-Schneider 1993) (Neches, Finin, Fikes, Gruber, Patil, Senator & Swartout 1991) whose aim was to develop protocols for the exchange of represented knowledge between autonomous information systems. The project had two main outcomes the development of a content language the Knowledge Interchange Format (KIF) and an "outer" language KQML. KQML is a message based language for agent communication based on speech act theory using performatives to represent the verbs in the acts. KQML was intended to be used with KIF as the content, but itself remains independent of the syntax for the content language and the ontology used. The language can be viewed as consisting of three layers as follows:

- *Content Layer.* Contains the actual content of the message in the programs own representation possibly as ASCII text or binary information. KQML is independent of this layer other than to know where it begins and ends.
- *Message Layer.* The bulk of a KQML message. Used to attach a chosen performative with the message so that the receiver(s) can determine the meaning that the sending agent attached to the message. Optionally it may also contain the ontology used.
- *Communication Layer.* Stores the lower level aspects of the KQML message such as the sender and intended receiver(s) of the message and its unique identifier. This information is primarily used when transporting the message although the receiving agent may also wish to know these details.

A KQML message is made of up parameter-value pairs and a performative using a LISP like syntax an example of which is shown below:

```
(ask-one
    :sender A
    :receiver B
    :content (PRICE IBM price?)
    :ontology NYSE-TICKS
)
```

Initially KQML excited the community and many agent systems were developed that used one of the many dialects of KQML that were implemented. Although there were some initial successes with such implementations there were numerous problems that were identified. The list presented here is adapted from Wooldridge (Wooldridge 2002)

- The performatives were never tightly constrained - different implementations contained different subsets.
- Transport mechanisms for messages were never fully defined.
- The semantics for KQML messages were never strictly defined, they were only specified in natural language. The meaning for performatives was left up to interpretation by the individual developers of systems. Attempts were made to define strict semantics (Labrou & Finan 1996) but none have been widely adopted.
- It was missing an entire set of performatives, commissives i.e. "I promise to.....".
- The performative set for KQML was overly large.

The result of the problems listed above is that it was often the case that two systems developed that used KQML as their agent communication language would be unable to interact with one another successfully, with the worst case being seeming interaction on the surface but a mis-interpretation of the meaning of the messages. A complete failure of one of the main reasons for having standardised agent communication languages. Having said this KQML was good at what it was designed for and facilitated some research, it is just that something better was needed for interoperability....

2.3.2 FIPA-ACL

Arguably the most important work produced by the FIPA organisation is the release of the agent communication language FIPA-ACL (FIPA0061 2002). Its development was inspired by KQML and indeed the syntax is almost identical as the example message below shows (compare with the example of KQML above):

```
(query-if
    :sender (agent-identifier :name A)
    :receiver (set (agent-identifier :name B))
    :content
        (PRICE IBM price?)
    :ontology (NYSE-TICKS)
)
```

Where we see that the performative still remains as the first item of a message and the other parameters come in key value pairs just as they did for KQML.

Virtually all of the changes that were made from KQML were based on the criticisms listed in the following section. FIPA-ACL addressed the problem of interoperability by providing strict definitions for both its syntax and semantics, the latter of which is expressed in the powerful language Semantic Language (SL). Other areas of interoperability to do with encoding and transportation of the FIPA-ACL messages are also dealt with in other FIPA specifications.

As mentioned above FIPA-ACL was developed after KQML and addressed many of the problems with KQML, as such it is a more mature language and is used more widely. One of the core objectives of this dissertation is the support for structured agent communication. FIPA-ACL provides a well defined syntax and semantics and as such provides a very good candidate language to be supported by the platform. Also since FIPA-ACL is more widely used - and indeed it is used in all of the existing platforms described in section 2.4 this will enable the agents running on this platform to more easily communicate with other platforms.

2.3.3 Content Language

One item of agent communication that has been skipped over up until now is the actual content this after all forms the bulk of the message. The content exists to express information that the sending agent is attempting to tell the receiver. The two agent communication languages discussed in the previous two sections exist mostly to provide the structure for the message and enable the transportation of the content to the correct agent(s) and enable the receiving agent to understand the meaning of the message. This includes specifying what method was used to encode the content, i.e. what language was used and with it is ASCII or byte encoded, what ontology was used and the performative tells the receiving agent the intended meaning of the content of the message.

There are a number of content languages that are available for agent communication, a good investigation into the individual strengths and weaknesses of a variety of candidate content languages was carried out for Agentcities (Agentcities 2004) RTD project (Botelho, Willmott, Zhang & Dale 2002). FIPA (see section 2.2) also maintain a library of content languages that they approve to run on FIPA compliant platforms called the Content Language Library (CLL) (FIPA0007 2001). For a language to be considered by FIPA it has to be able to express at least one of the three following items:

- *Objects* are representations of things in the agents universe.
- *Propositions* express a true or false belief about a given statement.
- *Actions* are things that can be conducted by an agent.

Since the agent communication language is assumed to be FIPA-ACL if a given content language is not able to support one or more of these then it may not be possible for all of the communicate acts to be used. At the time of writing there were 4 languages that made up the library:

- Knowledge Interchange Format (KIF).
- Semantic Langauge (SL).
- Resource Description Format (RDF).
- Constraint Choice Language(CCL).

As mentioned the agent communication languages discussed in the previous sections remain independent of the content language used, so the choice of ACL does not restrict the choice of content language by the agent developer. Also since the choice of content language will be dependent on the application domain for which the agent system is developed it would be beneficial for the platform developed for this project to be independent of one particular content language. All that is required is that the transportation mechanism for agent messages is capable of handling contents in a standard encoding.

2.4 Agent Platforms

As stated in section 1.1 the overall aim of this dissertation is to develop an Agent Platform, it is therefore both necessary and sensible to investigate what solutions have been made in the past in order to ensure that work conducted by someone else is not simply being repeated.

First of all we need to clarify exactly what an agent platform is and its purpose. Agent Platforms are concerned with creating developmental and operational support systems for agents (Michael, McBurney & Preist 2003), such platforms are also often known as agent middleware because they generally form the middle of a three layer conceptual model. Lying in between the operating system running on the host and the application layer that is the agents. It exists to provide a common programming interface for developers to ease the development of agent based systems and to enable platform interoperability so that they can develop distributed systems more easily. FIPA (see section 2.2) have developed their own description of the services and functionality that an agent platform should provide in the Abstract Architecture. These requirements are specified in an abstract way as to not restrict how the platform itself is actually instantiated. The main aim is to produce a template that platform creators can use so as to create platforms that are able to interoperate with one another, thus enabling distributed multi-agent systems to be developed.

There have been a large number of agent platforms and toolkits created to date, some of which have been developed for a commercial setting and others for research purposes. Some of these platforms are briefly described below, all of which to some degree have attempted to be compliant to FIPAs abstract architecture described in the previous paragraph.

2.4.1 JADE

The Java Agent Development Framework (JADE) (JADE 2004) is a framework that aims to simplify the development of multi-agent systems (MAS) that is completely developed in Java. The developers of the platform claim that it is FIPA 2000 compliant - although proof of compliancy to a standard is a delicate issue that has not been solved yet.

JADE acts as agent middle ware by providing both an agent platform and a framework for the development of your agents. The goal is to simplify the development of agents will still maintaining the ideal of agent interoperability. It is a scalable platform that can run on multiple computers at once, which due to the fact that it is written in Java don't have to be running the same operating system. It makes use of the advanced features in Java, such as exceptions, RMI and multi threading to make the internal communications mechanisms efficient. The platform also provides some useful graphical tools that you can use to monitor the platform and the actions of your agents which can prove invaluable during the development phase of a system.

JADE's functionality can also be extended through the use of third party applications for example the Java Expert System Shell (JESS) that supports the development of expert systems through rule based code and Lightweight Extensible Agent Platform (LEAP) (LEAP 2004) that is aimed at porting the Jade platform onto small mobile devices.

JADE's website claims that the platform and framework are independent of the agent internals and the application domain which is achieved by only providing core services such as message transport, parsing and encoding and agent life cycle management. But in order to use the platform you still have to "buy" into JADE's view of what an agent is, for example you have to use the behavior abstractions that the framework provides for you and you have to use the process model defined by JADE. In this respect the developer is restricted into developing agents based on JADE's implementation.

Also since JADE is written entirely in Java the developer is forced to write their agents in Java - which for the most part isn't a problem, but it does prohibit the use of legacy code not written in Java, which could increase the development time of a solution considerably.

2.4.2 AAP

The April Agent Platform (AAP) (AAP 2004) was developed by Fujitsu Laboratories and claims to be FIPA 2000 compliant. It is a lightweight agent platform that has been written in the proprietary language the Agent Process Interaction Language (APRIL) and it provides all of the core functionality of the FIPA abstract architecture. It uses the Inter Agent Communications Model (ICM) to provide all of the functionality for the transportation of messages to and from agents. The ICM is very powerful and supports multiple protocols for the transportation of messages above and beyond those required to be FIPA compliant and multiple encodings for the messages themselves. The platform provides two large libraries of code - a server library and a client library - that enable agents to use the facilities provided by the platform. This forms the basis of the main criticism of AAP since the API is written in APRIL it results in a steep learning curve for the platform and the exclusion of legacy software.

2.4.3 Zeus

Zeus is an open source agent platform developed in Java that aims to simplify the development of agents by abstracting into a toolkit. Collis and Lee (Collis & Lee 1997) state that the development of Zeus is based on the idea that developers should spend less time worrying about the intricacies of new technology and more time implementing solutions to their problems. To this end ZEUS contains a tool kit that has a large library of agent-level components that facilitate the development of distributed agent systems. The toolkit provides sophisticated support for planning and scheduling of agents actions and a graphical interface that enables agents to be designed by dragging and dropping of existing components to build up complex behaviors. The initial set of components can also be extended by developers if they so wish. Such an interface provides a method to create agents very rapidly, however the drawback with Zeus is that as discussed in section 2.1 there is no universally accepted view of what an agent is and therefore in order to use the platform you have to use ZEUS's abstraction of what an agent is. Similarly to both of the platforms described above it is also very hard to get legacy software to run on the platform.

2.4.4 Nuin

Nuin (Nuin 2004) was developed as a byproduct of a PhD thesis by Ian Dickinson at Liverpool University. It was born out of the observation from Ian Dickinson's research that there was a lack of agent platforms that are sufficiently well engineered to allow agent programmers to develop agents with abstractions that correspond to widely accepted agent theories. A paper written by Ian Dickinson and Mike Wooldridge (Dickinson & Wooldridge 2002) describes NUIN as a flexible agent architecture for the development of agents in semantic web applications. The platform runs as a layer on top of JADE (see section 2.4.1) and enables the development of belief-desire-intention (BDI) agents through the use of a language called AgentSpeak(L) (Rao & Georgeff 1996) that aims to bridge the gap between the theory and the problems of implementation of such agents. The platform also makes use of another toolkit called JENA for the handling of semantic web data.

NUIN is a good platform for what it is designed for - the creation of agents for the semantic web - but it has limited uses in other fields. For example it is very strongly biased toward one methodology of developing agents, namely BDI agents, although this fits the semantic web problem well there are other application areas where such a model may not be appropriate.

2.4.5 Problems

All of the platforms described above promote the use of one particular agent architecture, of which there are three camps:

- *Reactive* agents operate in a purely reactive way similar to stimulus-response cycles such as the subsumption architecture developed by Brookes.
- *Deliberative* agents reason about their environment and actions, such as the belief-desire-intention (BDI) model.
- *Hybrid* agents combine in layers combinations of the other two agent architectures attempting to benefit from the best of both worlds.

It could be argued that only supporting one of these types or architectures, and therefore forcing the agent developers to use one such architecture is too restrictive as they all have their own associated strengths and weaknesses for particular problems. Another problem that could be highlighted with the above platforms is that they often impose other restrictions on the developer such as a process or behaviour model, and often require the developers to learn to use a large monolithic API to manage their agents functionality. As such it would be interesting to develop an agent platform that simply provides the mechanisms that allow two agents to communicate with one another without imposing other restrictions on the agents.

2.5 Legacy Software

The majority of software in use today within industry is legacy software. Such software is often large and performs the majority of the critical applications for that business, and such systems tend to be information based for example database systems. In order to make these important services available to new distributed heterogenous systems that are being developed successfully with agent technologies they need to be updated. But herein the problem lies. One option could be to update the old software and re-engineer it with newer agent technologies in mind so that it can be incorporated directly into the new agent system. However legacy systems tend to be large complex beasts and the amount of resources required to perform this re-factoring would make the process unviable. This leads to a second option whereby the existing software is updated so that it services can be made available on the new system. The problem is that over time and as changes are made the structure of the legacy systems becomes corrupt making them harder to maintain and changes in the future often lead to the introduction of new errors. A third and more favorable option would be to develop and agent wrapper for the legacy software that would provide a bridge between the old software and the new agent systems, and would expose the softwares services to other agents. This process of making a services available to agents is often called agentification. It would be hard to see how to perform such a process with the platforms described in section 2.4 as they impose process models and agent architecture on the agents for the platform. The situation where legacy and agent systems co-exist will be around for some time as the transition to agent technologies will not happen overnight. Such a change over period requires mission critical platforms to be developed that enable agent wrappers to be written, the absence of such platforms at the moment provides a hurdle to the more widespread take up of agent technologies.

2.6 Summary

In this section we have noted that support for structured communication using a well specified agent communication language is a key component of what an agent is and that there are two main options KQML and FIPA-ACL. Other functions and services that the platform developed may provide have been specified by a standards organisation called FIPA. The need for this standards organisation was discussed along with why it is important to make use of these specifications. Some existing solutions for agent platforms have been briefly described along with the restrictions that they impose on the agents that developers can create. These restrictions prevent the incorporation of legacy software into these platforms that is an important consideration for many applications.

Chapter 3

Requirements

This section documents the requirements analysis process that was used in developing the agent platform for this dissertation resulting in the requirements specification given. Certain features that were considered but believed to be beyond the scope of this project are also briefly discussed.

3.1 Requirements Analysis

The core focus of the agent platform that was developed for this dissertation is the support for structured agent communication through the creation of a simple message oriented interaction layer. The process used to analyse and determine the requirements for this layer was two fold. Firstly investigation and evaluation of the functionality provided by existing communication layers resident in previously developed agent platforms (see section 2.4 for some examples). This enabled this project to build on previous without simply repeating the same steps from the ground up. Evaluations of the functionality of these layers also highlighted problems and deficiencies allowing the interaction layer developed to be differentiated from others. Secondly the use of the Abstract Architecture (AA) specification (FIPA0001 2000) (see section 2.2) which aims to identify and standardise common abstractions required for agent platforms to promote interoperability. Whilst the platform developed for this project will not contain all of the facilities described by the AA it was seen as best practice still to follow these standards for the components that were implemented. After all the standards have evolved over time and are maintained by experienced researchers in the field of agency, attempting to redo this work would have been undoubtedly resulted in a poorer solution. Following this path ensured that the problems of incompatibility in the past would not be repeated, and in the future the platform could form the basis of a fully FIPA compliant solution. The AA describes functionality for more than just the interaction layer but also the responsibilities of support services provided by the platform that are also needed for this project. For the same reasons as above the AA formed the majority of the input for analysis of the requirements for these components.

The API exists to expose the functionality of the services that the platform offers in a usable way for agent developers. To develop the requirements for this aspect of the project more traditional methods of requirements analysis were used. Potential end users of the system were questioned to determine what functionality they expected and thought would be useful to have and use cases were drawn up.

3.2 Requirements Specification

3.2.1 Overview

As stated in the Introduction the overall aim of the dissertation is to develop an agent platform whose main component is a simple message oriented layer capable of transporting structured messages and their contents. By using this platform programs will be able to interact as agents. This section identifies the key and challenging requirements that need to be implemented in the solution in order to meet the goals and objectives of the overall project. The problem can be seen as consisting of 3 logically different but related components, this fact is mirrored in the structure of the remainder of this section. First off is the description of the functionality of the interaction layer itself, second is the support services that are required to enable agents to make full use of this interaction layer and finally is the API that helps developers code agents that can use the abilities of the previous two components.

3.2.2 Interaction Layer

The interaction layer exists to enable agents to communicate with one another. As discussed in the literature review (see section 2.3) agent communication has to be semantically richer and more structured than traditional inter process communication. This structure comes from the use of a well defined agent communication language. It would have been possible to develop a new structured message format that this platform could use, since it is envisaged that the main application will be to communicate with legacy software which would require relatively simple messages. However section 2.3 has already examined two existing agent communication languages that have been standardised, using one of these languages promotes inter-operability and allowed other aspects of the problem to be concentrated on. The two choices are FIPA-ACL and KQML, the former of which is mandated by the FIPA specifications combine this with the argument presented in section 2.3 for FIPA-ACL's advantages it was decided that it would be a requirement of the interaction layer to support FIPA-ACL as the structured message medium.

FIPA-ACL does not mandate the use of a particular language to express the content a message. Since there are many possibilities and the most suitable choice depends on the application domain for which the agent system is being developed, which is out of the scope of this project, the interaction layer must remain independent of the content language used within the message.

It is the responsibility of the interaction layer to handle all matters relating to the transportation of the messages expressed in FIPA-ACL to the intended recipient(s). For this to be possible the interaction layer must have a transport bus that performs the physical transfer to which all agents that are running on the platform must be able to connect so that they can send and receive messages. The agent is responsible for building and filling in the fields of the FIPA-ACL as appropriate but delegates transportation to the interaction layer by passing it the completed message. Since it is the interaction layer itself performing the sending agent communication is asynchronous. Whatever transportation mechanism is employed by the interaction layer it must be capable of handling the following three types of message:

- *Unicast* message is sent from one sender to exactly one recipient.
- *Multicast* message is sent from one sender to multiple recipients.
- *Broadcast* message is a special case of a multicast message, instead of being sent to a set list of recipients it should be sent to all agents connected to the transport bus.

In transporting these messages the interaction layer must be able to determine the recipient(s) and deliver it over the transport bus to only those agents. For this to be possible the interaction layer must be able to uniquely identify each agent through an address on the transport bus. Therefore there needs to be some mechanism to ensure that every agent has a mapping from its name used within the agent system to a transport address, so that each agent is uniquely contactable. The maintenance of this registry of agent and transport address mappings will not be conducted by the interaction layer, rather it is the responsibility of one of the platforms support services called the Agent Management Service (AMS) that is described in the following section.

The transport mechanism must remain independent of the message sent, yet still be able to determine the contents of the sender and receiver fields so that it can fulfill its delivery responsibilities. In this respect the interaction layer is acting as the postal service for agent messages. In needs to be able to determine the final transport address on the transport bus (analogous to street addresses), and in delivering the message it does not and cannot care about the actual content (as a postman does by not reading the contents of the envelope). It is the concern of the receiving agent and not the interaction layer to attempt to make sense of the message. Therefore there must be some extra information created for each FIPA-ACL message that is solely used by the interaction layer for transportation.

3.2.3 Support Services

The interaction layer on its own is not a very useful tool, there needs to be some extra support services in order to make full use of its functionality for agent communication. The interaction layer supports the transportation of agent messages but it provides no means to discover what agents are running on the platform and hence who they can possibly send messages to. Therefore there is not any way

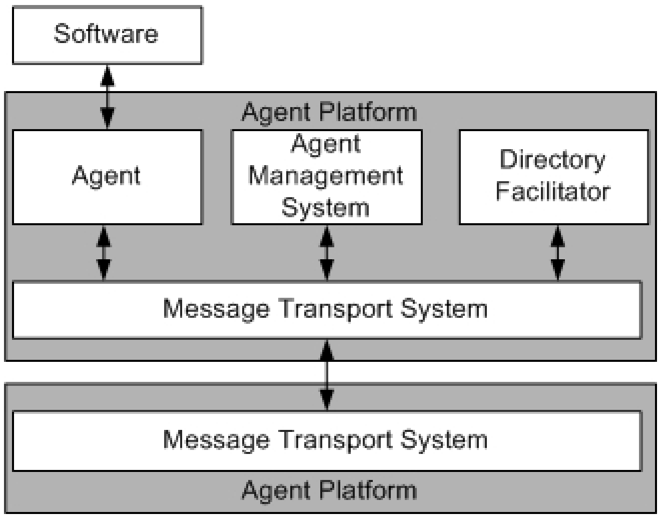


Figure 3.1: FIPA Agent Management Reference Model

for two agents to communicate with one another without explicit prior knowledge of their existence by the developers, and with the guarantee that they are running on the platform. Such a set up represents a very rigid environment that is not very useful and is the complete opposite of the dynamic open environments for which agent technologies were developed. Mechanisms are required to enable dynamic discovery of agents and services that are available at runtime so that agents can come and go on the platform and that this can be detectable. The mechanisms that supply this ability (among others) will be called support services within this dissertation, their primary purpose will be for this dynamic discovery of agents.

The exact services required and their specific requirements stated below have been largely influenced by the FIPA agent management reference model that is shown in figure 3.1. This model is found in the Agent Management Service specification (FIPA0023 2000) that forms part of the abstract architecture. In this diagram the interaction layer discussed in the previous section represents the Message Transport Service. The two support services identified as important for this project are the AMS and the DF and they are both described below.

Agent Management System (AMS)

There should be exactly one AMS running on the platform which is responsible for exerting supervisory control over the running of the platform. That is to say that it is the body that is charged with the responsibility of deciding whether a given agent is correctly registered with the platform and whether or not it has permission to use the services provided. In order to fulfill this role the AMS should be the first point of contact when an agent attempts to connect to the platform to ensure that registration is completed, and then maintains a registry of all the agents that have completed this registration. As mentioned in the discussion of the interaction layer the AMS is also responsible for maintaining a unique mapping of agents to their address on the transport bus so that messages can be delivered to the correct recipient. As such the AMS must control the assignment of agent-names when agents register so that each agent has a unique name and that each has at least one transport address. This mapping must be stored in an accessible database and maintained by the AMS that should be searchable. Agents must be able to query this database to find agents based on their name or transport address to allow for dynamic discovery. This is called the *white-page* service. The public functionality that the AMS must provide to user agents is summarised below:

- *Registration.* Every agent will be required to register itself with the AMS in order to make use of the services provided by the platform. The registration process is used to populate the table later used for searching, and to check to ensure that all agents on the platform are uniquely identifiable. The name that the agent wishes to use must be unique, and the agent must supply at least one transport address that is on the transport bus during registration.
- *Modify.* Once an agent has registered itself with the platform the AMS must enable the agent

to modify the agent entry in the AMS registry. Only the owner of the registry entry should be allowed to change it, also the name of the agent must remain constant throughout its lifetime.

- *Deregister.* Once an agent no longer wishes to use the services provided by the platform it should de-register from the platform so that its entry is moved from the registry for any subsequent searches by other agents. The agent will no longer receive any messages after de-registering from the AMS.

- *Search.* The AMS must provide a search function to allow for discovery of agents based on their agent descriptions in the AMS registry. The search function should be able to cope with partial matches for the registry and should return all entries that match the search criteria.

All of these operations should not be treated as function calls with possible return values but rather as conversations between the AMS and the user agent, with the AMS always responding even if there are no results and no error. This required the development of conversation and messages protocols for each of the functions. The search function described here will not be only be used by user agents but also the interaction when it comes across a message that is has to deliver for which the transport address has not been specified. The layer will query the AMS for the transport address for the agent of a given name in order to complete the delivery.

One aspect that is not going to be considered from the FIPA AMS specification is that of the agent life cycle. This is because as it is seen as inappropriate for this project and would not be possible to implement an elegant solution to this problem. This is due to the fact that agents will be running outside of the platform under their own thread of control, rather than as a child thread running within the platform. Hence it would be unclear how to implement the semantics of a suspend operation for example as the AMS would not have any control over the agent process and would not be able to force the agent to suspend itself. This process model of having user agents as their own process has been chosen in order to provide as much freedom to the developer of the application agent as possible. In contrast to existing agent platforms where agents are force to run inside the platform and this is seen as overly restrictive. This is especially important for legacy software which will already have a built in process model that would require extensive re-working to be retro fitted to an agent platform.

Directory Facilitator (DF)

The AMS provides a white-page service to agents for dynamic discovery. While this service is crucial to the interaction layer for message transport it has limited applications for user agents as it only allows searches to be based on agent names or transport addresses. A more useful feature would be the ability for agents to advertise the types of services that they offer, and then allow other agents to discover this agent based on that advertisement. The support service that must provide this type of functionality is called the DF, and its service is called the *yellow-page* service. Registration with the DF, in contrast to the AMS, is optional for agents. The DF has the sole responsibility for the maintenance and upkeep of the database of services offered by agents. The public functions that is must provide are:

- *Register.* An agent should be able to register one or more services with the DF, so that other agents can discover the agent based on these services.

- *Deregister.* An agent should be able to remove all service entries relating to itself within the DF.

- *Modify.* After an agent has registered on or more services with the DF it must also be possible for the agent to modify the contents of that entry. Only the agent that made the entry should be able to modify it.

- *Search.* Agents must be able to search the database of service entries held by the DF to discover agents. The search function must be capable of making matches based on partial information, and should always return a result even if no agents match the query.

These abilities as with the AMS should be implemented as conversations between the agent and the DF rather than as straight method calls using the same transport bus as for agent messages. Responses should always be given to a request even if there are no results or there are not any errors. This required the development of conversation and message protocols to enable these conversations to take place.

3.2.4 API and Multi Language Support

We have already discussed how the interaction layer on its own is not a very useful tool, so it has been extended with support services so that agents will be able to discover one another based on their names and or services (white and yellow page services). However in this state it would still be a complicated task for an agent developer to construct an agent that would be capable of using the functionality of the platform to its full capabilities. As it stands they would have to become familiar with the internal workings of the platform and how it has been implemented, this detail is not relevant to the application domain for which they are trying to solve a problem and represents unnecessary complexity. In order to address this problem it will also be required that an application programming interface (API) be developed for the platform that will allow for easy access to the functionality and simplify common operations. As such the developer will be free to concentrate on the logic required to solve the specific problem. The API in the absolute minimum must enable the interaction layer to be used to send structured messages and access should be provided to all of the functions described for the support services.

One problem that has been highlighted with existing platforms and their API's is that they tend to be very large complex beasts, and can often force a programmer to construct their agent within a framework that favors a particular agent paradigm. It is required that the API for this project be a simple small collection of accessible functions that provide the ability to perform common operations, and that a particular programming approach should not be forced on the developer.

The API produced should expose the functionality whilst hiding the underlying complexity of the implementation. The creation and manipulation of the FIPA-ACL messages should be handled by the API, providing simple getter and setter methods for the parameters. The creation of the transport information required by the interaction layer is a by product of the platform and should be automatically created by the API and not be forced on the developers. Native support for the proprietary conversation protocols between the user agents and the support services should also be provided to ease the required interactions. The main focus of the API should be make agent communication through the interaction layer easy whilst still remaining manageable and small.

Another common theme that was highlighted from the evaluation of existing platform's API's is that they tend to be developed in entirely new languages (as is the case with April) or in Java. In this way agent developers are forced to learn new languages or develop agents in Java. The has implications for the integration of agent systems with legacy software which is a key objective of this project. The API should not impose this restriction on the agent developers instead it should aid the integration of legacy software. Therefore the API should be made accessible from a number of common programming languages.

3.2.5 Additional Features

So far we have discussed the development of a interaction layer that is capable of transporting structured messages in between agents, and the use of support services and an API to make this functionality easy to use. This is fine when the layer is running, but so far we have not considered what is required to happen when the platform is started. The following paragraphs briefly describe what is required to happen when the agent platform itself bootstraps and then secondly what happens when an agent attempts to connect to it.

A bootstrap mechanism for the agent platform is required to make it easier to use and it should be provided as a simple one step procedure so that the interaction layer and its associated support services are started with their default settings. Once the bootstrap process is complete the interaction layer should be fully operational and ready to be used by agents. This process must consist of at least the following steps:

1. The internal message transport bus should be started and set into a state so that it is ready to accept connections.
2. The message transport service should be started and acquire its well known service name so that it can be found by user agents. It should be put into a state that it is ready to perform the semantics required to deliver FIPA-ACL messages over the transport bus.

3. The DF support service should be started, it should connect to the transport bus and acquire its well known service name so that it can be found. It must be put into a state that is ready to respond to requests for the public functions previously described.

4. The AMS support service should be started, it should connect to the transport bus and acquire its well known service name so that it can be found. It must be put into a state that is ready to respond to requests for the public functions previously described.

5. The service root for the platform should be built by the AMS that describes the services running on the platform and how they can be contacted on the transport bus. This should include as a minimum the interaction layer, the DF and the AMS.

When an agents starts it needs to perform some initialisation before it can use the functionality of the interaction layer. The steps that should be followed during this bootstrap process should be managed by the API and should include the following:

1. The agent should obtain the service root of the platform that was created by the AMS when the platform was started.

2. This service root should be used to connect to the appropriate transport bus in order to register the appropriate listeners so that it can receive messages.

3. The agent must register with the AMS running on the platform supplying its address that it was assigned when it connected to the transport bus so that it can obtain its true identifier so that it can be uniquely identified by the interaction layer for messages transport.

4. Optionally the agent can then use the service root to register the services it offers with the DF.

Once these start up functions have been successfully completed the agent platform must provide access to its functionality without bias.

3.2.6 Scoping of Project

This section exists to explicitly discuss some aspects of the agent platform and associated support services that will not be considered either because they are considered unimportant with respect to the overall goal of the dissertation or they are not possible within the constraints placed on the project.

The agent platform described so far consists of a simple interaction layer and two support services, the AMS and DF. This is a small subset of the functionality provided by the platforms described in the Literature Review (see section 2.4). Only the components that are directly related to the objectives of this project from the FIPA specifications were considered during the requirements elicitation. For example their will be no provision for the creation of ontologies and the automatic verification of the content of a message against an ontology. For this project the creation and manipulation of contents and their associated ontologies are considered part of the problem in the application domain and is therefore the responsibility of the agent and not the platform developer.

Security is another aspect that will not be considered while this is seen as an important requirement for commercial applications for business-to-business systems it is once again considered outside the responsibility of the communication layer and should be implemented as another part of a platform.

The API described in section 3.2.4 is required in order to make the process of developing agents for the platform easier. However this API only be concerned with providing functionality for agent communication and access to the support services. It will not provide other aspects of developmental support for example the ability to debug agents systems through the use of a sniffer agent. Although such functionality is considered to be very useful tool it would not be possible to provide a good solution within the time available.

Having discussed some aspects above that were not considered as part of this project it does not mean that they were ignored altogether. All of the items mentioned above are seen as useful tools and therefore kept in mind whilst designing the platform so that if required it could be extended at a later date to include such functionality.

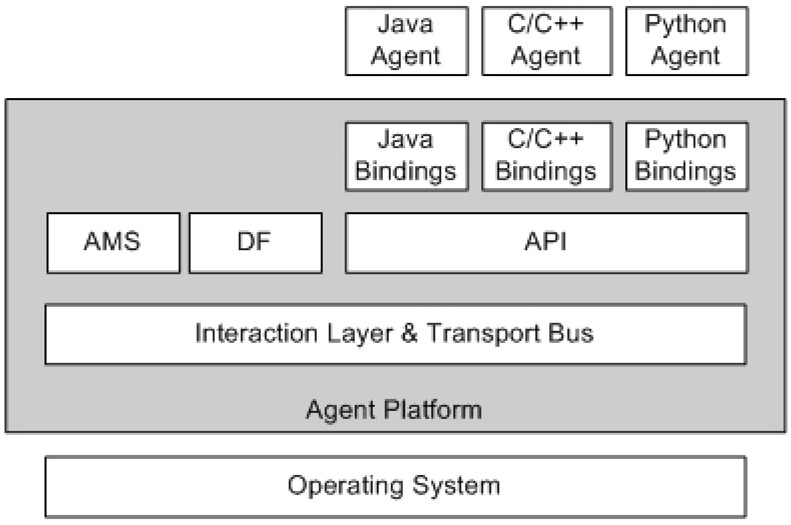


Figure 3.2: Logical layering of platform components

3.3 Summary

This section has set out all of the major requirements for the development of the software for this project. In short it has identified that an interaction layer needs to be developed that is capable of transporting structured messages in FIPA-ACL between agents. The transport layer should be capable of handling the content of such messages whilst remaining independent of the language used. On top of this interaction layer a collection of support services should also be developed to enable agents to discover other agents based on their names or services they offer. A small API is required that will enable agent developers to construct agents that utilise the abilities of the platform, and this API should be accessible from a variety of common programming languages. Mechanisms should also be provided to make the platform easy to use by providing bootstrap mechanisms for both the platform itself and agents. A common theme running through all of these requirements has been the examination of the standards developed by FIPA to inform and shape the specification. Such standards have provided valuable experienced input into aspects of the project that would otherwise have been constructed from the ground up. Figure 3.2 shows in diagrammatic form an overview of the platform architecture showing each of the components described and how they are envisaged to be logically related.

Note that the platform services considered to be part of the platform have a questionable placement within this layering model in figure 3.2. In the interest of performance the services will need to have direct access to the interaction layer for communication with agents, but to enable this communication they will have to make some calls through the API for some of their functionality. As such they can be seen as possibly being a layer below and above the API. They have been shown here as being on the same level as the API although there will be no direct communication from agents, all messages will be channeled by the API through the transport bus.

Some aspects of agent technologies have also been discussed but will not implemented within the platform either because they conflict with the objectives or are beyond the scope of what is attempting to be achieved. The idea behind the requirements, and a theme that will followed through in the design is to create the platform in a modular manner that lends itself to being extended in the future. The idea is to build in the core functionality and then let it be customised as required by the application domain. Also in taking this modular approach it is important that the core of the interaction layer remains independent of content language used thus allowing for more flexibility.

Chapter 4

Design

A high level overview of the solution created for this projet is presented in this section. It serves to document the approach and the key choices made whilst producing the platform. Initially the methodology used to analyse the requirements of the system is discussed, this is followed by an outline of the overall architecture of the solution along with a more detailed look at each of the core components identified in the previous section. The information shown here should be used to aid the reader in following and understanding the details of the implementation which is discussed in the next chapter.

4.1 Methodology

As has been mentioned previously the components that make up the interaction layer and its support services are described in an abstract manner by the FIPA Abstract Architecture. The description provided in these specifications can be interpreted as abstract classes that need to be made concrete in the solution, such an interpretation is suggested within the specifications. These abstract classes then formed the top level input into the design process. Such an approach provided a good method to conceptualise the requirements which was the main reason as to why a loose object oriented approach was taken to designing the system. Even though the solution would not be implemented using an object oriented language (see discussion in implementation section 5) it was still believed to be the best approach. The concepts and techniques used for object oriented design enabled the requirements to be fully analysed in order to provide a deeper understanding of an appropriate design, and the relationships existing between components was explored through class-resource-collaborator (CRC) modeling and use cases. The author of the project is also most familiar with applying such techniques from previous projects as compared to other competing options.

The use of object-oriented design techniques provided methods to analyse the requirements to determine the overall structure of the solution, for more detailed design of the interaction layer and support services a bottom-up approach was taken. The base functionality of each system component was designed first, with more complex functionality built on top utilising this initial work. Each of the main system components identified in the requirements were designed on at a time using this bottom up approach. Such a modular approach allowed the complexity to be managed as it fit naturally to the system to being developed. First off the interaction layer was designed, on top of that the support services and then finally the API. This modular design method also facilitated the independence of the platform from particular technologies (for example a given content language) as each component was designed in relative isolation without assuming dependencies.

4.2 Overall Architecture

From the requirements specification (see section 3) it can be seen that there are the following core components that must be provided by the solution, the message oriented interaction layer, the AMS, the DF and an API. Figure 4.1 provides a pictorial description of the overall architecture of the proposed solution showing these components in layers. Each higher layer is built using the functionality provided by the layers below it, the higher layers depend on the lower ones in order to perform their operations.

From figure 4.1 it can be seen that the base component of the whole architecture is the transport bus. All of the other system components run on top of this bus, it provides the glue that keeps the

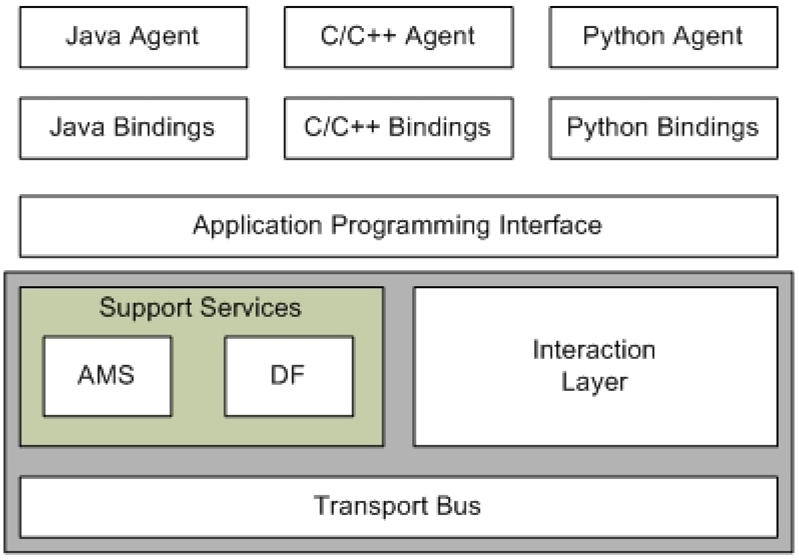


Figure 4.1: Overall architecture of proposed solution.

system together and every component connects to it. It provides the basic capability to transport data from one process to another. The structure of this data and the semantics applied to it is added by the other components to provide the agent technologies. The interaction layer provides the mechanisms and semantics for the delivery of FIPA-ACL messages that is used in agent communication from this basic ability of data transfer. The AMS and DF support services utilise the transport bus for their conversations with agents, they implement the conversation protocols required to access their functionality. Above these three components exists the application programming interface (API) that exposes the functionality in easy to use publicly accessible functions, and is therefore completely dependent on the abilities of the layers below it.

On of the key choices made at the start of the design process was whether to implement the platform services and the interaction layer as agents themselves or as more traditional programs. The agent solution can be seen as purer as all elements running on the transport bus would be agents and the same base code could be used for communication. However it was viewed that agent communication would overly complicate the implementation as the same semantically rich information is not required by the DF and AMS. They are essentially just services and do not enjoy the same autonomy over their behaviour that agents do as they must respond to all requests without bias. Plus simpler conversation protocols could be developed for the interactions with these services that do not use the FIPA request interaction protocol (FIPA0026 2000) that just use the abilities of the transport bus for sending typed data. Hence the AMS, DF and interaction layer were designed as services and not as agents. Problems to do with the discovery of these services by agents in discussed in the implementation section and are resolved during the bootstrap process.

So far in both the Agent Management Reference diagram taken from FIPA, and the overall architecture of the proposed solution the support services in the AMS and DF and the Interaction layer have been shown as separate entities. This has been useful up to now to aid in the understanding and analysis of the requirements and to demonstrate that they perform logically different functions and provide different services, i.e. white page services, yellow page services and message transport. However in order to make the final solution easier to maintain, develop and to promote reuse of code these three components will be coded to form a single process thread in the agent platform rather than as three separate ones. There is no reason why this can't be done, the FIPA standards state that the three components are "*logical compatibility sets*" and therefore pose no restrictions on whether they are provided separately or together. Such a solution simplified the development and had other benefits for message sending that we will consider now. For example consider the situation where AgentA sends a message to AgentB, with AgentA not knowing the full transport bus address for AgentB, just its unique name. The interaction layer will receive this ACL message and determine that it needs to route the message to AgentB. However as yet it will not know the final destination. To determine this it must

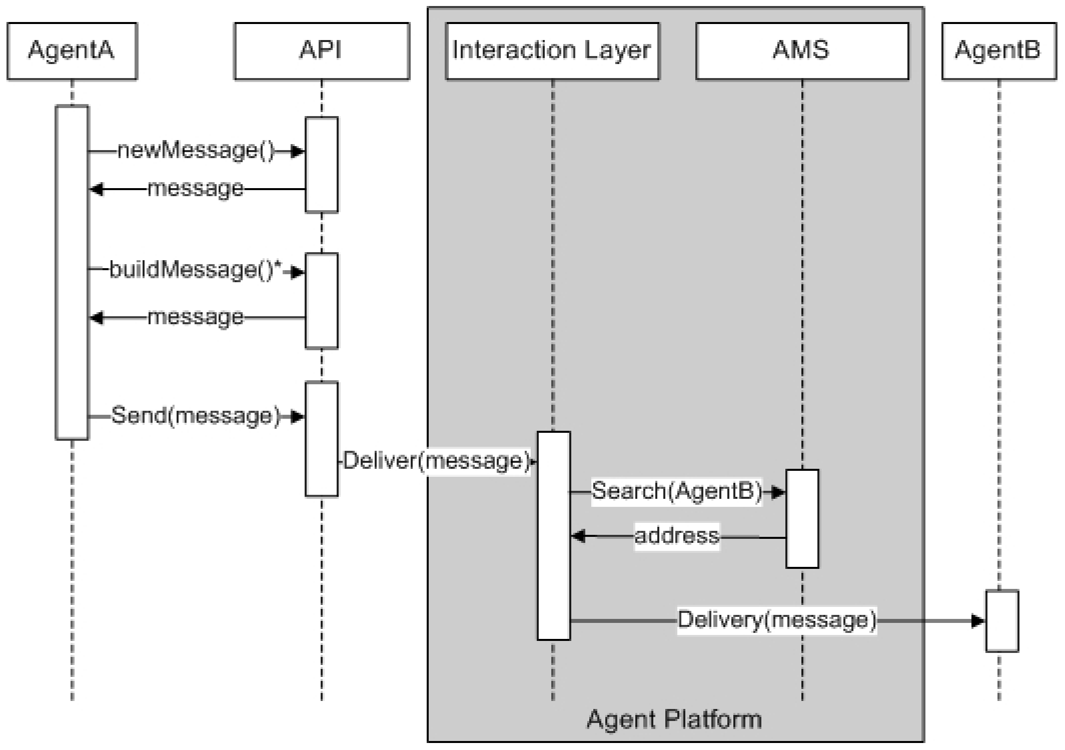


Figure 4.2: Agent Communication Swimlane diagram

perform an AMS search for AgentB's entry. If the AMS and interaction layer are running as separate processes then this can only be resolved through another message exchange between them over the transport bus. On the other hand as is proposed here if they are running in the same process in the same memory space then a more efficient direct method call to the search function can be used to find the address and the message can be routed to AgentB. This direct method call does not violate the AMS as it is a service and not an agent. It is this authors opinion that benefits such as this outweigh the possible bottleneck in the system that such a solution proposes, since it is believed that the amount of messages sent in agent communication will not be that high. Although the solution will not scale well when multiple routing requests need answering at once it is sufficient for this project where the number agents on the platform will not be large.

Another high level architectural design decision made early on was to make user agents lightweight, that is to say that the majority of the processing work to do with interacting with the agent platform will be conducted at the platform end. This is especially the case for agent communication. It would have been possible for every agent to its own implementation of an interaction layer which would connect to the transport bus and perform the semantics for the delivery of agent messages. It was decided however to provide one central interaction layer that would perform all message delivery. This puts less *strain* on the user agents as all they are required to do is build the structured ACL message (with support from the API) and immediately pass this structure to the interaction layer for it to be sent. It can then continue with whatever it wishes whilst the platform performs all of the processing required to actually deliver the message. This is possible due to the asynchronous nature of agent communication. This situation is show in figure 4.2 where AgentA is sending a message to AgentB (the same as discussed in the previous paragraph). As soon as AgentA has built the message through API calls it sends it through the API to the interaction layer in the platform, when this is complete the agent continues and the platform does the rest. The fact that in this design all agent messages pass through a single instance of the interaction layer has other benefits and enable possible extensions to be made in the future. For example such a set up enables the easy creation of a sniffer agent that could analyse all messages sent through the platform allowing for security implementations and powerful debugging tools. Making the agents lightweight also makes the API smaller and easier to manage, the process of binding it to various other languages easier, and uptake by developers simpler.

4.3 Overview of Interaction Layer and Support Services

A high level discussion of the design of the core system components will be presented here. As mentioned in the previous section there is one centralised interaction layer for message transport that will be running within the platform. In order to perform the function of delivery of FIPA-ACL messages there has to be a mechanism for unique agent identities. From the requirements specification this identifier must include an agent-name and the address on which it can be contacted on through the transport bus. Table 4.1 provides a description of the structure designed to hold this information, and an example structure for the AMS is given below that supports communication via HTTP is given below in a LISP like syntax:

```
(agent-identifier
    :name ams@cs.bath.ac.uk
    :address (set
        http://cs.bath.ac.uk/acc/ams
    )
)
```

Item Name	***Type***	***Description***
Agent Name	String	A globally unique identifier for the name of the agent
Transport Addresses	List of string	List of addresses at which the agent can be contacted on. The order of the items in the list is the order in which each of the addresses should be tried.

Table 4.1: Agent Identifier Structure

It is the responsibility of the AMS to maintain a registry of these agent identifiers for every agent registered on the platform. As specified in table 4.1 the agent name must be a *globally* unique string so that the agent is not only uniquely identifiable by this platform but also by others. This afterall is the reason for having a name in the first place as it remains a platform independent way to refer to agents whereas the transport addresses will be depend on the internal transport protocols employed by platforms. A user agent will be given a useful name by its developer, for example PingAgent, and this name is passed to the AMS when the agent registers with the platform. The AMS then appends the name of the platform and checks its database to ensure that no other agent is currently registered with that name. This ensures that the agent is uniquely named on this platform, similar functionality is provided by the AMS on each individual platform, and the platform name should be chosen so that the full agent name becomes globally unique. Any attempt to register an agent with a name that is already in use is denied. You can see this in the example given above with the AMS agent running on a platform called cs.bath.ac.uk giving a full name of *ams@cs.bath.ac.uk*. The value of entries in the transport address list are structured in such a way as to be self-descriptive and completely define how and where the agent can be contacted. It specifies the protocol to be used (http in the case above) and the transport location on that protocol where messages can be sent. This address is the only method that the interaction layer has in order to find the final destination of the message it is attempting to route. The protocol is important as it is possible for the interaction layer to support routing of ACL messages over multiple transport protocols. Although only one protocol is supported by the solution for this project due to time constraints, consideration of future expansion needs was made here for a modular interaction layer that was not solely dependent on one protocol.

The identity three core platform components use well known reserved values that are assigned to them during the bootstrap process. Well known means that the values are known by agents before they connect to the platform, and they are reserved so that no other agent can use them, so the platform services cannot be spoofed. The identifiers for each component is given in the list below:

- *Interaction Layer* becomes mts@hap-name[1] (name comes from the FIPA Abstract Architecture name for the interaction layer - Message Transport Service) and its transport address is MTS.

[1]hap-name is used to denote the fully qualified name of the machine on which the interaction layer is running.

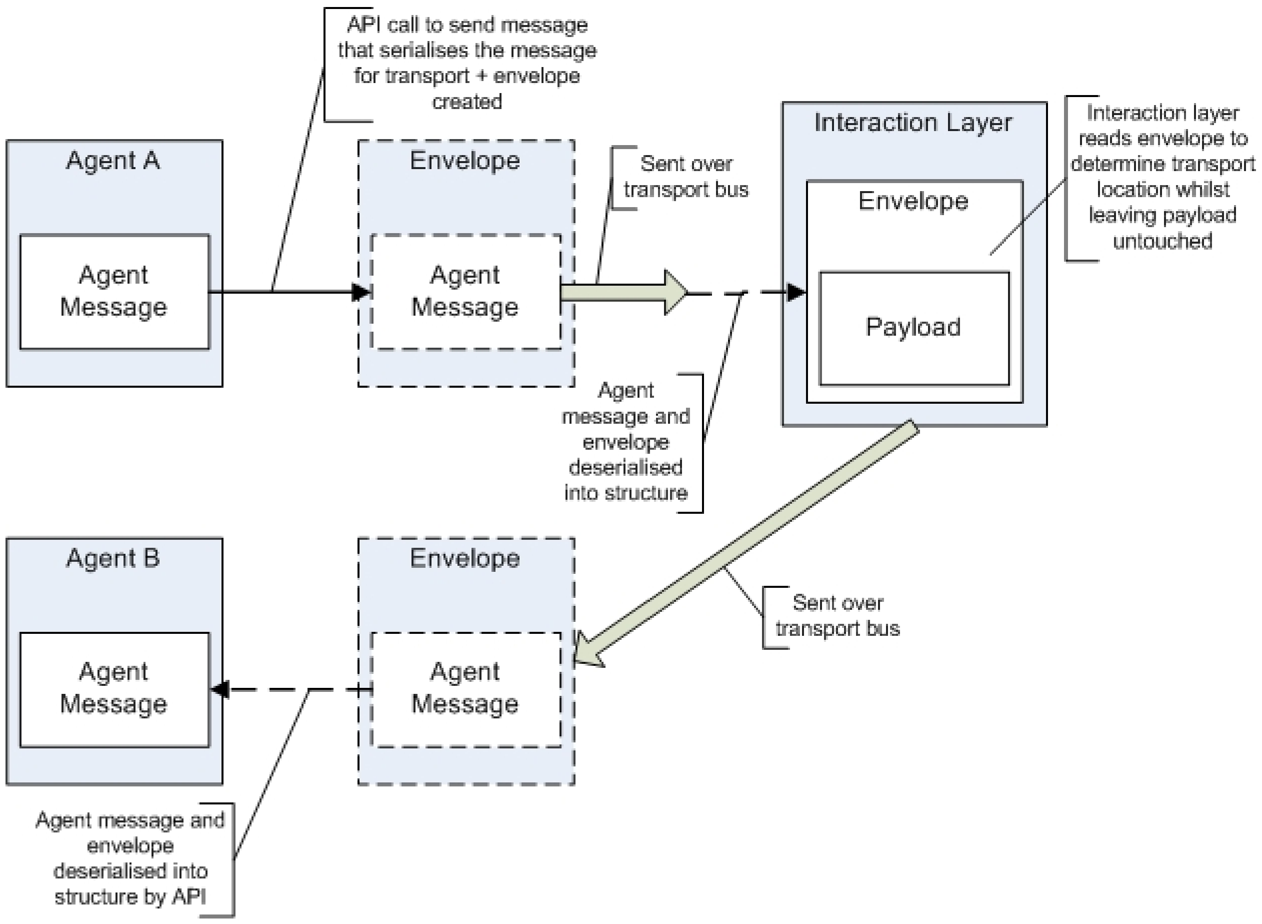

API call to send message that serialises the message for transport + envelope created
Agent A
Agent Message
Envelope
Agent Message
Sent over transport bus
Interaction Layer
Envelope
Payload
Interaction layer reads envelope to determine transport location whilst leaving payload untouched
Agent message and envelope deserialised into structure
Agent B
Agent Message
Envelope
Agent Message
Sent over transport bus
Agent message and envelope deserialised into structure by API

Figure 4.3: Process of sending an agent message

- *Agent Management System* becomes ams@hap-name and its transport address is AMS.
- *Directory Facilitator* becomes df@hap-name and its transport address is DF.

The use of these names is important for the bootstrap process that is discussed later.

We shall now consider how the API and the interaction layer combine to form the capability and semantics for sending agent messages. The interaction layer acts as the central service to which all messages are sent, it then decides how to route the message over the transport bus to all of the intended recipient(s). In order to perform this routing is must have some transport information, most notably who the message is intended for. This information is contained in an envelope that accompanies every ACL message, with the message itself forming the envelopes payload. The interaction layer reads the information from the envelope rather than from the message itself so that it remains independent of the language used to represent the message. Whilst this solution only supports FIPA-ACL this modular approach allows for future expansion. The envelope is created by the API and not the sending agent, which simplifies the process of sending a message from an agent developers point of view as they do not have to concern themselves with envelope creation indeed they do not even need to know that such an envelope exists. When the interaction layer receives an envelope and its associated payload over the transport bus it must read the envelope, without considering the payload at all, and then route the entire structure to the recipient(s) specified by agent identifiers in the envelope. In order to make the implementation of this functionality easier internal data structures will be used by the platform to represent the envelope. Serialisation and de-serialisation mechanisms are provided in order for these data structures to be sent over the transport bus, and these methods are dependent on the transport bus chosen. This overall situation is shown in figure 4.3 with AgentA sending a message to AgentB.

The contents of the envelope for an agent message is shown in table 4.2 (note the AID in the type column is used for the agent identifier structure defined in table 4.1). The to field is used to contain

Item	***Type***	***Description***
to	set of AID	List of all of the intended recipient(s) of the message
from	AID	The agent who sent this message
acl-representation	String	The encoding scheme that has been used to represent the message payload in an agent communication language
intended-receiver	AID	This is the intended receiver of this copy of the message

Table 4.2: Message envelope elements

all of the recipients of the message, it is important to note that this is a list and not just a single element. This is required for support of delivering unicast, multicast and broadcast messages. When the interaction layer receives a message with multiple recipients it should create a copy of the message for each agent and set the intended receiver to the agent for which the copy was made and recursively call the sending procedure with this new set up. If the interaction layer comes across a message with the intended receiver field set then it should ignore the contents of the to field and just route the message to the agent specified as the intended receiver. In delivering a message the interaction layer passes both the envelope and the payload. This is required to enable the API to successfully deserialise the message into program structures so that agents can easily access the information contained within the message.

Both the AMS and the DF keep a database of the information that has been registered with them and it is their responsibility to make sure that this information is correct, up to date and appropriate responses are given to requests. A data structure is required by each of these two services to store the information for which it is responsible and it was decided that a linked list would be the most suitable option. The most common operation that will be performed on these databases are searches through templates and partial matches. Such complex search semantics using structures negated the possibility of using an ordered data structure based on a key field for making the search facility more efficient. Every search would have to be conducted by iterating over all elements in the database, something which is very easy to do with linked lists. Other advantages of the linked list approach is that addition of new elements is very easy and the amount of memory used is efficient as the structure grows dynamically to suit the number of elements stored.

The AMS is used to store the agent-identifiers of the agents that are running on the platform, and it is a requirement that all agents register themselves with the platform before they are able to use the services provided by the interaction layer. When an agent registers with the platform the AMS must update its registry to include this new agent-identifier, and then that agent must also be able to modify this entry at a later date, this modification should only be possible by the agent that initiated the registration. The information that the AMS is required to store in order to provide the white-page services was described earlier in table 4.1. The AMS exerts supervisory control over the interaction layer and its services by only allowing access to registered agents. It also provides the service root to agents when they bootstrap which describes the services offered by the platform so that they can be contacted by the agent over the transport bus, as a minimum an AMS, DF and interaction layer must be provided. The full structure for this platform description is shown in table 4.3

Item Name	***Type***	***Description***
name	String	The name of the platform that the AMS has assigned
ap-services	set of ap-service	The list of all of the services that the platform provide. See table 4.4 for a description of the ap-service

Table 4.3: Contents of the AMS platform description

Item Name	***Type***	***Description***
name	String	The name of the ap service
type	String	The type of the ap service
addresses	set of URL	The addresses on which the service can be contacted on

Table 4.4: Contents of an ap-service description element that forms part of the platform description

The DF exists to provide yellow page services to the user agents running on the platform, as such it's database consists of agent service descriptions that can later be searched. The information that must be stored in this registry can be found in tables 4.5 and 4.6.

Item Name	***Type***	***Description***
name	agent-identifier	The identifier for the agent
services	set of service description	List of the services that this agent provides. Description of the structure of the service description can be found in table 4.6
protocols	set of string	List of protocols supported by the agent
ontologies	set of string	List of ontologies supported by the agent
languages	set of string	List of content languages understood by the agent

Table 4.5: Contents of an agent-service description for the DF

Item Name	***Type***	***Description***
name	string	The name of the service
type	string	The type of the service provided
protocols	set of string	List of protocols supported by the service
ontologies	set of string	List of ontologies supported by the service
languages	set of string	List of content languages understood by the service

Table 4.6: Contents of the service description element of an agent-service description in the DF

The search functions for both the AMS and DF expect to get a template agent-description and df-description respectively for which they search their database for matches. An object within the database matches the template if for every parameter in the template that has been specified there is at least one matching parameter for that object. The templates passed into the search functions must only contain atomic data and not any complex expressions.

4.4 Agent Communication

Highlighted during the requirements elicitation phase was the need for a standard well defined (both semantically and syntactically) agent communication language, and it was decided that FIPA-ACL should be supported. Whilst the interaction layer remains independent of one particular ACL through the use of envelopes, support for only the creation and manipulation of FIPA-ACL messages will be supported by the API. The choice of content language and ontology is left open to the developer so that they can make the best informed choice for their system. The only requirement is that the content language used has to be encodable using the mechanisms supported by the transport bus and interaction layer.

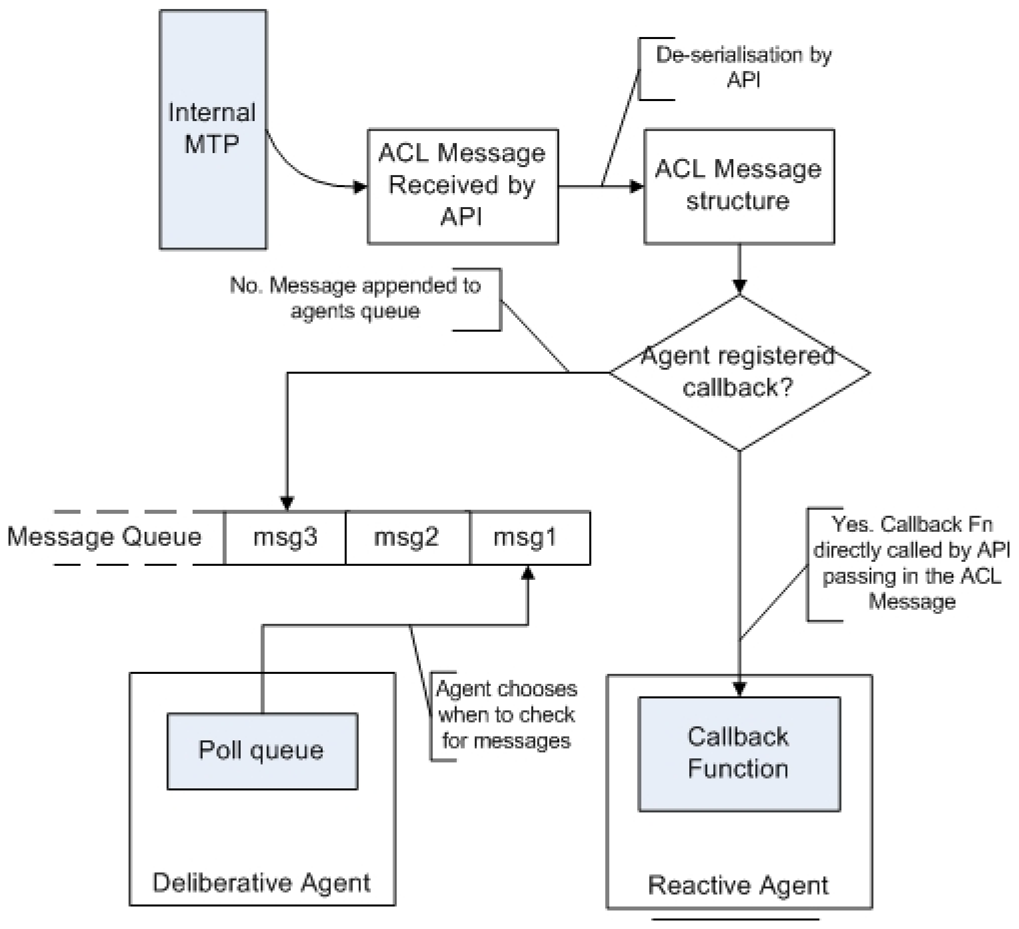

Internal MTP
ACL Message Received by API
De-serialisation by API
ACL Message structure
No. Message appended to agents queue
Agent registered callback?
Message Queue
msg3
msg2
msg1
Yes. Callback Fn directly called by API passing in the ACL Message
Agent chooses when to check for messages
Poll queue
Deliberative Agent
Callback Function
Reactive Agent

Figure 4.4: Two options for notification of receipt of an ACL message

It was decided to provide the developers with two choices of how they wished to be notified when an agent-message has been delivered to the agent by the interaction layer. Two options were offered to provide flexibility for the process model employed by the agent. The first and default is for the API to put all received messages on a queue within the agents memory space and it is then up to the agent to poll this queue regularly in order to get at the messages. The second option is to enable the user agent to register callback functions with the API so that as soon as a message is received the agent is notified via a method call. The API does not favour one method over another, it is up to the agent developer to choose which option best meets the needs of their system. It is envisaged that the polling of the queue would be used by deliberative agents, and the callback by reactive agents. This two option situation is shown pictorially in figure 4.4.

4.5 Application Programming Interface

The major tool used to analyse the requirements for the API, and to determine its required functionality was use cases, a sample of which can be found in appendix A. API's provided by existing platforms were also studied and evaluated to identify other features that were useful and had been overlooked. From this process it was evident that there were three main phases of an agents life cycle for which the API should provide support:

- *Bootstrap*. Support is provided for connecting an agent to the platform. Discovery of the platform, the AMS and obtaining the service root are all handled internally.
- *Alive*. This contains the majority of the API code. Support is supplied for manipulating the platform structures through simple getter and setter functions so that the agent can use the structures required by the implementation without knowing the inner workings. It is mainly concerned with agent communication by providing simple ways to send and receive FIPA-ACL messages. The encoding and decoding of these structures so that they can be sent over the

transport bus is handled internally by the API and is transparent to the developer. Effort was made to ensure that the functions remain independent of a particular language.

- *Exiting.* The API provides functions to allow the agent to gracefully exit and free up all resources on the platform to which it is associated when it no longer wishes to use the services. De-registration from the AMS and DF is performed as is disconnection from the transport bus. Once this process is complete the platform services will no longer be available through the API and the agent will not receive any messages.

The API is small and manageable with simple access to the required features. It was designed to be modular so that new capabilities could be added at a later date.

4.6 Summary

A high level overview of the solution to the project has been presented. Decisions made about the design of this solution that are independent of the implementation have been discussed. The semantics for the routing of, delivery and receipt of ACL messages has been shown. The structures that contain the information held by the AMS and DF have been designed along with how they will actually be stored. The detail of how this solution was implemented is given in the next section.

Chapter 5

Implementation

This section provides details of the implementation procedure and decisions that were made to produce the agent platform based on the design given in the previous chapter. Initially it describes the tools that were used to produce the solution and then it moves onto to highlight areas of the development that were particularly interesting or hard to implement. A common theme of the implementation process is the re-use of existing tools and libraries of code. Such code will have been rigourously tested over time and thus more stable than solutions that could have been developed here. These libraries were used to perform common programming operations and standard data structure management so that the problems associated with this project could be concentrated on.

5.1 High Level Implementation Decisions

Here we will briefly discuss some of the key decisions made early on in the implementation of the software after a short investigation. The points laid out here have consequences for the implementation of the whole platform.

It was decided than an incremental development process would be followed for implementing the platform. The base components of the system were developed first in their entirety, then more complicated functionality that depended on these components was built on top. This mirrors the layered architecture presented in figure 4.1 and the bottom-up approach used to design the platform. Testing was carried out through all stages of this incremental development when every milestone was reached. The order for the implementation of the components was as follows:

1. Transport Bus
2. Interaction Layer
3. AMS
4. DF
5. Bootstrap Process
6. API

Breaking the problem into logical components like this enabled it to be easily digested and understood by being able to concentrate on one individual aspect rather than being overwhelmed by the project as a whole.

5.1.1 Platform

It was decided that the implementation of the solution should be targeted for the Linux platform. The majority of the potential users for the system were using this operating system although there were some Windows users. It is easier to port a Linux based solution to the Windows operating system than vice versa, therefore Linux was seen as the logical choice. The author was unfamiliar with Linux before the project started and it required the set up and configuration of a Redhat 9 (Redhat 2004) box. This choice of Linux as the platform influenced the other tools that were used in implementing the solution which are discussed below.

5.1.2 Language Choice

The solution presented here is developed in the programming language C. The agent platform itself is a low-level architectural piece of software that exists as a layer above the operating system, therefore it required the use of an efficient language that could interface directly with Linux. C is very capable of meeting these requirements. Also the author has experience programming with both C and C++, therefore the skills required to implement the solution could be easily obtained. The use of C does have some potential pitfalls, for example dealing with memory management which has to be done manually rather than being handled by the language itself. To help with this existing libraries of code were used that provided better support than native C. The choice to use C influenced the other tools and libraries that were used for the project described in the sections that follow.

5.1.3 Transport Layer

The platform requires a transport bus to be used to provide the ability for inter process communication (IPC) so that agents can talk to the platform and vice versa. Whatever form the transport bus takes it must be capable of transporting typed data from one process to another. It would have been possible to develop an IPC layer specifically for this project but it would have detracted from the problem at hand and confused matters. It was decided that the best approach would be to make use of an existing IPC layer which would be free to use. Such a layer would have to be capable of running on the Linux platform and enable structured data to be passed from one application to another. After investigation it was decided that the D-Bus (DBus 2004) system was most suitable.

D-Bus is a simple message bus system that enables two processes to communicate with one another, it also provides an API accessible from C that you can use to interact with the system programmatically. The authors of the system say that it was originally developed for communication between the operating system kernel and user applications for notification purposes, however essentially it supports a generic message passing mechanism that allows messages containing typed data to be passed between processes, exactly what is required for this project. At its core it provides mechanisms for the one-to-one transfer of messages in a traditional client server architecture. On top of this two message buses have been built that enable routing of messages between multiple applications which is what is needed here. The message bus logically acts as the central component on a wheel with each process (an agent or the platform itself in the case of this system) acting as a spoke. Messages are sent down an applications spoke, with a recipient specified, to the message bus who ensures that the message is routed out through the spoke associated with the message recipient. This is what is required from the transport bus by the interaction layer for agent communication. The message bus makes its routing decisions based on service names. On top of this ability the interaction layer adds the semantics for agent communication through structured messages, possible address lookups in the AMS and the sending of multicast and broadcast messages. The actions of the message bus for sending a message through the interaction layer are shown in figure 5.1. The message buses routing capabilities simplified the implementation of the interaction layer, as it could delegate the managing of connections and routing decisions to the message bus. The key point here is that agents talk to one another through there globally unique names that form part of an agents identifier structure, and it is the interaction layer combined with the AMS that enable these conversations by mapping the names to the transport bus addresses on the D-Bus message bus. The message bus handles all matters to do with routing the message to the correct agent, but it only transports raw data the structure and meaning is applied to this data by the interaction layer.

It should be noted here that while D-Bus provides the functionality required for the interaction layer in order to meet the objectives of the dissertation the D-Bus project itself is still in the development stage and not yet reached version 1 and as such is under constant change. Version 0.20 was used in this implementation although it is likely that later versions of this tool will not work using the same protocol. Also there were some issues with omissions and errors due to its infancy in the documentation provided by the developers for the API which provided unnecessary complications.

After the initial development stage of the interaction layer that was built on top of the transport bus the author came across another IPC tool that could have been used instead of the D-Bus project, but since time was already invested in the development of a D-Bus based solution it was felt that it would not have been possible to have converted to it within the time constraints. But it is noted here as something that could be useful for future developments in the project. The tool is called ATK-Bus

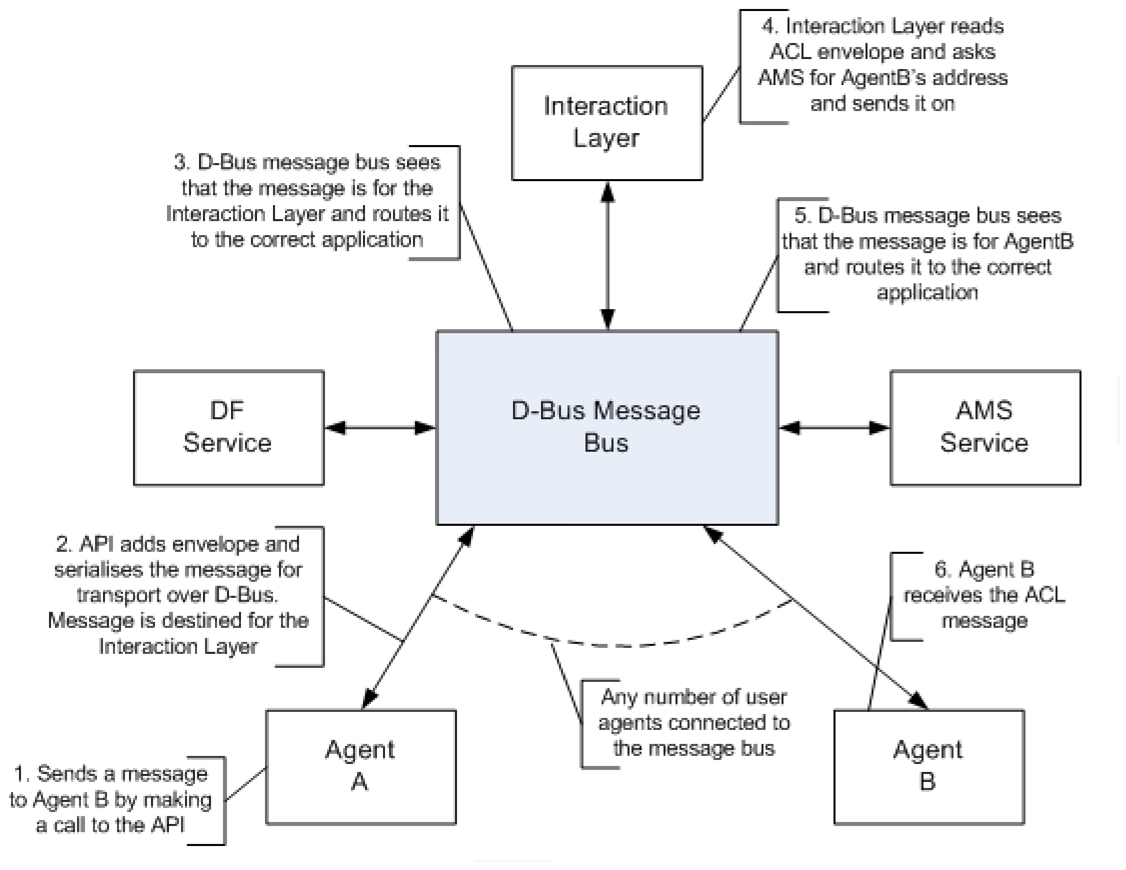

4. Interaction Layer reads ACL envelope and asks AMS for AgentB's address and sends it on
Interaction Layer
3. D-Bus message bus sees that the message is for the Interaction Layer and routes it to the correct application
5. D-Bus message bus sees that the message is for AgentB and routes it to the correct application
DF Service
D-Bus Message Bus
AMS Service
2. API adds envelope and serialises the message for transport over D-Bus. Message is destined for the Interaction Layer
6. Agent B receives the ACL message
Any number of user agents connected to the message bus
Agent A
Agent B
1. Sends a message to Agent B by making a call to the API

Figure 5.1: Role of the D-Bus Message bus for routing ACL messages

(AKT-Bus 2004) (Hui & Preece 2001) and is developed at Aberdeen University. One reason why it is of particular interest is that it is designed to enable inter-operability between knowledge applications allowing them to transfer knowledge in between processes. Essentially what agent communication is, and could therefore be argued to be more suitable for this project than D-Bus whose core reason for development was different.

5.1.4 Multi-Language Support

The API that is developed for the agent platform must be accessible from a variety of different programming languages to enable the incorporation of legacy software. It is beyond the scope of this project to implement a layer that will enable cross language support and therefore it was decided to use an existing tool to provide this functionality. The best tool found is called the Simplified Wrapper and Interface Generator (SWIG) (SWIG 2004) that provides functionality to enable programs written in C/C++ to be connected to programs written in a number of high level programming and scripting languages. The existence of this tool was another reason why the C programming language was chosen. The version of SWIG used for this project was 1.3.21.

5.2 Ancillary Tools

There were a number of tools that were used during the development phase of this project in order to make the implementation easier and to provide contingency plans to prepare for the worst. These tools are described below:

CVS

Management of the source code was handled through a system called Concurrent Versioning System (CVS) (CVS 2004). To support this a CVS server was set up (CVSNT 2004) to hold the source code on a separate machine from where the coding was carried out. The use of a separate machine, along with the central storage mechanism of the CVS repositories provided redundancy and a central location to implement contingency plans and source code back up facilities to ensure that the project was not lost if the worst happened. The ability for tracking versions and changes was also used to revert to old code when required.

GLib

Instead of programming raw C it was decided to make the use of existing stable libraries that implement many of the common utilities that would be required to develop this software, for example creation and manipulation of common data structures. The use of the GLib library (GLib 2004) provided a large code base that was used to simplify the implementation, it also provided safer ways to handle memory management within the code. The D-Bus application that is used for the transport bus also supports GLib bindings that makes the development of programs that use its functionality easier, which helped in the implementation.

Make

To support compiling of the source code the GNU make tool (Make 2004) was used. This required learning to write and maintain make files for the compilation process. It simplified building the code throughout the project and allowed the source to be transported to other machines and compiled in a standard way.

Eclipse

An integrated development environment (IDE) was used to develop the solution, and the choice made was to use the C/C++ plugin for the eclipse platform (Eclipse 2004). The use of this tool enabled the source code to be developed at a quicker rate through the use of the source browsing facilities and syntax highlighting. A managed make project was used to develop the project as this created and maintained the make file for the main solution (for the make tool described above) automatically making the development process simple.

Insight

For debugging the code a graphical front end for the GDB GNU debugger was used called Insight (Insight 2004). This was available for the Linux platform and provided the traditional capabilities including the setting of breakpoints, the use of watches for variables and the ability to step through the code one line at a time to see the interactions. The use of this debugger was invaluable when investigating the reasons behind test failures.

5.3 Overall Architecture

Here we will provide a more concrete description of the architecture of the solution. It should be seen as a natural progression of the explanations given so far in the requirements and design stages, except here it is presented with the specifics of the implementation tools used. Figure 5.2 shows the overview of the how the platform looks.

The D-Bus acts like the glue between all of the agents and the platform allowing communication between them. Figure 5.1 shows that all agents must connect to D-Bus, with each connection permitting two way communication. In order for the messages to be routed to the correct receiver by the message bus each agent must have a unique address, this is a key part of the implementation. In D-Bus the addressing system has 3 aspects, a service, an object path and a member on that object path which combine to form a unique transport end point. The agent platform has a well known service name, and each of the platform services have their own standard object paths and object members. Agents will each have a unique service name assigned to them by the D-Bus message bus when they connect and have standard object path and members. The platform has only one service and not separate ones for each support service as it runs in one process thread rather than as three different ones as described in the design. This set up allows each aspect of the system to be uniquely addressable, and the use of standard well known names for the platform services is required during the bootstrap process for agents. The three components of a D-Bus transport address are combined in the following way to form the transport address element of the agent identifier structure:

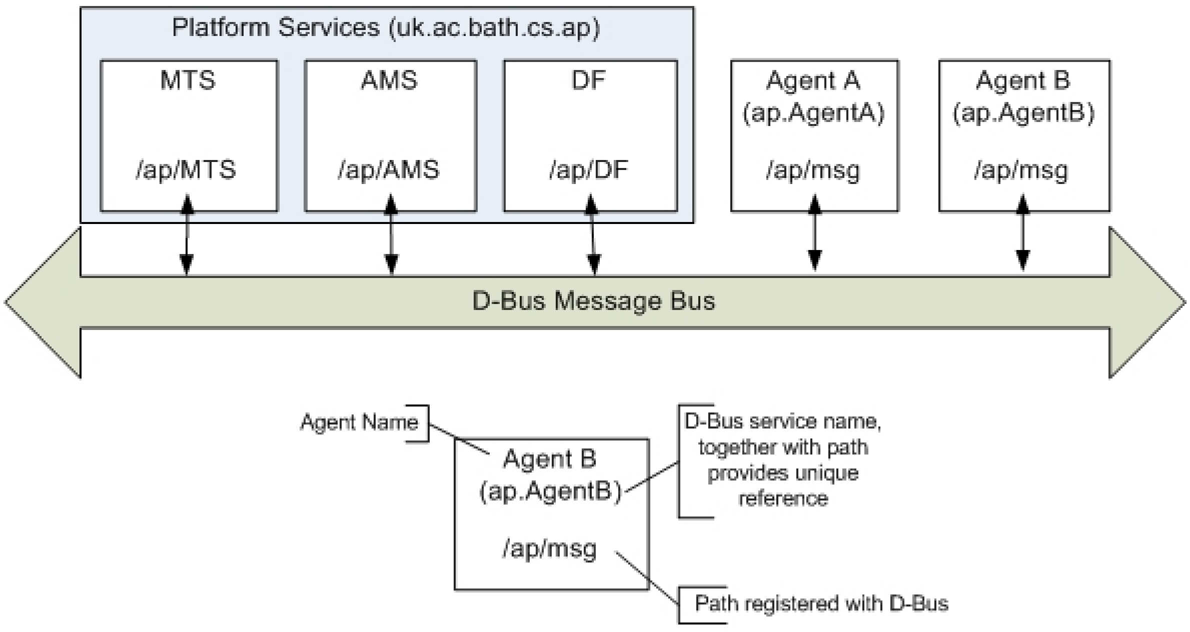


Figure 5.2: Overall Platform Structure

```
<protocol>:<unique-service-name>:<object-path>:<member>
```

For example for the Interaction Layer it becomes:

```
dbus:uk.ac.bath.cs.ap:/ap/MTS:ACLMessage
```

The address structures in the agent identifier are essential for communication. The other aspect of an agent identifier, the agent name, needs to be globally unique which is achieved by the AMS assigning a suitable name to the platform. It was decided that the best option would be to use the fully qualified domain name of the machine on which the platform is running. So an agents name becomes a combination of the name given by the developer and the platform name:

```
<developers-name>@<machine-name>
```

So for an agent called PingAgent running on the machine cs.bath.ac.uk this would become:

```
PingAgent@cs.bath.ac.uk
```

The message delivery process described in the design section shown in figure 4.3 ensures that all messages pass through the interaction layer. This set up enables the easy addition of another message transport protocol (other than D-Bus). Note that since the transport protocol is specified within each transport address entry in the agent identifiers the interaction layer is able to determine how to contact the agent. Therefore if a protocol other than dbus was specified then the interaction layer could pass the responsibility of sending this message to another module that implemented that protocol. Note that the D-Bus protocol will still be used to get the message to the interaction layer in the first place, the other protocol would only be used for external communication. This situation is shown in figure 5.3. This is only made easy due to the design decision to have all messages passing through one central interaction layer, and the nature of the self descriptive transport addresses that contain the protocol required. The solution for this project only provides an implementation for the D-Bus protocol for message transportation.

The specific transport addresses used by the platform services in the format previously described are shown below. These values are assigned to the services when the platform bootstraps and will not change throughout their lifetime.

- *Interaction Layer*. name:- MTS@cs.bath.ac.uk
  address:- dbus:uk.ac.bath.cs.ap:/ap/MTS:msg

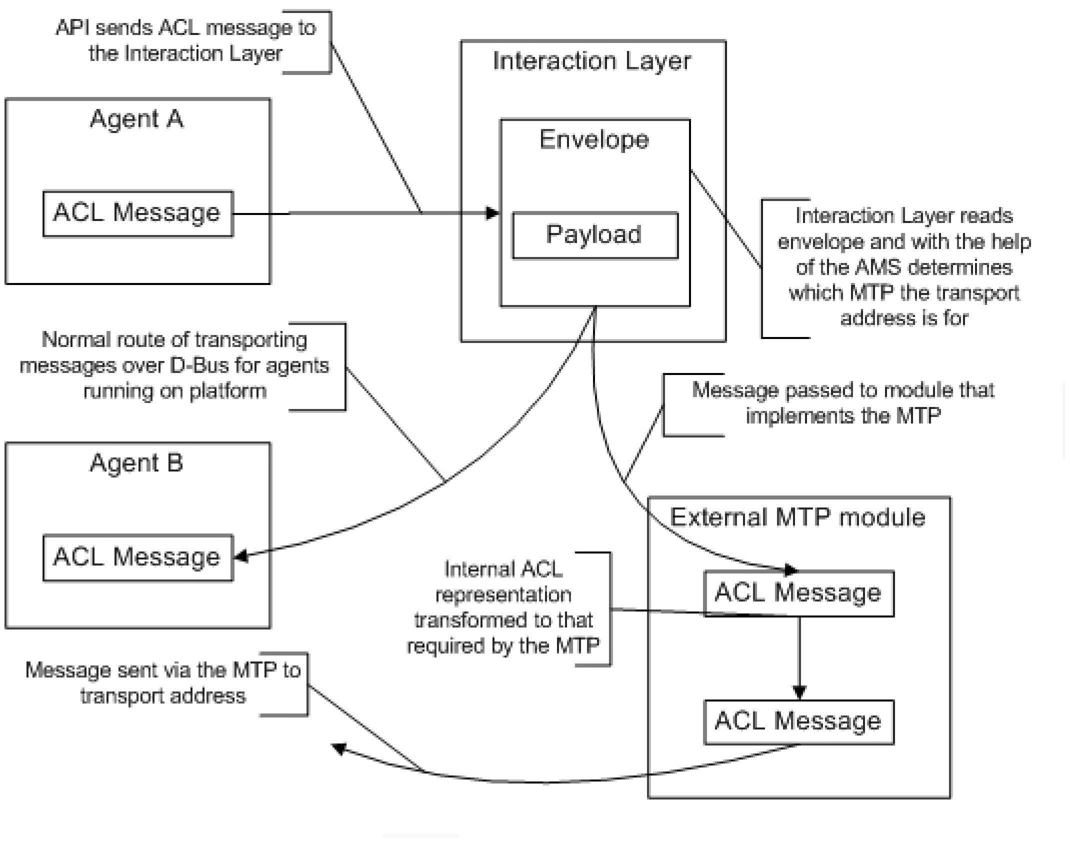

API sends ACL message to the Interaction Layer
Interaction Layer
Agent A
Envelope
ACL Message
Payload
Interaction Layer reads envelope and with the help of the AMS determines which MTP the transport address is for
Normal route of transporting messages over D-Bus for agents running on platform
Message passed to module that implements the MTP
Agent B
External MTP module
ACL Message
Internal ACL representation transformed to that required by the MTP
ACL Message
Message sent via the MTP to transport address
ACL Message

Figure 5.3: Implementation of an external MTP

- *AMS*. name:- AMS@cs.bath.ac.uk
  address:- dbus:uk.ac.bath.cs.ap:/ap/AMS:msg
- *DF*. name:- DF@cs.bath.ac.uk
  address:- dbus:uk.ac.bath.cs.ap:/ap/DF:msg

Details of the individual components featured in figure 5.2 can be found in the following sections.

5.4 Interaction Layer

The interaction layer exists solely to transport agent messages to the correct recipient(s) and this service is built on top of the transport bus. This subsection describes the implementation of this functionality. As stated in the requirements the agent communication language that will be supported by this system is FIPA-ACL (FIPA0061 2002) (FIPA0037 2002). To do this an internal C structure was created to represent the fields of a FIPA-ACL message, which in turn requires another internal structure for representation of agent identifiers. These internal structures are used rather than attempting to manipulate messages as a whole or in a string format to simplify actions and facilitates the coding of the API. The interaction layer also needs a representation of an envelope structure, on which it is dependent, for it to perform message delivery. The details of these structures can be found in appendix D. Whilst the FIPA-ACL message structures are the only language supported by the API effort has been made so that the interaction layer remained independent of this language for agent communication, this is achieved through encompassing all of the information that the interaction layer needs in the envelope.

To facilitate the transportation of these structures over the transport address a codec was written to perform serialisation and deserialisation of the structures. Both the ACL message and envelope structures are made up of fixed parameter value pairs, with the value being of a well defined type. The D-Bus system supports the transportation and receipt of messages containing sequential typed contents allowing for easy access through iterators that loop over the items in the exact order in which they were appended to the message. To provide structure to this sequential data stream the codec was

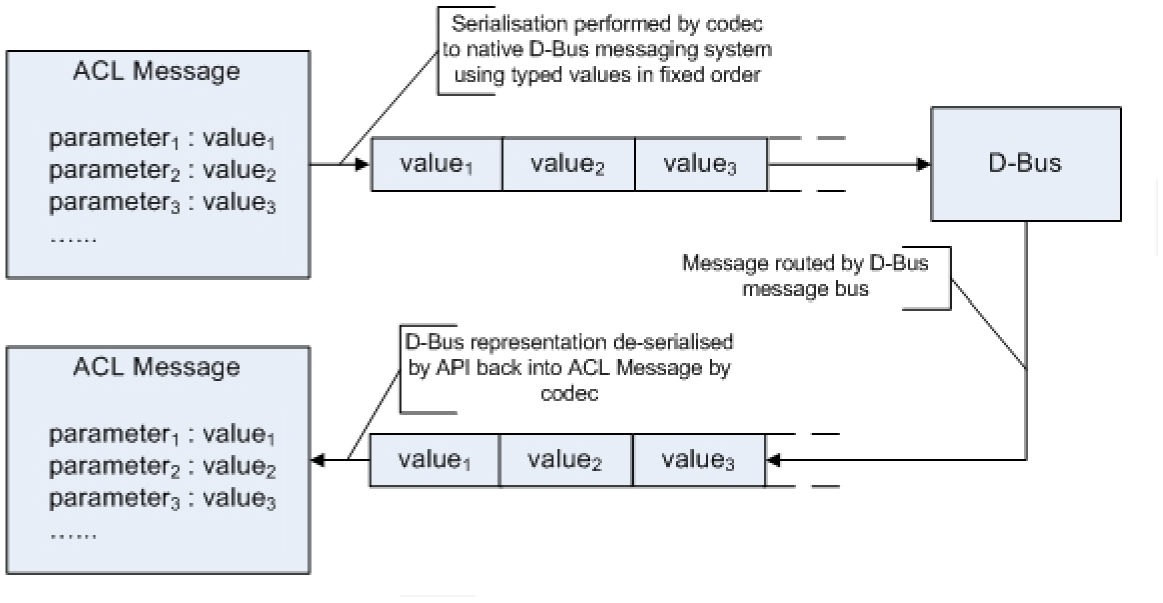

Serialisation performed by codec to native D-Bus messaging system using typed values in fixed order
ACL Message
parameter$_1$: value$_1$
parameter$_2$: value$_2$
parameter$_3$: value$_3$
......
value$_1$
value$_2$
value$_3$
D-Bus
Message routed by D-Bus message bus
D-Bus representation de-serialised by API back into ACL Message by codec
ACL Message
parameter$_1$: value$_1$
parameter$_2$: value$_2$
parameter$_3$: value$_3$
......
value$_1$
value$_2$
value$_3$

Figure 5.4: Serialisation and Deserialisation of ACL messages

written that fixed the order over which the parameter values were sent, at the receiving end the codec knows the order of the parameters and uses that information to build the structured information again. The full details of the specific format of the messages sent over the transport bus for interactions with the platform are shown in appendix F. The actions of the D-Bus codec are shown in figure 5.4.

When the interaction layer receives a message it must first get the codec to deserialise the envelope to into the internal envelope C struct, the payload of the envelope is left alone in order for the interaction layer to remain independent of the agent communication language used. The fields of the envelope are read and used to decide how to deal with the message. For a single recipient the message is simply forwarded onto the agent specified by the agent identifier in the to field. If there are multiple recipients then a copy of the message is sent to all of the agent identifiers specified in the to field one at a time. If the agent identifiers given only have an agent name and do not contain the transport address then an AMS search is performed to determine the transport address for the agent, if the agent cannot be found then the message will be silently discarded and no error message will be sent back to the original sender, since they expect no notification for the successful or unsuccessful delivery of a message. It is up to the receiver to reply according to the protocol being used in the conversation in order to notify the sending agent of receipt.

5.5 Support Services

As stated in the design section the support services run within the same process as the interaction layer and therefore have the same D-Bus service name, but they are uniquely addressable as they have different object paths and members in their full transport address.

Conversations between agents and these support services will not follow the same format as agent-to-agent communication as all messages sent to the support services expect replies even if these are empty. Therefore they do not use the same asynchronous message system for communication, all agents talking to the support services will block until they have received their reply. Since all interactions with the support services are requests for actions or information that effects how the agent interacts with the platform in the future the agent will not want to continue until the action was complete anyway. This pausing for a reply is hidden within the API. Conversation protocols have been implemented for each of the public functions that the AMS and DF provide as listed in the requirements. Structures are used to represent the ontology used for each of these methods, details of which can be found in appendix E. The same codec that is used for transportation of FIPA-ACL structures over D-Bus is used for these management functions. The exact format of the messages sent in the implementation can be found in appendix F which describe the interactions that form the conversation protocols.

5.5.1 AMS

The AMS is required to store a registry of all of the agents that are registered with the platform. This registry is implemented through the use of a linked list of agent identifier structures. Details of the conversations that occur between the user agents and the AMS can be found in appendix F, but some aspects will be discussed here. When an agent registers with the platform it is the responsibility of the AMS to make sure that the name requested by the user agent is unique on the platform by searching its database. The full agent name is assigned by appending the fully qualified domain name of the machine on which it is running, if the name if a duplicate of another then registration is refused. Also the transport address of the user agent is a mandatory element of the agent identifier structure when the agent registers so that there is always at least one known transport location for the agent. The same requirement holds for when an agent modifies their entry in the registry as there must always be at least one transport address on D-Bus.

The AMS also maintains the description of the platform and the support services that it provides. Initially this structure is set up when the platform bootstraps and will at the minimum contain an entry for the interaction layer, the DF and the AMS itself. This structure will be passed onto the user agent when they bootstrap so that it can find these services during its lifetime.

5.5.2 DF

The database of the services offered by the user agents is stored as a linked list using the implementation provided by GLib. Each agent may have either no or exactly one entry in the database, since each entry is extensible it is possible for each agent to advertise an arbitrary number of services within this limit of one actual entry. Details of the structures used in this database can be found in appendix D and the conversation protocols in appendix F.

5.6 Bootstrapping

In this section we will set out the process that is followed for the bootstrapping mechanism for both the platform and agents.

5.6.1 Platform

Upon start up the first thing that is done is that the message bus is started. Each of the three platform services, the interaction layer, AMS and DF, are initialised. A single connection is made to the message bus and the platform service name (uk.ac.bath.cs.ap) is obtained and unique listeners are set up on this connection for each of the services on their well known names. The AMS assigns the name to the platform based on the name of the machine on which it is running. Each agent including the AMS then register themselves with the AMS which uses this information to build the platform description that forms the service root given to agents. Once complete the services are put into the background and they sit around waiting for messages. This whole process is started by running a single script from the terminal, this process is summarised in figure 5.5

5.6.2 Agents

When a user agent starts up it must obtain the location of the D-Bus message bus that is running that has the platform support services connected to it. This location is determined from the environment variable DBUS_SESSION_BUS_ADDRESS which must be set prior to bootstrapping the agent and should be the location of the message bus started by the platform when it bootstrapped. Once connected the agent needs to check for the existence of the well known D-Bus service name for the platform, if it exists then it can continue to interrogate the AMS for the platform description so that it can find the interaction layer and DF services. It then registers listeners for the message bus so that it can receive agent messages from the interaction layer. Finally the agent must register itself with the AMS in order to be able to use the services building its agent identifier from its name and transport address it was supplied when it connected to the message bus. This whole bootstrap process is accessible through one simple API call in order to make this process as simple as possible from the agent developers point of view as this is an artefact of the platform and not associated with the application domain. This process is summarised in figure 5.6

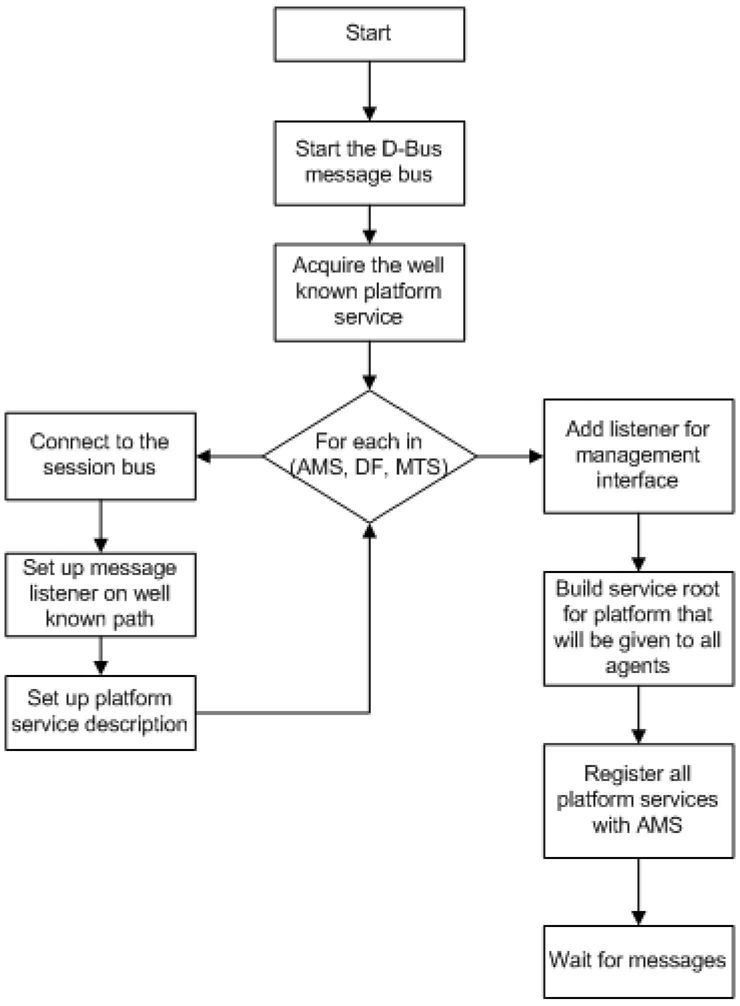

Start
Start the D-Bus message bus
Acquire the well known platform service
For each in (AMS, DF, MTS)
Connect to the session bus
Set up message listener on well known path
Set up platform service description
Add listener for management interface
Build service root for platform that will be given to all agents
Register all platform services with AMS
Wait for messages

Figure 5.5: Platform bootstrap process

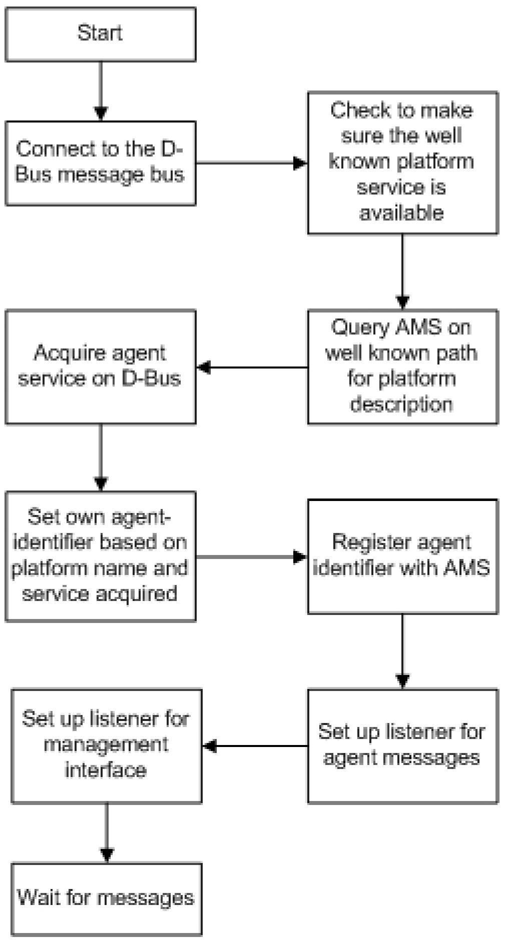

Start
Connect to the D-Bus message bus
Check to make sure the well known platform service is available
Query AMS on well known path for platform description
Acquire agent service on D-Bus
Set own agent-identifier based on platform name and service acquired
Register agent identifier with AMS
Set up listener for agent messages
Set up listener for management interface
Wait for messages

Figure 5.6: User agent bootstrap process

5.7 API

The API is written in C and exposes functions that enable the agents to interact and manipulate the structures that are used by the interaction layer to send and receive messages. It exists to make the programming task of agent developers easier. The API also provides the ability to create and manipulate structures that are used for the ontology for the conversations between the support services and the user agents in order to make this interaction as simple as possible. A simple error structure is also provided that enables the developer to detect any problems whilst using the API calls. The following headings provide a summary of the functions that were implemented as part of the API.

Bootstrap Functions

The API provides one single function call called AgentStart that is used to perform all of the bootstrap process for the user agent. The function returns an agent configuration structure that is used for all future interactions with the services on the platform as it stores the contact locations of the platform services based on the platform description obtained from the AMS. Details of this structure can be found in appendix D. If the bootstrapping is not successful then the platform structure is set to null and the agent will not be able to use the services.

Message Sending

To support the sending of messages functions are provided to initialise and manipulate ACL messages. Once complete the user agent can make a call to the sendMessage function which automates the creation of the envelope and passes the complete structure to the interaction layer using the D-Bus codec. Once the message is sent the agent is able to continue with whatever it wishes there is no need to wait around for confirmation. At present the only form of agent communication language that is supported by the platform is FIPA-ACL although the message sending mechanism is modular and capable of supporting other languages.

Support Services

Utility functions are provided for communication with the platform services that implement the management ontology described in appendix E. All of these functions will communicate directly with the support service in question through the dbus codec using the conversation protocols set out in appendix F. The reply is automatically handled by the API. The aim of these functions is to make the interaction with the support services as intuitive and as easy as possible with all calls made through a single function once the ontology object has been built.

Message Receiving

Methods are provided to support two mechanisms for message retrieval by the user agent. The first has incoming messages stored on a queue and it is the responsibility of the agent developer to check the queue for new messages. The second allows the agent to register a callback function that will be called automatically whenever a message is received on that agents interface. It is the choice of the agent developer of which mechanism that they wish to use. The decoding of the ACL message structure is handled by the API using the underlying dbus codec but the understanding of the content is pushed onto the user agent.

Deregistration

A simple call allows the agent to gracefully exit from the platform. This single call takes care of all of the deregistration from the support services that is required and disconnection from the D-Bus message bus. All agents should make a call to this function before ending in order to notify the platform that they no longer wish to be contacted.

5.7.1 SWIG Interfaces

Here we will provide a brief discussion of the process that was followed to build SWIG interface that allows the API to be used from programming languages other than C. This interface file only contains the functions that provide the functions described in the previous subsection that enable agents to interact with the platform. As a result some of the core abilities of the platform which will not generally be required by agent applications will only be accessible from C, for example an agent written in C would have access directly to the dbus codec, whereas the interface provided by SWIG only allows access indirectly through other functions.

The interface file was written after the API was complete by simple looking at each function in turn and deciding whether it should be made available. Problems were encountered during the implementation of this interface, due to insufficient investigation into the capabilities of SWIG at the initial stages of implementation. It was found half way through due to testing that SWIG was not capable of dealing with passing by reference of complex data types which were used throughout the API for manipulating the platform structures. This meant that some of the functionality that had been written into the API would not be accessible to other languages since any changes made within the API would not be accessible after the function call returned. The solution to this problem was to write extra functions solely for the use of the SWIG interface that would duplicate the functionality provided by these API function calls by wrapping them up in such a way as to remove the passing by reference. This was decided to be the best route to take rather than rewriting these functions as it would have taken too long to rewrite the existing API code and fully re-test it within the time constraints of the project, although that would have resulted in a neater overall solution.

The final interface produced works and exposes the API's functionality so that it can be called via a number of different programming languages so that software written in these can use the functionality provided by the agent platform in order to communicate as agents. This is demonstrated by the test results shown in appendix C

5.8 Summary

This chapter has presented the implementation of the agent platform and its associated support services described in the previous two chapters. The tools and libraries used for this implementation have been justified and their influences described. Most notably the use of D-Bus for the transport bus that enables the inter process communication, and the use of SWIG to provide multi-language support for the API. Important decisions made during the implementation have been detailed especially to do with how the agent messages are sent. The result has been a solution that has met all of the requirements stated in section 3 based on the overall design shown in section 4.

Chapter 6

System Testing

It was decided that the best approach to employ while testing the system was to use a black box (Pressman 1992) rather than white box testing. White box testing would not have been appropriate due to time constraints as suitable coverage could not have been obtained whereas with black box testing all of the important functionality could be examined. The overall aim of this project was to develop a simple interaction layer that enabled agents to communicate with one another through structured messaging therefore it was decided to focus the black box testing on determining whether these aims had been met by the implemented solution. The requirements from section 3 were used as input to the test plan to highlight the specific areas that needed testing. An overview of the process used to develop the plan and the conducting of the tests is provided below and sample test results can be found in appendix C.

As described in the previous chapter the implementation of the system was conducted in an incremental manner and the test plan was designed to complement this. Tests were designed for each individual component of the system and were made in such a way as to only be dependent on other components that had been implemented before the component in question. These tests were created immediately before the implementation of the component and as each stage of the implementation was completed the tests associated with that phase were run to make sure that the required functionality had been met. This continual testing approach was taken as to ensure that errors in one component were not propagated through the system. Also test errors could be solved more efficiently whilst the implementation was freshest in the mind. The tests themselves were in the form of small C programs that called the appropriate functions with some carefully chosen input and checked to make sure that the associated output was as expected outputting its progress and findings to text logs. To facilitate the running of these C programs a simple test-driver program was written in C that enabled a given test to be run by specifying a command line parameter, allowing the running of some tests to automated through the use of scripts. Gradually as more of the final system was implemented this resulted in a test suite that contained more and more tests for the ever growing system. The entire suite to date was run after the implementation of every component as to test the new functionality and the interactions between this and the old to check that no new errors had been created by the changes. The test suite was used at other regular intervals throughout the project independent of implementation milestones to test the system, the results of which were used to focus and drive the development process to ensure that a solid stable platform was being built. The graphical front end for the GNU debugger GDB called Insight was used extensively to investigate the reasons behind why certain tests had failed. As mentioned time constraints on the project influenced the amount of testing that could be conducted therefore the tests were designed to focus on the following elements:

- *Support Services.* Designed to test that all of the required functions as specified in section 3.2.3 are met by the support services that have been implemented. Tests were designed to test both positive and negative results. These tests were written so that they used the conversation protocols for each of the functions as described in appendix F.

- *Agent Communication.* Test agents were written that attempted to send agent messages through the interaction layer. The general approach was to create a client and server agent and to make sure that they were able to communicate with one another. The test programs were designed to make sure that no information was lost during transport and only the correct recipients received the message. The ability of AMS lookup for transport addresses was also extensively tested.

- *Deregistration.* Designed to make sure that the deregistration mechanism provided by the API removed all references from the platform and that no agent would be able to communicate with that agent afterwards.
- *Multi-Language Support.* Simple sender and receiver agents were written in the Java programming language in order to test that the API and hence the interaction layer was indeed accessible from other programming languages. These agents mirrored the functionality of the test agents written for testing agent communication with messages being sent to and from Java agents to C agents.
- *Bootstrapping.* Designed to make sure that the platform initialises itself correctly and that all of the support services are available to the user agents. Tests were written to ensure that the services were reachable by the user agents by requesting the common functions offered by them. A similar approach was used to test the bootstrap mechanism for the user agents by creating simple agents that were able to connect to and interact with the platform.
- *System Testing.* Some very simple agent systems were developed to test the systems abilities as a whole, so that bootstrapping, agent discovery, communication and deregistration all worked as specified. This was the final round of testing once the implementation was complete.

Overall the tests were designed to show positive results of the system developed and to demonstrate that the platform was indeed capable of meeting the requirements and hence the aims of the dissertation. The building of the test suite provided an easy and comprehensive method for regression testing as new functionality was added and enabled the testing process to be interleaved with the implementation. This testing strategy ensured that the system was stable and that each component met its requirements. The results of the test were shown through the production of text logs by the programs/agents involved as to document the processes that they had followed, the results of some of these tests are shown in appendix C.

Chapter 7

Conclusion

The outcome of this project has been the development of an agent platform that consists of a simple message oriented interaction layer and associated support services for agent discovery. The implemented solution meets all of the aims and objectives that were set out in section 1.1 demonstrated by the test results given in appendix C. Most notably the platform enables communication through the use of a standard structured agent communication language between agents written in a number of programming languages. The requirements elicitation process was thorough and successful due to the two pronged approach of using the FIPA standards and evaluation of existing agent platforms. The bottom up approach to the design and incremental implementation process enabled the complexity of the problem to be managed and has resulted in a stable modular solution that can be easily extended. Problems were encountered during the implementation phase whilst using existing tools. In hindsight these problems could have been preempted and hence avoided if more thorough investigation of their capabilities has been conducted at the start. The platform produced here has addressed some of the criticisms that could be made about existing agent platforms. It provides a small simple API that enables agents to interact with the platform whilst still retaining their independence by not being forced to run "inside" the platform under its control. This API is also accessible from multiple programming languages through the SWIG interface providing easier incorporation of legacy software, written in traditional languages, with new agent systems.

The platform developed for this project does not have the same level of functionality of the major existing agent platforms. This is due to three reasons, either the functionality was considered but the decision was made to leave it out as it was part of the criticisms for the platform or it conflicted with the aims of the project, or the functionality was deemed beyond the scope the project. This second set provides many possible avenues for future development and enhancement of the solution, which is made easier the approach adopted to the design and implementation, some of these possibilities are given below:

- Native support within the API for the use of and automatic validation of message contents against an ontology is seen as a very useful extension.
- Similar to above support for using and manipulating contents of agent messages in a well specified content language within the API would be a worthwhile extension, possibly RDF or FIPA-SL.
- A short discussion of how the interaction layer could be extended to include support for external communication with agents running on other platforms using another message transport protocol was given in section 5. Such an extension would open up many other possibilities for the platform.
- On top of the basic messages transport mechanism support for well defined interaction protocols such as the contract net could be provided to make it easier to develop systems using standard conversations.

Agent technologies are an exciting development in computer science and have shown promise in application within a number of domains. They provide a natural abstraction to managing the complexity of building systems that consist of multiple interacting processes which is becoming an increasing requirement of modern software. There have been a number of successful applications of agent technologies that demonstrate that they are very useful, examples are management of raw material flow in manufacturing pipelines in a car plant (Bussman & Schild 2000) and automatic load balancing of mobile phone networks (Bigham & Du 2003). There are still however a number of key challenges that need to be addressed before the use of agent technologies becomes more widespread. In particular there needs to be more research into the development of stable industrial strength agent infrastructure that provides similar functionality to the software developed here and methodologies for system design.

Bibliography

AAP (2004), 'The april agent platform can be found at https://sourceforge.net/projects/networkagent/'. (Checked May 2004).

Agentcities (2004), 'The agentcities home page http://www.agentcities.org/'. (Checked May 2004).

AKT-Bus (2004), 'The akt bus home page http://www.aktors.org/technologies/aktbus/'. (Checked May 2004).

Austin, J. (1962), 'How to do things with words', *Oxford University Press* .

Bigham, J. & Du, L. (2003), 'Cooperative negotiation in a multi-agent system for real-time load balancing of a mobile cellular network', *AAMAS* pp. 14–18.

Botelho, L., Willmott, S., Zhang, T. & Dale, J. (2002), 'Review of content languages suitable for agent communication', *EPFL IC Technical Report* .

Bussman, S. & Schild, K. (2000), 'Self-organizing manafacturing control: An industrial application of agent technology', *4th International Conference on Multi-Agent Systems* .

Collis, J. & Lee, L. (1997), 'Building electronic marketplaces with the zeus agent toolkit'.

CVS (2004), 'The concurrent versioning system (cvs) home page http://www.cvshome.org/'. (Checked May 2004).

CVSNT (2004), 'The cvsnt home page http://www.cvsnt.org/wiki/'. (Checked May 2004).

DBus (2004), 'The dbus home page http://www.freedesktop.org/Software/dbus'. (Checked May 2004).

Dickinson, I. & Wooldridge, M. (2002), 'Towards practical reasoning agents for the semantic web', *AAMAS-03* .

Eclipse (2004), 'The eclipse home page http://www.eclipse.org/'. (Checked May 2004).

FIPA (1996), 'Fipa online specification repository can be found at http://www.fipa.org/repository/index.html'. (Checked May 2004).

FIPA (2004*a*), 'The fipa home page http://www.fipa.org'. (Checked May 2004).

FIPA (2004*b*), 'Fipa interaction protocols specifications http://www.fipa.org/repository/ips.html'. (Checked May 2004).

FIPA0001 (2000), 'Fipa abstract architecture specification http://www.fipa.org/specs/fipa0001'. (Checked May 2004).

FIPA0007 (2001), 'Fipa content language library specification http://www.fipa.org/specs/fipa0007'. (Checked May 2004).

FIPA0023 (2000), 'Fipa abstract management specification http://www.fipa.org/specs/fipa0023'. (Checked May 2004).

FIPA0026 (2000), 'Fipa request interaction protocol specification http://www.fipa.org/specs/fipa00026'. (Checked May 2004).

FIPA0037 (2002), 'Fipa communicative act library specification http://www.fipa.org/specs/fipa0037'. (Checked May 2004).

FIPA0061 (2002), 'Fipa acl message structure specification `http://www.fipa.org/specs/fipa0061`'. (Checked May 2004).

FIPA0067 (2000), 'Fipa agent message transport service specification `http://www.fipa.org/specs/fipa0067`'. (Checked May 2004).

Genesereth, M. & Ketchpel, S. (1994), 'Software agents', *Communications of the ACM* **37**, 48–53.

GLib (2004), 'The glib reference manual `http://developer.gnome.org/doc/API/2.0/glib/index.html`'. (Checked May 2004).

Graesser, A. & Franklin, S. (1996), 'Is it an agent, or just a program? a taxonomy for autonomous agents', *Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Springer Verlag* .

Hui, K. & Preece, A. (2001), 'An infastructure for knowledge communication in akt version 1.0', *White Paper* .

Insight (2004), 'The insight home page `http://sources.redhat.com/insight/`'. (Checked May 2004).

JADE (2004), 'The java agent development framework home page `http://sharon.cselt.it/projects/jade/`'. (Checked May 2004).

Labrou, Y. & Finan, T. (1996), 'Semantics for an agent communication language', *Intelligent Agents IV: Agent Theories Architectures and Languages* .

LEAP (2004), 'The leap project home page `http://leap.crm-paris.com/`'. (Checked May 2004).

Make (2004), 'The gnu make documentation `http://www.sunsite.ualberta.ca/Documentation/Gnu/make-3.79/html_chapter/make_toc.html`'. (Checked May 2004).

Michael, L., McBurney, P. & Preist, C. (2003), 'A roadmap for agent based computing', *Agent Technology: Enabling Next Generation Comuting* .

Neches, R., Finin, T., Fikes, R., Gruber, T., Patil, R., Senator, T. & Swartout, W. (1991), 'Enabling technology for knowledge sharing', *AI Magazine* **12**(3).

Nuin (2004), 'The nuin home page `http://www.nuin.org.uk`'. (Checked May 2004).

Patil, R., Fikes, R. & Patel-Schneider, P. (1993), 'The darpa knowledge sharing effort: Progress report. principles of knowledge representation and reasoning', *Proceedings of the Third International Conference* .

Pressman, S. (1992), *Software Engineering - A Practioners Approach*, Macgraw Hill.

Rao, A. & Georgeff, M. (1996), 'Bdi agents speak out in a logical computable language', *In the proceedings of the 7th European workshop on Modeling Autonomous Agents in a Multi-Agent World (MAAMAW'96)* .

Redhat (2004), 'The redhat home page `http://www.redhat.com`'. (Checked May 2004).

Searle, J. (1969), 'Speech acts: An essay in the philosophy of language', *Cambridge Univeristy Press* .

Stoham, Y. (1997), 'An overview of agent-oriented programming', *Software Agents ed Bradhsaw, AAAI press* .

SWIG (2004), 'The simple wrapper and interface generator (swig) home page `http://www.swig.org/`'. (Checked May 2004).

Wooldridge, M. (2002), *An Introduction to Multiagent Systems*, John Wiley and Sons.

Wooldridge, M. & Jennings, N. (1995), 'Intelligent agents: Theory and practice', *Knowledge Engineering Review* pp. 115–152.

Appendix A

API Use Cases

Agent Startup

1. Agent needs to connect to the transport bus.
2. Agent needs to know the location of the Interaction layer.
3. Agent needs to know the location of the AMS support service.
4. Agent needs to know the location of the DF support service.
5. Agent needs to determine the service root for the agent platform.
6. Agent needs to register with the AMS.
7. Agent needs to be assigned its globally unique name.
8. Agent needs to register its method for receiving the ACL messages.

Sending Messages

1. Agent needs to create and initialise a new ACL message.
2. Agent needs to set the appropriate fields of the ACL message.
3. Agent needs to add the content to this message.
4. Agent needs to send the message to the intended recipient(s).

Receiving Messages

1. Agent needs to know when it has received a message.
2. Agent needs to gain access to the envelope for the message that was used during transport.
3. Agent needs to gain access to the values of the parameters in the ACL message.
4. Agent needs to gain access to the encoding and language used for the content of the message.
5. Agent needs to be able to access the content of the message to determine the meaning.
6. Agent needs to be able to create a reply to this message.

Agent Discovery

1. Agent needs to be able to modify and add entries to the AMS specifying available transport addresses.
2. Agent needs to able to modify and add service entries to the DF describing the services that it offers.
3. Agent needs to be able to convert an agent name into a fully qualified agent identifier.
4. Agent needs to be able to find other agents on the system based on their names.

5. Agent needs to be able to discover other agents running on the interaction layer based on the services that they offer.

Agent Shutdown

1. Agent needs to be able to remove all service entries for itself in the DF.
2. Agent needs to deregister from the AMS support service.
3. Agent needs to disconnect from the transport bus.
4. Agent needs to disconnect from all of the support services running on the platform.

Appendix B

Code Listing

The code written for this project can be found on the disk accompanying this dissertation. It will also be available at `http://www.bath.ac.uk/~ma0cap/AP` until June 2004. The file readme.html provides a description of how the source code files are structured, highlights files that may be of particular interest and gives instructions on how to compile it.

Appendix C

Test Results

This section contains the log output of some of the tests conducted. For the sake of brevity not all of the test results are presented here, rather a representative selection has been chosen that demonstrate that the solution meets the core aims and objectives of the project. The output shown is always in the form of text logs. Where multiple programs/agents were involved in a test the output for each process is identified at the top with a line separating each program, the interactions are shown by giving the log output in chronological order interleaving the different processes. The tests are broken down into sections for the type of functionality that they were testing.

C.1 Bootstrapping

Below are the test results for the bootstrap process of both the platform and an example user agent. Both of the tests show a successful bootstrap where the platform is set up correctly and is contactable.

```
PLATFORM BOOT
Connection to D-Bus successful
Platform base service is :1.138
Plaform acquired the service uk.ac.bath.cs.CAP
Platform registered handler for /ap/terminate
MTS: Listening on /ap/MTS
AMS: Listening on /ap/AMS
DF: Listening on /ap/DF
Platform sleeping...
MTS: Ping message received from :1.139
AMS: Ping message received from :1.139
DF: Ping message received from :1.139
Platform : Sent terminate message from :1.140
Platform Terminating...
MTS Disconnecting from the DBus
AMS disconnecting from the DBus
DF: disconnecting from the DBus

AGENT BOOT
Obtaining platform description
Platform Name : rpc-ma0cap
MTS found at : dbus:uk.ac.bath.cs.CAP:/ap/MTS:msg
AMS found at : dbus:uk.ac.bath.cs.CAP:/ap/AMS:msg
DF found at : dbus:uk.ac.bath.cs.CAP:/ap/DF:msg
Acquired service ap.test
My identifier is
(agent-identifier :name test@rpc-ma0cap :addresses
(dbus:ap.test:/ap/msg:agentMessage ))
Registering with AMS...
Registration with AMS suceeded
Listening on /ap/msg
```

```
Agent bootstrapped successfully
```

C.2 AMS

For the sake of brevity only a few tests for the AMS will be presented here that demonstrate some of the core functions working. In order to perform these tests some simple agents were created that would simulate interactions between the AMS and agents.

Register

The platform was bootstrapped and asked to print the contents of its AMS agent directory. Then a simple "boot" agent was run that bootstrapped and then immediately exited deliberately bypassing the normal deregistration procedure which would have removed its AMS entry. After the agent had finished the platform was requested to print the contents of the directory again. The results below clearly demonstrate that the AMS now also contains an entry for the "boot" agent as well as the platform services.

```
There are 3 entries in the directory
(agent-identifier :name AMS@rpc-ma0cap :addresses
(dbus:uk.ac.bath.cs.CAP:/ap/AMS:msg ))
(agent-identifier :name MTS@rpc-ma0cap :addresses
(dbus:uk.ac.bath.cs.CAP:/ap/MTS:msg ))
(agent-identifier :name DF@rpc-ma0cap :addresses
(dbus:uk.ac.bath.cs.CAP:/ap/DF:msg ))
AMS: received register request from :1.146
AMS: identifier read as (agent-identifier :name boot@rpc-ma0cap :addresses
(dbus:ap.boot:/ap/msg:agentMessage ))
AMS: registration succeeded
There are 4 entries in the directory
(agent-identifier :name AMS@rpc-ma0cap :addresses
(dbus:uk.ac.bath.cs.CAP:/ap/AMS:msg ))
(agent-identifier :name MTS@rpc-ma0cap :addresses
(dbus:uk.ac.bath.cs.CAP:/ap/MTS:msg ))
(agent-identifier :name DF@rpc-ma0cap :addresses
(dbus:uk.ac.bath.cs.CAP:/ap/DF:msg ))
(agent-identifier :name boot@rpc-ma0cap :addresses
(dbus:ap.boot:/ap/msg:agentMessage ))
```

Search

The platform was started and a simple server agent was run that performed the bootstrap process and then simply went to sleep. The AMS was then asked to print the contents of the agent directory. Another agent (the client) was then executed that searched the AMS for the server agent, the results below show that the search was successful and the client did retrieve the correct agent identifier that could be used in subsequent communication with the server.

```
AMS
There are 4 entries in the directory
(agent-identifier :name AMS@rpc-ma0cap :addresses
(dbus:uk.ac.bath.cs.CAP:/ap/AMS:msg ))
(agent-identifier :name MTS@rpc-ma0cap :addresses
(dbus:uk.ac.bath.cs.CAP:/ap/MTS:msg ))
(agent-identifier :name DF@rpc-ma0cap :addresses
(dbus:uk.ac.bath.cs.CAP:/ap/DF:msg ))
(agent-identifier :name server@rpc-ma0cap :addresses
(dbus:ap.server:/ap/msg:agentMessage ))

CLIENT AGENT
Performing AMS search for agent server@rpc-ma0cap
```

```
AMS search ok
(agent-identifier :name server@rpc-ma0cap :addresses
(dbus:ap.server:/ap/msg:agentMessage ))
```

C.3 DF

For the sake of brevity only a few tests for the DF will be presented here that demonstrate some of the core functions working. In order to perform these tests some simple agents were created that would simulate interactions with the DF

Register

The platform was started, and then a simple agent was run that bootstrapped and then entered a dummy agent description into the DF. The platform was requested to print the contents of the DF directory both before and after the registration took place. The results below show that registration was successful.

```
PLATFORM
There are 0 entries in the DF directory

TEST AGENT
My DF entry is (df-agent-description :name (agent-identifier :name test@rpc-ma0cap :addresses
(dbus:ap.test:/ap/msg:agentMessage )) :services :protocols fipa-request :ontologies ap-management
:languages fipa-sl0 fipa-sl1 fipa-sl2 )
Registering entry with the DF
DF registration suceeded

PLATFORM
DF: register request received from :1.127
DF: DF entry read as : (df-agent-description :name (agent-identifier :name test@rpc-ma0cap
:addresses (dbus:ap.test:/ap/msg:agentMessage )) :services :protocols fipa-request :ontologies
ap-management :languages fipa-sl0 fipa-sl1 fipa-sl2 )

There are 1 entries in the DF directory
(df-agent-description :name (agent-identifier :name test@rpc-ma0cap :addresses (dbus:ap.test:/ap/ms
)) :services :protocols fipa-request :ontologies ap-management :languages fipa-sl0 fipa-sl1
fipa-sl2 )
```

Search

The platform was bootstrapped and then two "server" agents were started that registered an entry with the DF advertising that they understood the test-server ontology. A simple agent was bootstrapped and added its own entry to the DF. It then continued to perform a search for all agents on the platform that advertised that they were capable of using the test-server ontology. The results below show that the search returned the correct response.

```
PLATFORM
There are 2 entries in the DF directory
(df-agent-description :name (agent-identifier :name server1@rpc-ma0cap :addresses (dbus:ap.server1
)) :services :protocols fipa-request :ontologies test-server :languages )
(df-agent-description :name (agent-identifier :name server2@rpc-ma0cap :addresses (dbus:ap.server2
)) :services :protocols fipa-request :ontologies test-server :languages )

AGENT
My DF entry is
(df-agent-description :name (agent-identifier :name test@rpc-ma0cap :addresses (dbus:ap.test:/ap/ms
)) :services :protocols fipa-request :ontologies ap-management :languages fipa-sl0 fipa-sl1
fipa-sl2 )
Registering entry with the DF
```

```
DF registration suceeded
Searching DF for agents advertising test-server ontology

PLATFORM
DF: search request received from :1.7
DF: template read as : (df-agent-description :name (agent-identifier :name :addresses)
:services :protocols :ontologies test-server :languages )

AGENT
Search produced 2 results
(df-agent-description :name (agent-identifier :name server1@rpc-ma0cap :addresses (dbus:ap.server1
)) :services :protocols fipa-request :ontologies test-server :languages )
(df-agent-description :name (agent-identifier :name server2@rpc-ma0cap :addresses (dbus:ap.server2
)) :services :protocols fipa-request :ontologies test-server :languages )
```

C.4 Interaction Layer

Tests were run to ensure that the interaction layer was capable of routing messages to the correct recipients (multi-cast messages were used in the tests). The test messages sent deliberately did not have the transport addresses complete for the agent identifiers used in the to fields of the envelope to test whether the interaction layer would correctly perform the appropriate AMS lookups for the transport addresses. In the test scenario two server agents were set up that would respond to ”ping” messages, and a client was started that created a message to ping both of these servers. The results below demonstrate that communication was successful. Only one of the server agents responding is shown as exactly the same output was given by the other. The client did receive both a reply from server1 and server2

```
AGENT
Sending the message
(query-if
:sender (agent-identifier :name test@rpc-ma0cap
:addresses (dbus:ap.test:/ap/msg:agentMessage ))
:receiver (set (agent-identifier :name server1 :addresses)
(agent-identifier :name server2 :addresses) )
:reply-to (set (agent-identifier :name server1 :addresses) )
:content ping
:language string
:encoding std.string
:ontology ap-tests
)
Message sent

PLATFORM
MTS: Received route request from :1.75
MTS: message sent by test@rpc-ma0cap
MTS: Delivering message to dbus:ap.server1:/ap/msg:agentMessage
MTS: Delivering message to dbus:ap.server2:/ap/msg:agentMessage

SERVER AGENT
Received message
(query-if
:sender (agent-identifier :name test@rpc-ma0cap
:addresses (dbus:ap.test:/ap/msg:agentMessage ))
:receiver (set (agent-identifier :name server1 :addresses)
(agent-identifier :name server2 :addresses) )
:content ping
:language string
:ontology ap-tests
)
```

```
Sending reply

PLATFORM
MTS: Received route request from :1.73
MTS: message sent by server1@rpc-ma0cap
MTS: Delivering message to dbus:ap.test:/ap/msg:agentMessage

AGENT
Received Reply
(inform
:sender (agent-identifier :name server1@rpc-ma0cap
:addresses (dbus:ap.server1:/ap/msg:agentMessage ))
:receiver (set (agent-identifier :name test@rpc-ma0cap
:addresses (dbus:ap.test:/ap/msg:agentMessage )) )
:content im here
:language string
:ontology ap-tests
)
```

C.5 Multi-Language Support

To test the SWIG interface some of the tests described above were re-created but this time the test agents were written in Java that used the API functions made public through the SWIG interface. For the sake of brevity only two of these tests are shown.

Bootstrap

The platform was bootstrapped using a pure C implementation and then a simple test agent written in Java was bootstrapped to demonstrate that it could successfully use the API exposed via SWIG to connect to the platform and receive messages. This is demonstrated by sending a ”ping” message over the management interface.

```
AGENT
ma0cap@rpc-ma0cap Java$ java -Djava.library.path=../ main agent
Obtaining platform description
Platform Name : rpc-ma0cap
MTS found at : dbus:uk.ac.bath.cs.CAP:/ap/MTS:msg
AMS found at : dbus:uk.ac.bath.cs.CAP:/ap/AMS:msg
DF found at : dbus:uk.ac.bath.cs.CAP:/ap/DF:msg
Platform description obtained
Acquired service ap.JavaAgent
My identifier is
(agent-identifier :name JavaAgent@rpc-ma0cap :addresses
(dbus:ap.JavaAgent:/ap/msg:agentMessage ))
Registering with AMS...
Registration with AMS suceeded
Listening on /ap/msg
MANAGEMENT: Ping message received from :1.35
De-Registering from the DF
Unable to de-register from DF - No entry was found for that agent
De-Registering from the AMS
De-Registration from AMS successful

PLATFORM
AMS: received request for platform description from :1.34
AMS: received register request from :1.34
AMS: identifier read as (agent-identifier :name JavaAgent@rpc-ma0cap :addresses (dbus:ap.JavaAgent
))
AMS: registration succeeded
```

```
DF: de-register request received from :1.34
DF: de-registering JavaAgent@rpc-ma0cap
AMS: received de-register request from :1.34
AMS: agent to de-register is JavaAgent@rpc-ma0cap
```

Communication

A single server agent was started that would simply reply to a ping message with the content ”i'm here”. A test agent written in Java was bootstrapped that then proceeded to send a message to the server and wait for a reply. The results below show that the Java agent behaved exactly the same as the equivalent C agent shown in the previous interaction layer tests. The server agents remain unchanged from the previous tests so its output is not shown

```
AGENT
ma0cap@rpc-ma0cap Java$ java -Djava.library.path=../ main agent JavaAgent
Obtaining platform description
Platform Name : rpc-ma0cap
MTS found at : dbus:uk.ac.bath.cs.CAP:/ap/MTS:msg
AMS found at : dbus:uk.ac.bath.cs.CAP:/ap/AMS:msg
DF found at : dbus:uk.ac.bath.cs.CAP:/ap/DF:msg
Acquired service ap.JavaAgent
My identifier is
(agent-identifier :name JavaAgent@rpc-ma0cap :addresses
(dbus:ap.JavaAgent:/ap/msg:agentMessage ))
Registering with AMS...
Registration with AMS suceeded
Listening on /ap/msg
Agent bootstrapped successfully
Envelope and payload is
(
:from (agent-identifier :name JavaAgent@rpc-ma0cap
:addresses (dbus:ap.JavaAgent:/ap/msg:agentMessage ))
:to (agent-identifier :name server :addresses)
:acl-representation dbus-acl
)
(query-if
:sender (agent-identifier :name JavaAgent@rpc-ma0cap
:addresses (dbus:ap.JavaAgent:/ap/msg:agentMessage ))
:receiver (set (agent-identifier :name server :addresses) )
:content (ping)
:language string
:encoding std.string
:ontology ap-tests )
Message Sent
Sleeping

PLATFORM
MTS: Received route request from :1.21
MTS: message sent by JavaAgent@rpc-ma0cap
MTS: Delivering message to dbus:ap.server:/ap/msg:agentMessage
MTS: Received route request from :1.20
MTS: message sent by server@rpc-ma0cap
MTS: Delivering message to dbus:ap.JavaAgent:/ap/msg:agentMessage

AGENT
Message received and added to queue
Envelope and payload is
(
:from (agent-identifier :name server@rpc-ma0cap
:addresses (dbus:ap.server:/ap/msg:agentMessage ))
:to (agent-identifier :name JavaAgent@rpc-ma0cap
:addresses (dbus:ap.JavaAgent:/ap/msg:agentMessage ))
:acl-representation dbus-acl
:intended-receiver (agent-identifier :name JavaAgent@rpc-ma0cap
:addresses (dbus:ap.JavaAgent:/ap/msg:agentMessage ))
)
(inform
:sender (agent-identifier :name server@rpc-ma0cap
:addresses (dbus:ap.server:/ap/msg:agentMessage ))
:receiver (set (agent-identifier :name JavaAgent@rpc-ma0cap
:addresses (dbus:ap.JavaAgent:/ap/msg:agentMessage )) )
:content im here
:language string
:ontology ap-tests )
```

Appendix D

Platform Structures

This section describes the core structures that are used throughout the implementation of the agent platform. The structures described here are required in order for the platform to provide all of the functionality that was set out in the requirements specification. These internal representations are used to make the process of developing the API that will manipulate this information easier.

D.1 Agent Identifier

Every agent running on the platform, including the platform services, must have an agent-identifier in order for them to be uniquely contactable. See table D.1

Parameter	***Type***	***Presence***	***Description***
`name`	string	Mandatory	This is the name of the agent assigned by the user with the platform name appended. Once set it cannot be changed throughout the lifetime of the agent
`addresses`	set of string	at least one	The transport addresses on which the agent can be contacted. Must always contain at least one address that is on the D-Bus

Table D.1: Agent Identifier Structure

D.2 Message Envelope

Every ACL message that is sent by a user agent is encapsulated in an envelope created by the API and read by the interaction layer in order to provide the semantics for delivery. The D-Bus codec deals with the serialisation and deserialisation of the structure when it is transported over D-Bus to the interaction layer. See table D.2

Parameter	***Type***	***Presence***	***Description***
`to`	set of AID	at least 1	The agents to which the message should be sent
`from`	AID	mandatory	The agent who sent the message
`acl-representation`	string	mandatory	How the ACL message is represented
`intended-receiver`	AID	optional	If specified, this is the agent that the message should be sent to, it has priority over the to field

Table D.2: Envelope Structure

D.3 Agent Message

Every message that is sent within the platform is packaged up as an agent message, which has both an envelope and its associated message as its payload. See table D.3

Parameter	***Type***	***Presence***	***Description***
`envelope`	Envelope	mandatory	The envelope for the message that is intended for the MTS service that is uses to transport the message to its final destination
`payload`	ACL message	mandatory	The message that the agent sent, it is ignored by the MTS service and just sent on to the recipient

Table D.3: Agent Message Structure

D.4 Platform Description

The platform has exactly one of these structures that describes the platform services that are running. It is maintained by the AMS and initialised during the platform bootstrap process. This structure is returned to the user agents upon request when they start and forms the service root that they use to contact the platform services. See tables D.4 and D.5

Parameter	***Type***	***Presence***	***Description***
`name`	string	mandatory	The name of the service - one of (MTS, DF, AMS)
`address`	string	mandatory	The address on which this service can be contacted on

Table D.4: Platform Service Description Structure

Parameter	***Type***	***Presence***	***Description***
`name`	string	mandatory	The name of the agent platform. Usually set to the fully qualified domain name of the machine running the platform
`services`	set of service descriptions	at least 3	The list of platform services

Table D.5: Platform Description Structure

D.5 DF

There is exactly one DF structure that contains all of the configuration information for the DF platform service running on the platform. The DF is solely responsible for maintaining the information in this structure. See table D.6

D.6 AMS

There is exactly one AMS structure that contains all of the configuration information for the AMS platform service running on the platform. The AMS is solely responsible for maintaining the information in this structure. See table D.7

D.7 ACL Message

This structure is used to represent every ACL message sent using the platform. The structure is required to enable the API to expose functions that enable agent developers to manipulate the parameters of

Parameter	***Type***	***Presence***	***Description***
`configuration`	agent configuration	mandatory	The configuration structure for the DF
`description`	platform service desc	mandatory	Description of the platform service
`agentDirectory`	set of agent service desc see 4.6	mandatory	The agent directory for the yellow page services

Table D.6: Directory Facilitator Structure

Parameter	***Type***	***Presence***	***Description***
`configuration`	agent configuration	mandatory	The configuration structure for the DF
`description`	platform service desc	mandatory	Description of the platform service
`agentDirectory`	set of agent-identifiers	mandatory	The agent directory for the white page services

Table D.7: Agent Management Service Structure

the message. The structure is serialised and deserialised when it is sent of the D-Bus for transport between agents using the D-Bus codec. See table D.8

D.8 Agent Configuration

Every agent that is running on the platform, including the platform services, have a structure that stores all of their configuration. For user agents the maintenance of this structure is largely conducted by the API in order to make the agent developers lifer easier. See table D.9

Parameter	***Type***	***Presence***	***Description***
`performative`	string	mandatory	Defines the intended meaning of the content. Must be one of the FIPA approved communicative acts specified in (FIPA0037 2002)
`sender`	AID	mandatory	The agent who sent the message
`receivers`	set of AID	at least 1	The agents to whom this message should be sent
`reply to`	set of AID	optional	If specified the receiving agent should reply to these agents rather than the sender of the message
`language`	string	optional	The language that the content is expressed in
`encoding`	string	optional	The encoding used for the content
`ontology`	string	optional	The ontology that the content is rooted in
`protocol`	string	optional	The interaction protocol used for this conversation as specified in (FIPA 2004*b*)
`conversation ID`	string	optional	The conversation that this message belongs to - originally set by the agent that starts the conversation
`reply-with`	string	optional	Reference to be used when replying to this message
`in-reply-to`	string	optional	Reference to the message that this is a reply to
`reply-by`	string	optional	Time represented in UTC by which a reply is expected
`content`	string	optional	The content of the message that is the knowledge/belief that the sending agent is trying to convey

Table D.8: ACL Message Structure

Parameter	***Type***	***Presence***	***Description***
`connection`	D-Bus connection	mandatory	This agents connection to the session D-Bus
`main loop`	GMain loop	mandatory	The main loop that the D-Bus connection was set up with by the API that can be used to sleep and wait for messages
`base service`	string	mandatory	Artefact of using D-Bus - each connection has a unique base service
`identifier`	AID	mandatory	The identifier structure for this agent that holds its name and transport addresses
`message queue`	List	mandatory	List of messages that have been received by the agent and have not yet been dealt with
`MTS address`	string	mandatory	The transport address on the D-Bus that should be used for all interactions with the MTS platform service
`DF address`	string	mandatory	The transport address on the D-Bus that should be used for all interactions with the DF platform service
`AMS address`	string	mandatory	The transport address on the D-Bus that should be used for all interactions with the AMS platform service
`platform name`	string	mandatory	The name of the platform. Used when building the agents full name so that it is globally unique
`DF entry`	Agent DF desc	optional	The agents entry that is held in the DF for yellow page services
`conversation counter`	int	mandatory	Counter used to make sure that all conversations id's for conversations that this agent initiates are unique
`callback fn`	fn pointer	optional	Function registered by the user to be called whenever a ACL message is received. If not specified then all messages are place on the queue

Table D.9: Agent Configuration Structure

Appendix E

Management Ontologies

This section provides details of the classes of objects that make up the management ontology. They are used in conversations between agents and the AMS and DF. The tables below describe the structures used for this management ontology the precise format of how these structures are sent over D-Bus is described by the conversation protocols given in appendix F.

E.1 AMS

The AMS provides a white page service to the user agents running on the platform, in order to perform this function it stores a directory of all of the agents that are registered, the structure of this database is shown in table E.1.

Parameter	***Type***	***Presence***	***Description***
`name`	string	Mandatory	This is the name of the agent assigned by the user with the platform name appended. Once set it cannot be changed throughout the lifetime of the agent. This is the key field used when searching the directory
`addresses`	set of string	at least one	The transport addresses on which the agent can be contacted on. Must always contain at lest one address that is on the D-Bus

Table E.1: Structure of the agent directory maintained by the AMS

E.2 DF

The DF provides yellow page services to the agents running on the platform allowing user agents to discover other agents based on the services that they offer. In order to do this it maintains a registry of the services offered by agents which have been registered, the structure for each entry is described in tables E.2 and E.3.

Parameter	***Type***	***Presence***	***Description***
`name`	string	required	The name of the service provided by the agent
`type`	string	optional	The type of the service provided
`protocols`	set of string	optional	The protocols supported by the agent for this service
`ontologies`	set of string	optional	The ontologies supported by the agent for this service
`languages`	set of string	optional	The content languages supported by the agent for this service

Table E.2: Agent service structure that forms part of an agents DF entry

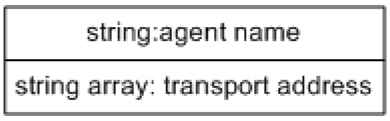
Parameter	***Type***	***Presence***	***Description***
`id`	agent-identifer	required	The identifier for the agent that owns this DF entry. This is the key field that ensures there is only one entry per agent
`services`	set of DF services	optional	Description of the services offered by the agent see table E.2
`protocols`	set of string	optional	The protocols supported by the agent
`ontologies`	set of string	optional	The ontologies supported by the agent
`languages`	set of string	optional	The content languages understood by the agent

Table E.3: Structure of the agent directory maintained by the DF

Appendix F

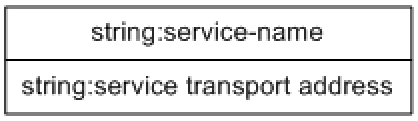
Support Service Conversation Protocols

This section provides details of the protocols used in conversations with the platform services. It lays out the precise details of how the management ontology objects described in appendix E are sent over D-Bus. The format, order and types of messages are specified along with reasons for why errors are sent. The D-Bus message bus provides a system to transmit data using typed contents for basic types like int and string and arrays of these types. This basic types are used wherever possible to build the structures but the system cannot cope with sending arrays of complex types therefore a mechanism was built on top of this functionality to do this. All complex arrays are sent by first an integer denoting how many items are in the array and then each complex type follows one after another encoded exactly as it would if only one was being sent. The following diagrams show the exact order and types of the contents of messages that must be used when talking to the platform services. The tables used for individual messages should be read top down and that is the order that the contents must appear - the order is crucial. For each item of the content its type is specified first, and the field of the ontology object that should be sent is given second. The labels on the arrows denote the name of the member on the services interface that should be called, which is a required part of a function call message in D-Bus. All types are native to D-Bus except for arrays of complex types, they are denoted by array<complex-type> which must be encoded using the array mechanism described above. If the value of the ontology object is optional and not specified then an empty string must be sent. Any contents of messages that appear in quotes denote a string constant that must be sent exactly as written. All of the conversations are initiated by the agent, and all of them are two way conversations as the platform service will always reply. If the action does not require any results to be returned there will still always be a reply. Success is denoted by the string ”OK” otherwise an error message is returned giving the reason why the action could not be performed. First the message format for some of the common ontology objects are given that are used in arrays for other messages, after that the conversation protocols are given for each of the public functions offered by the platform services.



string:agent name
string array: transport address

Figure F.1: Message Format: Agent Identifier



string:service-name
string:service transport address

Figure F.2: Message Format: Platform Service

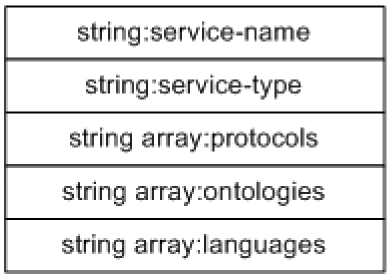


Figure F.3: Message Format: DF Service

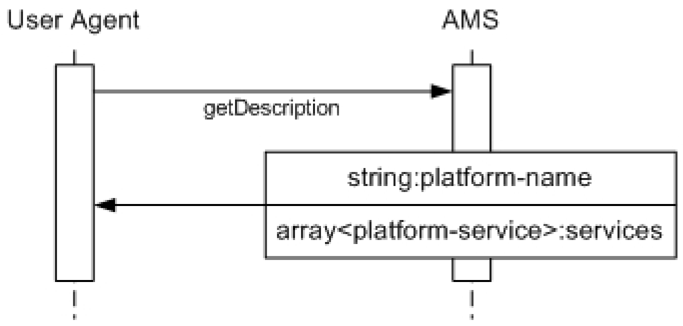


Figure F.4: Conversation Protocol: AMS get-description

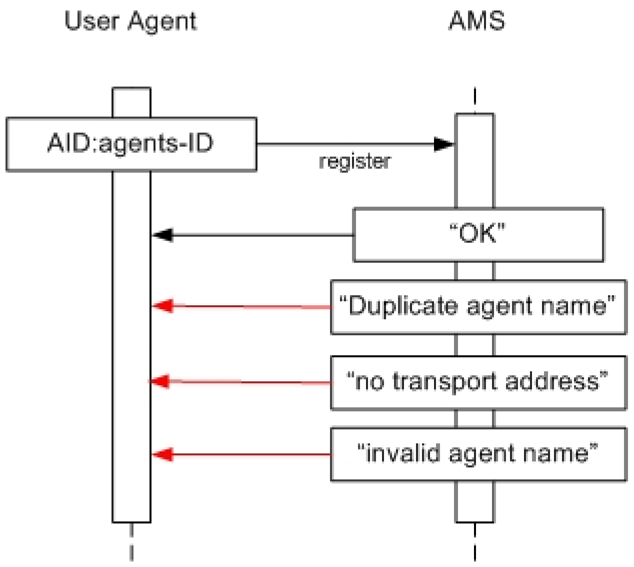


Figure F.5: Conversation Protocol: AMS register

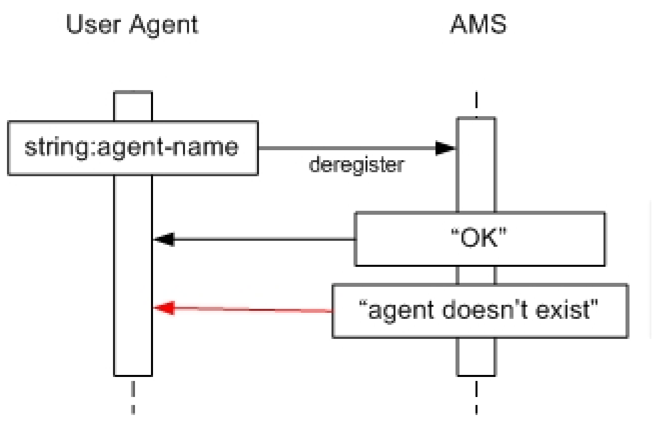


Figure F.6: Conversation Protocol: AMS de-register

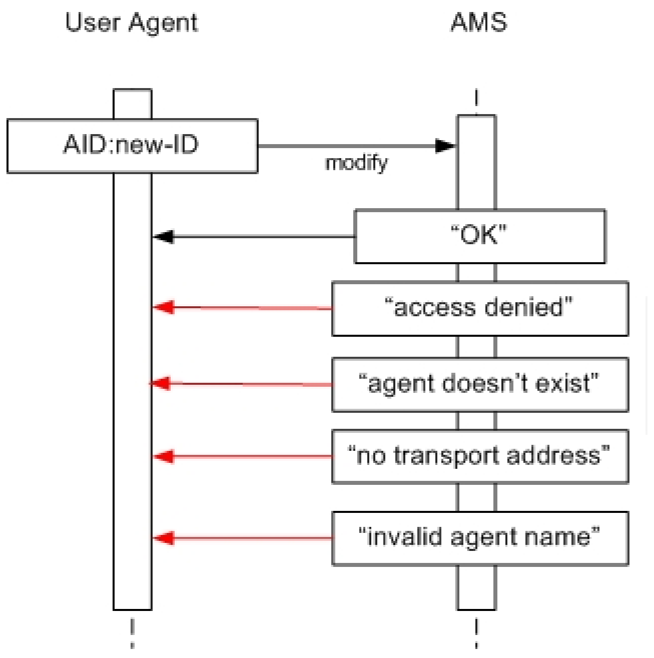


Figure F.7: Conversation Protocol: AMS modify

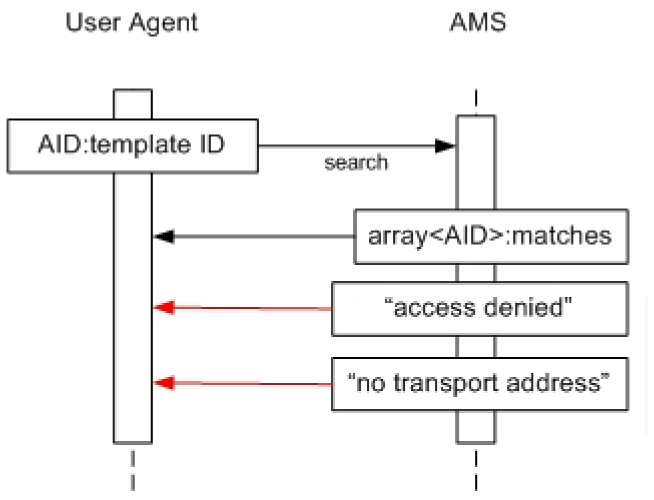


Figure F.8: Conversation Protocol: AMS search

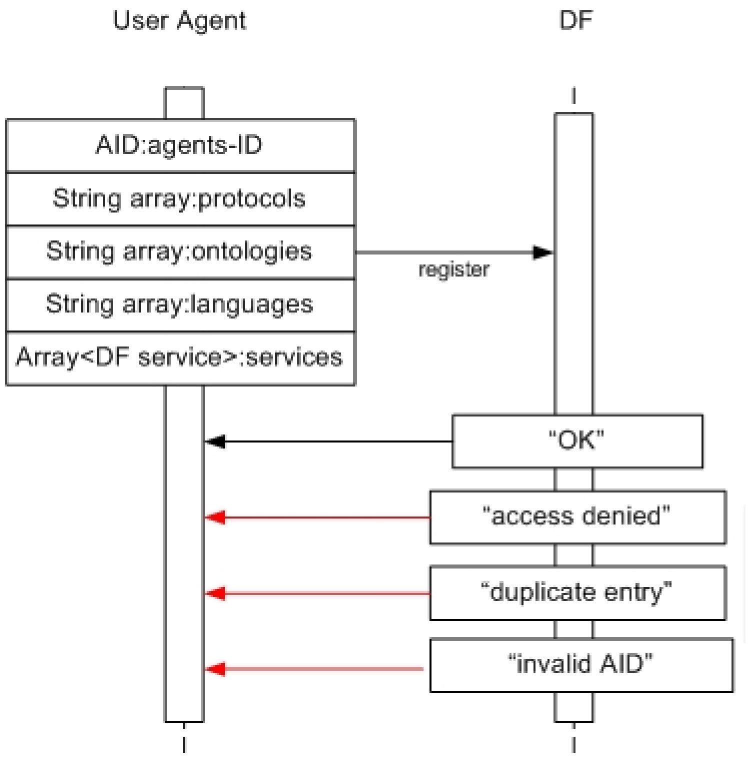


Figure F.9: Conversation Protocol: DF register

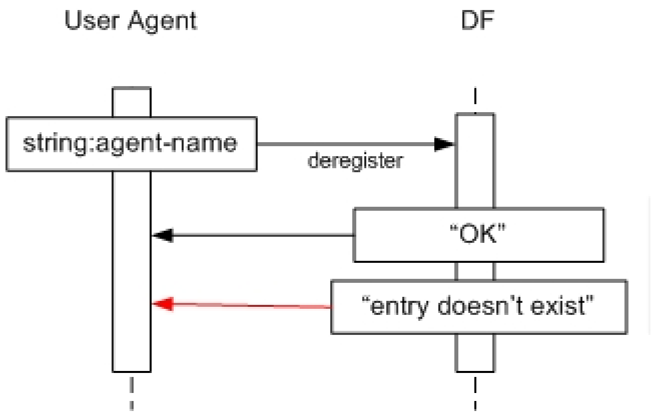


Figure F.10: Conversation Protocol: DF de-register

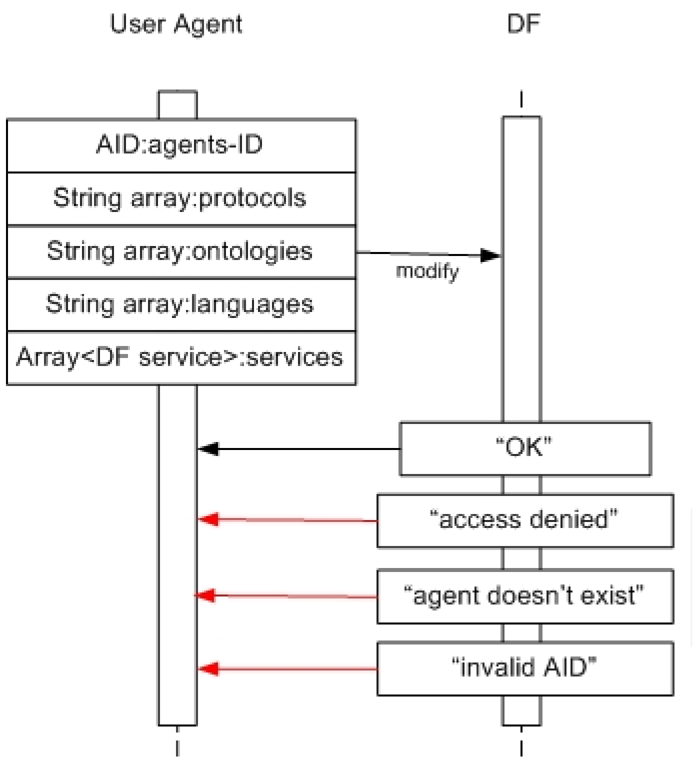


Figure F.11: Conversation Protocol: DF modify

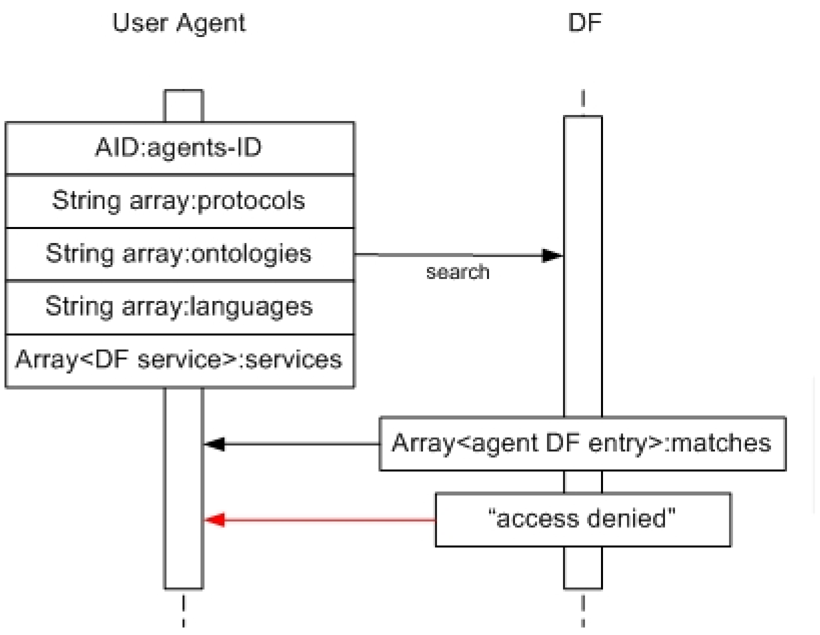


Figure F.12: Conversation Protocol: DF search

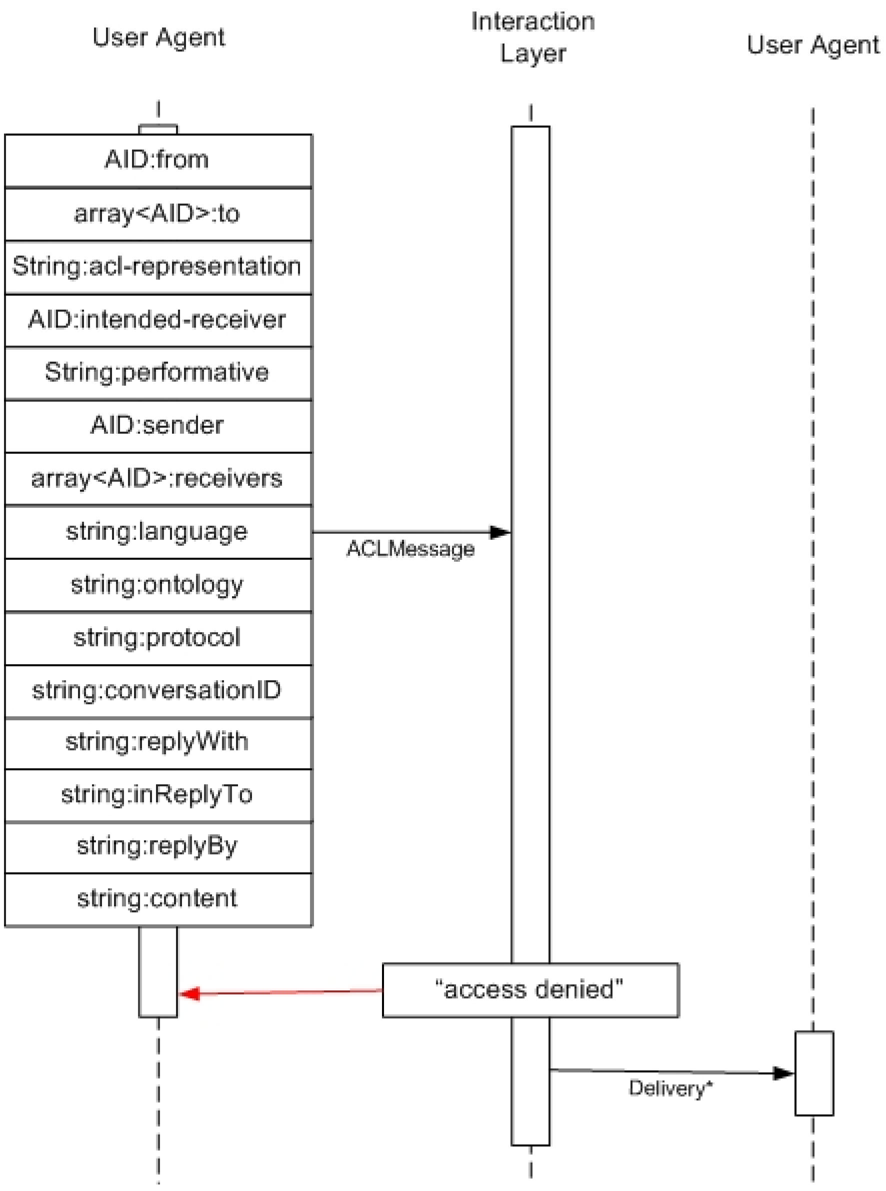


Figure F.13: Conversation Protocol: MTS send message