MASIC : A secure mobile agent framework for Internet computing

Submitted in part fulfilment of the requirements for the award of Doctor of Philosophy of the University of Surrey

Author : Nikolaos Antonopoulos

October 2000
This thesis is dedicated to my beloved family Panagiotis, Aikaterini, Hlias and Ioanna and my sweet wife Yu-lin.
ABSTRACT

Software mobile agents is a new distributed computing paradigm which was developed to support efficient computing over the Internet. Since its inception there has been a significant research effort to produce concrete agent-based artefacts. This phenomenon resulted in the proliferation of a large number of agent systems, mostly based on proprietary programming languages, each with its own characteristics, peculiarities and assumptions. Hence the agent technology has largely remained hidden and incomprehensible by Internet end users. Moreover the issue of interoperability and integration of agents with existing legacy software has only just started to be addressed by the agent research community in a rather ad hoc way. In this thesis we attempt to design an agent architecture which is independent of any programming language and therefore is directly suitable for Internet end users. The proposed architecture, labelled as Mobile Agent System for Internet Computing (MASIC), addresses several important contemporary issues in agent research. It defines an agent as a container of reusable components that can be copied or moved to other agents. Each agent has a symmetric I/O access control module and is also equipped with associative access collaboration facilities. Additionally every agent contains a navigator module which stores the agent's itinerary plan and provides an interface via which the agent itself or other authorised agents can dynamically adapt the plan to reflect run-time events and constraints. The agent system provides an integrated access control architecture which enables an agent to define customised access control structures that can be fully or partially shared with other agents. Existing access control structures can be combined to create new structures that represent more complex access mechanisms. Agents can discover other agents offering pertinent services via an adaptive, customisable agent discovery architecture incorporated in MASIC. This discovery architecture enables the full interaction of links with queries and supports the definition of access paths which are tightly coupled with access control and other customised services. MASIC also provides the conceptual architecture of a message-oriented agent communication system integrated with a mobility management scheme. Finally, this thesis presents the design and implementation of a prototype graphical interface which enables the potential user of the system to create, manage and interact with agents in real time. In conclusion the research presented in this thesis aims to provide a comprehensive, language-neutral, secure, collaborative environment within which mobile agents can interact with their peers in order to perform their tasks efficiently while human operators can oversee and manage these activities through a user friendly interface. The architecture is generic in nature as it can support general-purpose, agent-based computations. Its concepts, entities and mechanisms can be fully or partially re-used to provide architectural solutions to challenges in various application domains such as Knowledge Management, the GRID and E-Commerce.
ACKNOWLEDGEMENTS

I would like to thank a number of people for helping me go through this big adventure called PhD. First of all I would like to thank my project supervisor Prof. Alex Shafarenko for his guidance, good comments and assistance in producing the final version of this thesis. Prof. Khurshid Ahmad also gave me enormous moral support and helped me a lot with the proper structuring of the thesis. Special thanks to Mrs Caroline Jones for correcting a plethora of grammatical and spelling mistakes in the initial draft of the thesis.

At this point I would like to thank my dear wife Yu-lin and my parents for helping me through all the difficult moments and for supporting me in every possible way throughout my 5-year PhD studies.

Finally I would like to thank my colleague Mr George Aggelou for our fruitful collaboration and his contributions in our joint publications.
List of Publications


# Table of Contents

Abstract .......................................................................................................................... i  
Acknowledgements ......................................................................................................... ii  
List of Publications .......................................................................................................... iii  
Table of Contents ............................................................................................................ iv  
List of Figures ................................................................................................................... vi  
List of Acronyms .............................................................................................................. viii  

## Chapter 1 Introduction ........................................................................ 1  
1.1 Background ................................................................................................. 1  
1.2 Motivations .................................................................................................... 5  
1.3 Contributions .................................................................................................. 6  
1.4 Outline of the thesis ....................................................................................... 7  

## Chapter 2 Literature survey ................................................................. 9  
2.1 Definition of agents ...................................................................................... 9  
2.2 Agent taxonomy .......................................................................................... 12  
2.3 Applications of agents ............................................................................... 14  
2.4 Agent models ............................................................................................... 16  
2.5 Approaches for the analysis and design of agent-oriented systems ........... 19  
2.6 Basic modules of agent systems .................................................................. 23  
2.7 Mobile software agents issues ..................................................................... 24  
2.8 The early years of mobile agents ............................................................... 31  
2.8.1 Telescript ............................................................................................... 33  
2.8.2 Wave ..................................................................................................... 36  
2.8.3 Agent Tel ............................................................................................... 40  
2.8.4 Tacoma .................................................................................................. 43  
2.8.5 HTTP-based agents (FP-main) ............................................................. 45  
2.8.6 Evaluation of early agent systems ......................................................... 46  
2.9 Modern mobile agent systems .................................................................... 49  
2.9.1 An overview of Java ............................................................................ 49  
2.9.2 An overview of CORBA ....................................................................... 52  
2.9.3 Foundation for Intelligent Physical Agents (FIPA) ............................. 56  
2.9.4 A critical review of contemporary agent systems ............................... 58  
2.10 Conclusions ............................................................................................... 62  

## Chapter 3 The MASIC Agent Architecture ........................................ 64  
3.1 Introduction ................................................................................................ 64  
3.2 Conceptual Design of the Framework ..................................................... 65  
3.3 The Initial Abstract Computational Model .............................................. 71  
3.3.1 Program representation ....................................................................... 74  
3.3.2 Program cooperation ............................................................................ 77  
3.3.3 Evaluation of the abstract model ......................................................... 78  
3.4 The tokens .................................................................................................. 81  
3.5 The Information Store ............................................................................... 84  
3.6 The Matching Store ................................................................................... 87  
3.7 Agent Navigator .......................................................................................... 92  
3.7.1 Agent Navigator: An overview ........................................................... 93  
3.7.2 Plan Items ............................................................................................. 95  
3.7.3 The Split/Join procedure ..................................................................... 99  
3.7.4 Conclusions ........................................................................................ 100  
3.8 Discussion .................................................................................................... 101
### Chapter 4 Mobility management and agent discovery

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>104</td>
</tr>
<tr>
<td>4.2 Mobility management background</td>
<td>105</td>
</tr>
<tr>
<td>4.2.1 Overview of the proposed system</td>
<td>106</td>
</tr>
<tr>
<td>4.2.2 Functional Specification of arbiters</td>
<td>108</td>
</tr>
<tr>
<td>4.2.3 Access paths</td>
<td>109</td>
</tr>
<tr>
<td>4.2.4 The MARSA Management Agents</td>
<td>111</td>
</tr>
<tr>
<td>4.2.5 Evaluation</td>
<td>113</td>
</tr>
<tr>
<td>4.3 Introduction to agent organisation &amp; discovery</td>
<td>113</td>
</tr>
<tr>
<td>4.4 Agent organisation as name management</td>
<td>117</td>
</tr>
<tr>
<td>4.5 Requirements of a concrete agent discovery system</td>
<td>121</td>
</tr>
<tr>
<td>4.6 Overview of the proposed architecture</td>
<td>123</td>
</tr>
<tr>
<td>4.7 The links</td>
<td>126</td>
</tr>
<tr>
<td>4.8 Query and its interface</td>
<td>129</td>
</tr>
<tr>
<td>4.9 Access paths coupled with customised security</td>
<td>131</td>
</tr>
<tr>
<td>4.10 Operations on group managers</td>
<td>133</td>
</tr>
<tr>
<td>4.11 Top level view of a group manager</td>
<td>138</td>
</tr>
<tr>
<td>4.12 The name registry</td>
<td>140</td>
</tr>
<tr>
<td>4.13 Evaluation</td>
<td>142</td>
</tr>
<tr>
<td>4.14 Implementation issues</td>
<td>145</td>
</tr>
<tr>
<td>4.15 Conclusions</td>
<td>147</td>
</tr>
</tbody>
</table>

### Chapter 5 The MASIC Access Control Architecture

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction</td>
<td>148</td>
</tr>
<tr>
<td>5.2 Overview of abstract model</td>
<td>150</td>
</tr>
<tr>
<td>5.3 A generic authorisation scheme</td>
<td>152</td>
</tr>
<tr>
<td>5.4 A security example</td>
<td>154</td>
</tr>
<tr>
<td>5.5 The concrete access control model</td>
<td>156</td>
</tr>
<tr>
<td>5.6 Security features of cell</td>
<td>157</td>
</tr>
<tr>
<td>5.7 An object-oriented view of the architecture</td>
<td>162</td>
</tr>
<tr>
<td>5.8 Conclusions</td>
<td>163</td>
</tr>
</tbody>
</table>

### Chapter 6 Prototype client side

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Introduction</td>
<td>164</td>
</tr>
<tr>
<td>6.2 Design Methodology</td>
<td>164</td>
</tr>
<tr>
<td>6.3 GUI Design Requirements</td>
<td>166</td>
</tr>
<tr>
<td>6.4 GUI Design Process</td>
<td>166</td>
</tr>
<tr>
<td>6.5 Description of GUI</td>
<td>169</td>
</tr>
<tr>
<td>6.6 GUI Implementation</td>
<td>174</td>
</tr>
<tr>
<td>6.7 Limitations of the implementation</td>
<td>175</td>
</tr>
<tr>
<td>6.8 Discussion of source code</td>
<td>176</td>
</tr>
<tr>
<td>6.8.1 Authorisation check</td>
<td>177</td>
</tr>
<tr>
<td>6.8.2 Handling incoming tokens</td>
<td>177</td>
</tr>
<tr>
<td>6.8.3 The main window</td>
<td>179</td>
</tr>
<tr>
<td>6.8.4 CreateCellForm and CreateProgramForm classes</td>
<td>180</td>
</tr>
<tr>
<td>6.9 Conclusions</td>
<td>181</td>
</tr>
</tbody>
</table>

### Chapter 7 Conclusions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Summary</td>
<td>182</td>
</tr>
<tr>
<td>7.2 Evaluation</td>
<td>183</td>
</tr>
<tr>
<td>7.3 Future research</td>
<td>186</td>
</tr>
<tr>
<td>Bibliography</td>
<td>188</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Agent taxonomy according to Gilbert</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Agent taxonomy according to Nwana</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Müller taxonomy of agent models</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Conceptual model of a BDI agent</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>Conceptual model of reactive agents</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>Müller’s Interop architecture</td>
<td>19</td>
</tr>
<tr>
<td>2.7</td>
<td>Burmeister’s agent-oriented analysis</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>Blackboard-based agent communication</td>
<td>25</td>
</tr>
<tr>
<td>2.9</td>
<td>Franklin’s cooperation typology</td>
<td>27</td>
</tr>
<tr>
<td>2.10</td>
<td>Matchmaker operation</td>
<td>28</td>
</tr>
<tr>
<td>2.11</td>
<td>Broker operation</td>
<td>29</td>
</tr>
<tr>
<td>2.12</td>
<td>RPC based distributed computing</td>
<td>31</td>
</tr>
<tr>
<td>2.13</td>
<td>Remote programming</td>
<td>32</td>
</tr>
<tr>
<td>2.14</td>
<td>Telescript system</td>
<td>35</td>
</tr>
<tr>
<td>2.15</td>
<td>Agent-Tcl architecture</td>
<td>41</td>
</tr>
<tr>
<td>2.16</td>
<td>Java RMI architecture</td>
<td>51</td>
</tr>
<tr>
<td>2.17</td>
<td>CORBA architecture</td>
<td>53</td>
</tr>
<tr>
<td>2.18</td>
<td>OMA architecture</td>
<td>54</td>
</tr>
<tr>
<td>2.19</td>
<td>FIPA agent management reference model</td>
<td>57</td>
</tr>
<tr>
<td>3.1</td>
<td>Framework after step 1</td>
<td>66</td>
</tr>
<tr>
<td>3.2</td>
<td>Framework after step 2</td>
<td>67</td>
</tr>
<tr>
<td>3.3</td>
<td>Framework after step 3</td>
<td>68</td>
</tr>
<tr>
<td>3.4</td>
<td>Framework after step 4</td>
<td>69</td>
</tr>
<tr>
<td>3.5</td>
<td>The final conceptual model of the agent framework</td>
<td>71</td>
</tr>
<tr>
<td>3.6</td>
<td>A simplified view of a cell</td>
<td>73</td>
</tr>
<tr>
<td>3.7</td>
<td>Program token conceptual structure</td>
<td>75</td>
</tr>
<tr>
<td>3.8</td>
<td>Program representation</td>
<td>77</td>
</tr>
<tr>
<td>3.9</td>
<td>An XML token</td>
<td>82</td>
</tr>
<tr>
<td>3.10</td>
<td>The agent Navigator</td>
<td>93</td>
</tr>
<tr>
<td>3.11</td>
<td>The Plan Stack example</td>
<td>98</td>
</tr>
<tr>
<td>3.12</td>
<td>XML representation of the plan item</td>
<td>98</td>
</tr>
<tr>
<td>4.1</td>
<td>An addressing example</td>
<td>107</td>
</tr>
<tr>
<td>4.2</td>
<td>Layered Architecture of arbitrer</td>
<td>108</td>
</tr>
<tr>
<td>4.3</td>
<td>Current approach to agent discovery</td>
<td>115</td>
</tr>
<tr>
<td>4.4</td>
<td>The abstract name management model</td>
<td>119</td>
</tr>
<tr>
<td>4.5</td>
<td>The structure of a link</td>
<td>126</td>
</tr>
<tr>
<td>4.6</td>
<td>An XML link</td>
<td>128</td>
</tr>
<tr>
<td>4.7</td>
<td>The structure of a query</td>
<td>129</td>
</tr>
<tr>
<td>4.8</td>
<td>An XML query</td>
<td>129</td>
</tr>
<tr>
<td>4.9</td>
<td>An example of complex paths</td>
<td>137</td>
</tr>
<tr>
<td>4.10</td>
<td>Structure of a group manager</td>
<td>138</td>
</tr>
<tr>
<td>5.1</td>
<td>The abstract access control model</td>
<td>150</td>
</tr>
<tr>
<td>5.2</td>
<td>Overall security mechanism</td>
<td>153</td>
</tr>
<tr>
<td>5.3</td>
<td>Security protocol example</td>
<td>155</td>
</tr>
<tr>
<td>5.4</td>
<td>The concrete access control architecture</td>
<td>156</td>
</tr>
<tr>
<td>5.5</td>
<td>Data structure of the input access controller</td>
<td>159</td>
</tr>
<tr>
<td>5.6</td>
<td>An object-oriented model of the architecture</td>
<td>162</td>
</tr>
<tr>
<td>6.1</td>
<td>Prototyping method</td>
<td>165</td>
</tr>
</tbody>
</table>
Figure 6.2 An early version of the main window ............................................................... 167
Figure 6.3 The final layout of the interface ................................................................. 168
Figure 6.4 The main window of the interface ............................................................ 169
Figure 6.5 The data cell creation form ................................................................. 171
Figure 6.6 Connected Cells List ............................................................................. 171
Figure 6.7 Send Token form .................................................................................. 172
Figure 6.8 Sending a command to a connected cell ............................................. 173
Figure 6.9 Answer appears on the main screen .................................................. 173
Figure 6.10 Creation of connection and handling of the incoming message ......... 178
List of Acronyms

ACL: Access Control List
API: Application Programmer’s Interface
ASN: Abstract Syntax Notation
BDI: Belief Desire Intention model for intelligent agents
BPEM: Binary Predicate Execution Model
CORBA: Common Object Request Broker Architecture
DCOM: Distributed Common Object Model
DOM: Document Object Model
DTD: Document Type Definition
FIPA: Foundation for Intelligent Physical Agents
GUI: Graphical User Interface
GUID: Global Unique Identifier
HTTP: HyperText Transfer Protocol
IDL: Interface Definition Language
IP: Internet Protocol
KIF: Knowledge Interchange Format
KQML: Knowledge Query and Manipulation Language
MARSA: Mobile Agents Routing Services Architecture
MASIC: Mobile Agent System for Internet Computing
MASIF: Mobile Agent System Interoperability Facility
MOM: Message Oriented Middleware
OMA: Object Management Architecture
OMG: Object Management Group
PGP: Pretty Good Privacy
PKI : Public Key Infrastructure
RMI : Remote Method Invocation
RPC : Remote Procedure Call
TCP : Transmission Control Protocol
XML : eXtensible Markup Language
Chapter 1

Introduction

1.1 Background

The information explosion and overload phenomena that occurred during the past few decades signalled the start for the design and development of the largest computer communications infrastructure system, the Internet. The Internet can be thought of as a vast, physically distributed set of interconnected resources (software and hardware). In the early years (early 1980's) the Internet was small and restricted into interconnecting a number of academic institutions across the world. At that time the number of hosts connected to the Internet was in the order of a few hundred. With the invention of the TCP/IP the Internet started growing exponentially and by 1999 the number of Internet hosts was more than 57 million. This incredible growth is accredited to the fact that not only did computers start becoming cheaper and more powerful but the need for computer communications became more urgent because of the fast growing desire for information sharing and exchange. Furthermore it was realised that the Internet could provide basic communications facilities on top of which any network (distributed) application could be implemented. This realisation changed the perception of the Internet from a simple computer communications infrastructure to a medium that supports distributed processing on a global scale (Internet Computing). Nowadays, the Internet offers great business opportunities and is open to commercial exploitation, facilitating the very rapid growth of Internet computing during the last decade.
Until the beginning of the 90's, the predominant computational paradigm on which Internet computing was based was the client-server. According to this approach a dedicated machine, the server, contains a number of programs that can be remotely invoked via Remote Procedure Calls (RPC) by programs running on other machines, the clients. The central mechanism of this paradigm, the RPC, was designed with the aim of providing transparency for remote procedure calls such that the invoking program does not observe any difference in both the invocation mechanism and the performance between local or remote procedures. In reality this aim was never achieved because of the unreliability and bottlenecks of the underlying communication infrastructure. Moreover RPC was inherently inefficient for two reasons. Firstly clients were forced to use only the existing server functionality which could not be adapted to suit the needs of each individual client. Secondly ongoing interaction between the client and the server required ongoing communication. The latter characteristic becomes a serious drawback in the context of Internet computing where the cost of local processing is negligible compared to the cost of communication. Efficient Internet computing should be based on the aforementioned principle and therefore minimise the need for communication while perhaps paying the small price of extra local processing.

The RPC's inherent inefficiencies shifted the focus of research into developing new computational paradigms that better match the Internet's characteristics. The result was the notion of programs that can migrate from one physical machine to another during their execution. Such programs are called software mobile agents and offer the strategic advantage of minimised communication over the traditional RPC. In reality the concept of agency has existed for several decades within the artificial intelligence community. However it was only in 1994 when the distributed computing research community realised that the mobility of programs can be a serious candidate in minimising communication. The same year also witnessed the birth of the first commercial, general-purpose, agent-based system: Telescript. This development signalled the start of a massive research and development frenzy in the area of mobile agents and agent-based systems. This process resembles the gold rush experienced in the United States during the 18th century. Individual researchers, research groups, institutions together with major commercial companies grabbed this research opportunity and got involved in the production of more and more agent models, languages and systems.
The result as expected has been rather chaotic. Each researcher/group of researchers viewed the software agents from a different perspective according to which agents were given different labels such as intelligent, autonomous, mobile, collaborative, mobile objects, personal assistants, robots, etc. Each agent system was based on its own assumptions and sometimes it was oriented towards specific applications. Most of these agent systems were strongly coupled with a proprietary programming language frequently with awkward semantics thus severely restraining their usability and applicability. Moreover various aspects of agent-based computing such as security and agent organisation were neglected as these systems’ main objective was to support code mobility.

In general there are four main characteristics of this agent research effort. Firstly the existing agent scenery does not exhibit any interoperability. The development of the Java programming language and its adoption as the de facto standard for distributed computing triggered the development of Java-based agent models and systems which could theoretically interoperate with each other. Such interoperation has not been realised until then because the developers of each agent system were competing against each other as to who will first produce the best, most complete agent system which will render the rest of the agent systems obsolete. Secondly most of the proposed solutions for issues such as security, discovery, communication, etc. are already existing approaches in other domains of the computer science which have been just re-labelled as agent solutions. Thirdly the current agent systems are not integrated at all to the existing software scene since most of them provide no portals to existing legacy software. Fourthly standardisation is clearly missing from the area of software agents. It was only in 1997 that the first international body FIPA (Foundation for Intelligent Physical Agents) was formed to promote standardisation of several agent-based techniques, models and architectures. A similar effort started earlier in the area of distributed objects and has resulted in a set of specifications collectively labelled as CORBA (Common Object Request Broker Architecture) that nowadays every distributed object system developer is eager to comply with. The reason for the wide adoption of CORBA, which comprises more than 750 organisations, as the standard in distributed objects is that this standardisation effort was embraced from the beginning by all prominent academic institutions and companies. In contrast FIPA has unfortunately remained largely in the background of agent research and as a result its agent specifications have had very little impact in the agent world until now.
The immaturity of the agent world clearly shows if we observe that the agent research community is more influenced by the developments in Java which is essentially a distributed object programming language rather than the work of its own standardisation body (FIPA).

All the aforementioned characteristics of the agent revolution indicate that this field is still in its infancy. A lot of ground work is needed and more successful real world agent applications are required before this field becomes stable and mature. Ultimately, like any other software, the software agent's success will depend on whether this technology reaches the big mass of Internet users or whether it will stay within the boundaries of research laboratories as yet another one proprietary intellectual artefact.

There are some necessary conditions that agent software must satisfy in order to maximise its social impact and thus its usefulness. Firstly agent-based technologies must retain a good level of adoption. This in turn depends on addressing issues such as integration, interoperability and portability. Moreover their design must take into account the end user and thus it should empower the user to monitor, update and manage this technology (usability). Secondly the agent technology must ensure that it clearly exhibits strategic advantages over other software technologies. Programming efficiency is the keyword here. A user or a business will definitely consider deploying agent technologies within their infrastructure only if there are some clear efficiency gains for doing so. The issue of programming efficiency should be embedded in both the data representation and the program representation and execution. The object-oriented technology quickly became a success because of the software reuse mechanism they incorporate (inheritance). In conclusion agent technologies can persuade of their usefulness if they show their adoptability, programming efficiency and usability.

One approach to address the aforementioned challenges in the area of software agent research is to design and develop language-neutral frameworks that provide an environment within which agents can execute, interact and perform their tasks efficiently. Although the agents can act as completely autonomous entities, the agent research literature indicates strongly that there are a number of services that need to be offered to agents on a group-basis because of their nature/importance. Examples of such services include security, agent organisation/discovery and mobility management. Agent frameworks offer a natural context within which these services can be offered.
1.2 Motivations

The main aim of this thesis is to design a framework for general-purpose Internet computing based on autonomous, mobile, software agents.

Firstly the framework as a whole should satisfy the conditions mentioned in the previous section. Namely it should integrate well with existing legacy software which is often written in a plethora of programming languages. It should also possess a high degree of usability by enabling the end-user to manage and use the framework. It should incorporate the concept of software re-use as a means of efficient data representation. Finally it should also provide program representation and execution schemes.

Secondly the framework should support several important services that the agents can use in order to execute their tasks. Four services have been identified as being necessary to facilitate the harmonic and efficient operation of the individual agents as well as the safe collaboration and interaction among them. Specifically the services that the proposed framework must offer are as follows:

- Agent collaboration and co-ordination. Agents frequently need to collaborate with each other in order to accomplish a given task. The mobile agent paradigm provides a good foundation for decomposing a task into a number of concurrently/parallel executing agents thus reducing the overall time required to complete the task compared to traditional monolithic applications. However the more sub-tasks a task is divided into the more complex and time consuming becomes the problem of synchronising and co-ordinating the individual sub-tasks, especially when there are strong interdependencies among them. The agent framework therefore must provide the agents with simple, efficient collaborative facilities that minimise the overhead of intertask co-ordination.

- Agent organisation and discovery. Single-agent frameworks do not need this service. However as the number of user-defined agents as well as service-offering agents increase, the issue of how to find agents that possess certain expertise becomes more important. Like a human society, an agent society (group of agents) must be organised in such a way that individual agents can discover efficiently other pertinent agents.

- Access control. Any computation roaming across the Internet poses serious security hazards to its recipient since the underlying communication network is untrusted.
Security must therefore be treated as being of at least the same importance as the computational model of the framework. The security mechanism must be flexible, allow for user customisation and keep the processing overhead to a minimum.

- Mobility management. Mobility is itself the characteristic that gives the mobile agents paradigm a strategic advantage over other approaches. Unfortunately, though, the existence of mobility creates a series of other problems such as how to ensure continuous access to a mobile entity (agent), etc.

The aforementioned points together with the general requirements identified in the previous section form the set of objectives for this thesis.

### 1.3 Contributions

The contributions of the work presented in this thesis are:

- Design of the conceptual model of an architecture-based (thus language-neutral) agent framework that offers the aforementioned four services.

- Design of an individual agent model that possesses symmetric access control (input and output), contains onboard collaborative facilities, represents the agent’s state as a set of re-usable, movable, components expressed in XML and enables the efficient representation of flexible, dynamic navigational plans of the agent.

- Design of a general purpose agent organisation and discovery mechanism. The mechanism is adaptive through the peer-to-peer interaction of the queries and the organisation nodes. Moreover existing organisation nodes can be used to create new and more complex ones (organisation knowledge re-use).

- Design of an integrated access control mechanism that supports the distribution and re-use of security knowledge can be customised on a per agent basis and enables the agents to take more knowledgeable access control decisions. The proposed mechanism is also well integrated with the agent organisation and discovery facility of the framework.

- Provision of a number of different types of agents to enable the user to utilise any mixture of these in order to represent its task (program) more efficiently.
• Design of the conceptual architecture of an integrated message-oriented and mobility management communication component. This component enables inter-agent communication, guarantees continuous access to the moving agents and allows for run-time customisations/modifications such as the creation/deletion of message-passing nodes, the addition of new participating system servers, etc.

• Design and prototype implementation of a graphical interface via which the users can create/delete/interact with existing agents. The user-agent interaction is mutual because agents can also send messages to users which are then displayed directly on the graphical interface.

1.4 Outline of the thesis

The remainder of the thesis is organised as follows:

Chapter 2 contains the literature review. The review is divided in two parts. The first part gives a primer on mobile agents covering the definition, characteristics and applications of mobile agents. The second part presents and analyses the two predominant approaches in designing mobile agent frameworks: the language and the architecture approach. The general characteristics of each approach are identified and criticised. Each of the approaches is then compared against the objectives of this thesis. The results of this comparison are used in following chapters to justify certain design decisions.

Chapter 3 presents the conceptual model of the proposed framework which consists of an agent facilitator (cell) and the agent system. An abstract model of the agent facilitator is then presented and it is used to build a more concrete model of the facilitator. The design and functionality of the facilitator’s components, namely, the token store, the information store and the agent navigator, is described in detail.

Chapter 4 presents the models of the mobility management, communication and organisation layers of the agent system. The first part of the chapter begins with an overview of the conceptual model of the integrated mobility management and communication layer of the agent system. Next it presents the supported addressing schemes for mobile agents, which is followed by a brief description of the contents and functionality of the main entities of the model. The second part of the chapter starts with a
general overview and criticism of the current approaches in the area of information and resource discovery. It then proceeds by presenting an abstract model for the organisation of agents. Similar to chapter 3, this abstract model is then used as a basis to build and describe the design of a concrete model of the organisation and discovery layer of the agent system. The chapter presents in detail the architecture of the organisation nodes (group managers), the internal structure of the links (the building blocks of the group managers), the structure of the queries (discovery messages) as well as the query-link interaction. It then continues to present how existing group managers can be used to build new ones (composite group managers). Throughout the chapter there is an analysis of the consistency mechanisms the described model provides to ensure the integrity of the organisational structure.

Chapter 5 presents the access control model of the framework. The presented model contains two layers. The first layer is the access control facilities embedded within the cell. The second layer is the access control facilities (lock cells) the agent system provides to support security knowledge sharing and re-use. Like chapters 3 and 4 this chapter begins with the presentation and analysis of an abstract model for access control based on the idea of locks and keys. It then describes the semantics of the locks and keys as well as the operations that can be performed on them. The chapter continues with the presentation of the concrete access control model including the description of the contents and functionality of lock cells. Finally it is shown how lock cells can be combined to produce composite lock cells in a similar fashion to the process of creating composite group managers from existing ones in chapter 4.

Chapter 6 presents the design and prototype implementation of a GUI via which potential users could interact and manage an implementation of the framework. The first part of the chapter shows and justifies the main components of the GUI. The second part outlines its prototype implementation in Java 1.1.

Finally chapter 7 has the conclusions of the thesis and briefly describes possible avenues for future research.
Chapter 2

Literature survey

2.1 Definition of agents

Humans have always been fascinated with the idea of anthropomorphic artificial beings such as robots, cyborgs, androids, etc. The word robot, derived from the Czech word for drudgery has been used extensively to describe a whole range of intelligent mechanical machines from factory workers to digital butlers. The public image of robots has two sides. On the one hand there is profuse excitement about the new quality of life that these artificially intelligent beings will bring into the human society. On the other hand there is serious scepticism regarding the building and deployment of highly intelligent machines at a massive scale. This scepticism stems from the fact that the more intelligent robots become the more doubtful it becomes that these robots will obey their human masters. Although no-one can be absolutely sure, there is a danger that the more humanlike the robots become, the more likely it is that the robots could exhibit certain unwanted or dangerous human behavioural patterns such as frailties and eccentricities.

For similar reasons this scepticism has expanded in the area of software where the design and activation of powerful software robotic entities could pose serious threats especially in a human society whose main functions become rapidly fully dependent on computer software. These concerns must be taken into account in the design of the software agents.

Mechanical automata of various kinds and functionality have existed for a long time. However the notion of autonomous agents have recently begun to appear, concurrently with advances in the areas of control theory, computer science and engineering. Norman [92] notes that probably “the most relevant predecessors to today’s intelligent agents are servomechanisms and other control devices, including factory control”. The focus of the
research community has shifted from hardware to software during the past few decades (Negroponte [90]). According to Kay [65] the idea of a software agent actually originated with John McCarthy in the mid 1950’s who had a view of an autonomous system that would carry out the necessary computer operations in order to perform a user-defined task and interact with the user only to request some advice. In this context an agent would effectively be a software robot that lives and operates within a computer environment.

Agent research is classified by Nwana [93], who identifies two main streams of work. The first originates in the late 1970’s and has its roots in the area of Distributed Artificial Intelligence (DAI). This research stream focuses on agent macro issues such as cooperation, communication and inter-task coordination. The second stream is relatively recent. It started in the early 1990’s and focuses on the study of a broader range of agent types.

The output of these two agent research streams has resulted in the proliferation of a wide variety of agent models, systems and applications. This in turn resulted in an explosion in the definition and use of the term agent. Nowadays there is a very long list of agent definitions given by different researchers in different areas. This list represents the confusion and possibly conflicting views of the researchers as to what an agent is.

According to Bradshaw [24] this list is the product of two separate approaches to the definition of agents. The first approach is based on “the notion of agenthood as an ascription made by some person”. The second is based on “a description of the attributes that software agents are designed to possess”.

The ascription-based definition of agents proclaims that the notion of agency cannot be characterised as an attribute listing but as an attribution on the part of some person (van de Welde [118]). It is effectively the expectations and viewpoint of an individual that determine what an agent is. These different viewpoints can naturally be conflicting, for example one person’s “intelligent agent” is someone else’s “mobile object”. Dennett [35] observes that perhaps agents are a notion that developers use to define the behaviour of their software in the same way object-orientation is used to express certain algorithms more efficiently. The Collins New English Dictionary defines an agent as “one that acts or has the authority to act on behalf of or represent another”. The term agent is derived from the Latin verb agere: to act or do. Therefore, we can argue that a software agent is essentially a computer program which acts on behalf of a user and carries out a specific task that has been delegated to it. Dennett goes further and describes three predictive
stances that people can take towards systems in general: the physical, design and intentional stances. Singh [114] utilises Dennett’s classification and claims that there are pragmatic and technical reasons for viewing agents as intentional systems.

The description-based definition of agents seems to be more attractive to agent researchers. Shoham [113] defines an agent as a continuous, autonomous, software entity that operates within an environment populated with other agents. There are three keywords in this definition: continuous, autonomous and agent environment. Continuity is a desirable characteristic of an agent because it is a necessary factor for embedding into the agents learning capabilities. Autonomy is also required because ideally agents should be capable of carrying out their tasks without direct, continuous assistance from their human users. The existence of a multi-agent environment (society) implies that individual agents are expected to communicate and interact with each other in order to achieve their targets perhaps more efficiently. Nwana [93, 94] claims that a multi-agent society is more powerful than an individual agent. He says that agents as individuals might have very restricted functionality but as a society can be powerful enough to simulate a “digital sister-in-law” as defined by Negroponte [90]. According to the requirements of the problem they are assigned to, agents can possess several attributes to a certain degree. Several key agent researchers have attempted to provide a comprehensive list of such attributes. Etzioni and Weld [40] and Franklin and Graesser [42] propose among others the following key characterising attributes for software agents:

- Reactivity. The ability of an agent to sense its environment and react accordingly.
- Autonomy. An agent can also be pro-active and initiate certain actions in order to achieve a specific goal (goal-directed).
- Mobility. Agents can physically migrate from one host to another in order to perform their tasks more efficiently.
- Adaptivity. Agents can learn from their experience and adapt to a changing environment.
- Collaborative behaviour. Agents can collaborate with each other in order to achieve a common goal.

In a similar manner, Marc Belgrave [17] in his Unified Agent Architecture attempts to categorise such attributes into main and auxiliary.
2.2 Agent taxonomy

A simpler way of characterising all possible types of agents than would result if we attempted to describe all possible combinations of all agent attributes, is to attempt to provide various taxonomies of agents [29]. Moulin and Chaib-draa [86] classify agents into reactive, intentional and social categories. A reactive agent is the simplest type of agent since it only reacts to changes in its environment. An intentional agent is more sophisticated and possesses intentions and beliefs based on which it can reason and create different plans of action. A social agent builds on the intentional agent and contains explicit models of other agents which it can use to interact with them. Gilbert et al [50] (figure 2.1) classifies intelligent agents into a three dimensional space of agency, intelligence and mobility. Below a brief explanation of these dimensions is provided:

![Figure 2.1 Agent taxonomy according to Gilbert](image)

- **Agency.** The agency represents the degree of freedom an agent has. At the minimum level, an agent should be able to operate asynchronously without interacting or being guided by the user-owner. At the other end, an agent should be able to interact with other agents belonging to other user-owners, in order to accomplish the task in the most beneficial way for the owner.

- **Intelligence.** The intelligence represents the ability of an agent to learn and possibly to reason as well. At the first level of intelligence, an agent should only be able to follow a predefined set of rules. The next level is to be able to follow a user-model which is not so strict and instead of specifying a set of rules for accomplishing a task,
it provides a plan that the agent must follow. Finally, at the last level, an agent should be able to learn from its behaviour and adapt to its environment. Such an agent could make decisions on its own, reason about them and eventually produce information of higher value to its owner.

- **Mobility.** Mobility, as the word implies, represents the ability of an agent to move among locations in the network while working to accomplish its task. Mobility increases the efficiency of an agent since it can visit remote sites and use remote resources which otherwise would have been unavailable or very costly in terms of communication bandwidth. On the other hand though, mobile agents are a security risk since in the real world agents can be generated to serve malicious purposes. Agents can be static, or simple mobile scripts created on one machine and transported to another, or mobile objects in which case they are transported from machine to machine in the middle of their execution carrying accumulated state data with them.

Nwana [93] proposes a different classification scheme (figure 2.2). According to this taxonomy agents are characterised by mobility, a symbolic reasoning model, ideal and primary attributes, roles, hybrid philosophies and secondary attributes. The Nwana taxonomy effectively creates seven agent categories: collaborative agents; mobile agents; reactive agents; hybrid agents; smart agents; information agents; and interface agents.

![Figure 2.2 Agent taxonomy according to Nwana](image)
Franklin and Graesser [42] provide a hierarchical classification of agents whose root element is the type of autonomous agents. Petrie [102] argues for a specific class of agents: the typed-message agents. Typed-message agents are different from other software because they can communicate with each other using a shared message protocol. An example of such a protocol is the Knowledge Query and Manipulation Language (KQML) [41]. He also specifies that the shared communication protocol must be application-independent and peer-to-peer.

In conclusion according to Bradshaw [24] the term agent, like many other computing terms, began with a metaphor but will end up representing concrete software artifacts. As time goes by the public exposure of agent-based software and technology will either eventually produce a globally agreed semantic of the term agent or will increase today's confusion around this term even more. However, regardless of the success or failure of the term agent to denote something globally understood, it is certain that the motivations and advantages for building agent-based software will persist over time [24].

Indeed agent-based distributed computing offers several advantages over traditional approaches. Lange [75] describes seven important benefits of agents: reduction of network load because of their mobility; overcoming of network latency because agents can operate at the location of the data; encapsulation of protocols; asynchronous and autonomous execution; adaptivity; heterogeneity and robustness; and fault-tolerance. Furthermore according to Bradshaw [24] agents can provide intelligent interoperability in software systems and help overcome the limitations of direct manipulation interfaces.

### 2.3 Applications of agents

The agent model is very attractive and suits well the design of Internet-based applications. Areas in which agents have been applied widely include: telecommunications, interfaces for PDA systems, information retrieval, etc. [122]

Information retrieval is an excellent application area for agents since agents act on behalf of the user and can automate routine tasks. Some of the problems in this area, which agents have been used to solve are:

- Automation of information filtering and retrieval tasks.
- Automation of navigation and processing in heterogeneous networks.
- Modelling the user's requirements and preferences for information.
- Listing of available resources in a network.
- User-friendly access to resources.

Apart from information retrieval agents, at this moment there are hundreds of different agents in the Internet. Some of the most common types of agents are the following:

1. **Advisory agents.** Advisory agents do not carry out tasks. Instead, they offer advice to facilitate users' work. Initially such an agent does not know the work patterns and the preferences of its user. But gradually it starts learning and builds a user model which is persistent, in other words it remains associated with a user between different sessions [59]. A characteristic example of such an agent is the Coach, developed by Ted Selker at IBM, which tutors the Lisp programming language.

2. **Assistant agents.** Assistant agents are agents which can actually carry out tasks for the users. They are more ambitious than the advisory ones since they can often operate without feedback by the user and they possess a higher level of artificial intelligence. Because of this fact, they are a controversial topic which is still under debate. Assistant agents are more independent than the advisory ones but this implies that the user-control on such an agent is lower. Trust and competence are equally important attributes that such an agent must possess in order to be maximally beneficial for its user.

3. **Watcher agents.** A watcher agent operates autonomously, looking for specific information. When information relevant to the user is found or when an event the user is interested in has occurred, the user is either directly notified (for example by Email) or the information is stored for future access. A characteristic example of this type of agent is the MIT's Fishwrap newspaper which collects stories and articles from different sources and then produces a personalised version of the newspaper.

4. **Shopping agents.** Shopping agents are doing comparison shopping for their users and finding the best price for an item. The best known shopping agent on the Internet is the Bargain Finder from Andersen Consulting. This agent does comparison shopping for CDs and finds the best price of the CD specified by its user. Retailers can protect themselves from this competition by actually denying access to a shopping agent.

There are various end-user taxonomies of agents and agent applications. [88] classifies the agents into desktop, Internet and Intranet agents according to three factors. Firstly the environment the agents operate within. Secondly the type of task they are
designed to perform and thirdly their internal knowledge architecture. The three aforementioned agent types are decomposed into further sub-categories. The desktop agent type includes operating system agents, application agents, etc. The Internet agents include Web search, information retrieval and filtering, notification and mobile agents [69]. The Intranet agents include database agents, resource brokering agents.

2.4 Agent models

A wide range of agent models exists. Müller [87] classifies these models into deliberative, reactive and hybrid. Figure 2.3 shows a table of agent architectures classified according to the Müller taxonomy.

<table>
<thead>
<tr>
<th>Existing System Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Deliberative Agents</strong></td>
</tr>
<tr>
<td>IRMA (Bratman et al)</td>
</tr>
<tr>
<td>BDI (Rao, Georgeff)</td>
</tr>
<tr>
<td>(Shoham)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Reactive Agents</strong></td>
</tr>
<tr>
<td>Subsumption Architecture</td>
</tr>
<tr>
<td>(Brooks)</td>
</tr>
<tr>
<td>Pong (Agricola, Chapman)</td>
</tr>
<tr>
<td>Dynamic Action Selection</td>
</tr>
<tr>
<td>(Maes)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Hybrid Agents</strong></td>
</tr>
<tr>
<td>RAP (Firby)</td>
</tr>
<tr>
<td>AIS (Hayes-Roth, Dubiaji)</td>
</tr>
<tr>
<td>Sim. Agent (Sloman, Poli)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Figure 2.3 Müller taxonomy of agent models

Deliberative agents possess an explicit model of their environment which they utilise as a basis for logical reasoning. The environment model these agents possess is typically static, predefined and very complex. Deliberative agents are not suitable for use in dynamic environments because their environment model is rigid and cannot be updated dynamically to reflect any changes that occur at run time. Probably the most important characteristic of deliberative agents is their ability to reason. During the decision-making process, the agent utilises the knowledge contained in its model to modify its internal, or often called mental, state.

Rao/Georgeff [104] argued that this mental state consists of three base components: beliefs, desires, intentions which resulted in giving the deliberative agents the alias BDI.
Müller [87] extended the classical BDI concept with two new components: goals and plans. Below is a brief description of the semantics of these five elements of the deliberative agents' mental state:

- **Beliefs.** These are the views the agent has of its environment. These views are used to express its expectations of possible future states.
- **Desires.** These contain the agent's judgements of future situations. The desires of an agent can be unrealistic or even contradict each other.
- **Goals.** These are a subset of the agent's desires that the agent could fulfil in the future. Thus in contrast to the rest of its desires, the agent's goals must be realistic and not contradictory. Essentially the goals define the agent's potential future actions.
- **Intentions.** These are a subset of the agent's goals. When an agent decides to follow a specific goal then this goal becomes an intention.
- **Plans.** The plans combine the agent's intentions into consistent units [25].

The BDI agent model is a typical representative of deliberative agents. Figure 2.4 shows the conceptual architecture of a typical deliberative agent.

![Figure 2.4 Conceptual model of a BDI agent](image)

The rigid structure and complexity of deliberative agents have raised a number of serious concerns which lead eventually to a diametrically opposite concept, the so-called reactive agents. Reactive agents do not have any internal model of their environment and they do not have the ability to perform complex logical reasoning. Brenner [25] argues that "the reason for these restrictions lies in the creation of compact, error-tolerant and flexible agents". Reactive agents obtain the necessary knowledge from their interaction with their environment. Therefore the intelligence a reactive agent exhibits is directly related to the level and amount of interaction the agent has with its environment. Figure
2.5 shows a simplified architecture of a reactive agent. Sensors record information from the environment which is subsequently forwarded to a number of task-specific competence modules which in turn produce reactions that are transferred to the environment via the agent's actuators. A typical example of a reactive agent is the Brooks subsumption architecture (Brooks [26]).

![Figure 2.5 Conceptual model of reactive agents](image)

The reactive agents have a number of advantages over the deliberative agents including no centralised structures such as planners which results in better fault-tolerance and robustness, flexibility and simplicity. However reactive agents do have certain limitations in comparison to the deliberative agents such as no capability for planning. This results in the creation of a third agent model that attempts to integrate the reactive and deliberative agent models in order to harness the advantages of both approaches. Such agents are called hybrid agents. The Interrap architecture (figure 2.6) from Müller [87] is probably the most famous agent architecture that falls into the category of hybrid agents.

A typical hybrid agent contains a reactive and a deliberative module. The reactive module is used primarily for interaction with the environment whereas the deliberative module is used to provide the services of logical reasoning and planning. The hybrid agents are normally designed as a hierarchical architecture. The lower levels of this hierarchy are reactive modules which are used to sense and acquire information from the environment. The upper levels are occupied by deliberative modules which typically perform goal determination and planning.
2.5 Approaches for the analysis and design of agent-oriented systems

Like any other software, the development of agent-based software goes through the phases of analysis, design and implementation. In this section we will present different methodologies for the analysis and development of agent systems.

Burmeister [27] specifies three models for the analysis phase of agent-oriented systems (figure 2.7): the agent model, the organisation model and the cooperation model.

The agent model specifies the internal structure of the agents. This structure defines the internal attributes and methods (in the object-oriented context) as well as the behavioural patterns and intentions of the agents. The organisation model specifies the static relationships between agents and agents' categories. Such relationships can represent inheritance, aggregation or different roles that the agents assume within their environment. The cooperation model defines cooperation and interaction among the agents. Messages form the basis for the communication and cooperation processes. The model must specify the cooperation objectives, the message types and message exchange sequences (protocols).
Kinny [66] developed a similar methodology to Burmeister. However whereas Burmeister's approach focuses more on the agent interaction and cooperation, Kinny assigns more importance to the agent's internal structure. Kinny's methodology introduces two abstractions, the internal and external view, and consequently provides both internal and external models. Kinny separates the agent's architecture from that of the agent system. The internal model specifies each individual agent's architecture (i.e. attributes, knowledge, methods, etc.) whereas the external model defines the agent's responsibilities and interaction with other objects. Kinny identifies two external models: the agent model and the interaction model. The agent model describes the static relationships between agents in the form of agent hierarchies. Essentially it has the same objectives with Burmeister's organisation model. The interaction model represents the responsibilities, communication, services and cooperation among agents. This model is very similar to Burmeister's cooperation model. In a similar manner Kinny defines three internal models: the belief model, the goal model and the plan model. These models can be used for the analysis of agent systems whose basic entities are BDI agents.

There are three predominant approaches for agent-oriented design: the facilitator and agent communication language approach; the language based approach and the architecture based approach.

Genesereth [49] provides a comprehensive methodology for the design of agent-oriented systems which is based on Kinny's analysis model. In this approach agents are
defined as entities that can communicate with each other by using a specific agent communication language such as KQML. Based on this definition he argues that there are three ways of designing an agent (or an internal agent model in Kinny's terminology): rewriting; using wrappers or using transducers. An agent can be written from scratch so as to be able to use directly the standard agent communication language. Another way is to take an existing program and wrap it with specialised code so that the resulting code is capable of using the agent communication language. Alternatively there can be specialised software called transducers that mediate the communication between traditional programs and agents. The traditional programs are separate entities that communicate with transducers with a proprietary protocol. The transducers act as gateways and transform the proprietary messages from the traditional software into messages that satisfy the rules of the agent communication language. Genesereth also specifies that the agent system (external model in Kinny’s terminology) is either direct or federated. In a direct agent system the agents communicate with each other directly. Genesereth focuses though on federated systems. A federated agent system consists of a set of entities called facilitators. Agents do not communicate directly with each other but only through facilitators which act as message routers. Facilitators could also be assigned higher level functionality to make them capable of providing security, content routing, agent discovery mechanisms, etc.

In the language-based approach both the agents and the agent system are built upon a specific programming language’s characteristics, functionality and run-time environment. In the Genesereth sense agents are rewritten in a specific language and the agent system is federated in which the specific language’s run-time environment entities act as facilitators. Clearly in this approach the most important design decision that the developer has to take is the selection of the programming language that will form the basis of the design. Several researchers have attempted to identify the criteria that a programming language has to satisfy in order to be suitable for the development of agents and agent systems. Hohl [56] and Knabe [67] specified the following:

- **Object-oriented.** Agents can be considered as objects whose internal state (data) can only be accessed and manipulated through the invocation of a method. Each agent has a public interface which comprises a set of publicly available methods.
- **Platform independence.** The agent programming language must possess a high degree of platform independence because agents should be capable of operating within
different hardware and software platforms. The Internet, arguably one of the best application areas of agents, is a vast heterogeneous distributed environment.

- Communications capability. The language must provide sufficient and efficient mechanisms for implementing inter-agent communication.
- Security. The agent language must provide a very high degree of security for the execution of agents. Security after all is one of the primary concerns of agent sceptics.

Together with these requirements an agent programming language must also provide a range of additional features such as persistence and multitasking. The languages agent researchers have used until now to develop agent-oriented systems fall broadly speaking into three categories. The first category is object-oriented languages that are widely available such as Java [51]. The second category is proprietary languages such as Telescript [45, 48]. The third category is scripting languages which are either widely available such as Tcl [16] and Python [3], or proprietary such as WAVE [108, 109] and Messengers [20]. Nwana and Wooldridge [94] provide a comprehensive overview and classification of agent programming languages.

The architecture-based agent design approach provides a different perspective. Language-based agent systems are bound to have limited end-user popularity and integration with existing software because: a) they require that the user has a good knowledge of the language they are based on; and b) existing software written in other languages cannot easily be converted and interoperate with the language-based agents. The architecture-based approach attempts to resolve these limitations by separating the programming languages (implementation) from the internal structure of both the agent and the agent system. In the Genesereth sense the architecture-based approach defines an agent as a number of existing software components together with a transducer which insulates these components from the agent environment. The agent system is a federated system where specialised software entities provide a variety of services in a distributed manner. In the late 1990's the architecture-based design approach was effectively split into two further categories: the object-oriented category mainly represented by the CORBA standard [98] and the non-object-oriented category represented by FIPA [88]. Both CORBA and FIPA are currently the only standards in the domain of software agents and will be presented in subsequent sections of this chapter. In the next section we will attempt to define some basic services that an agent system must offer to the agents. In section 2.7 we will elaborate on the points of section 2.6 and define a number of
important issues that an agent system must support. These issues will form the design objectives of the agent architecture presented in this thesis.

2.6 Basic modules of agent systems

The infrastructure needed to support agent systems comprises the following five fundamental components according to [91]:

1. **Execution facility.** The execution facility is essentially the agent's runtime environment. In other words it is the environment within which an agent 'lives'. It must be taken into account the fact that the agent can be mobile, in which case it moves from machine to machine. The execution facility includes the hardware and software necessary for an agent to execute tasks. One of the most important parameters that affects the execution component is the selection of the agent programming language. This factor will be discussed in the following section.

2. **Communication facility.** The communication facility deals with the communication between agents or between agents and other entities. It is the procedure that must be followed in order for information to be exchanged in a transaction. Exchanging messages is the most fundamental means of communication. Messages have *content*, which is the data included in the message, and *context*, which is what changes the content to information. According to [17] the communication facility is implemented as a standard set of protocols and it should support both synchronous and asynchronous communication methods.

3. **Transport facility.** The transport facility deals with how the agent communicates. It refers to the means by which an agent can move from a network location to another in order to perform a task. It also refers to the distribution of static agents to network sites. Transport mechanisms explored for possible use as agent transportation means include Electronic Mail, the UNIX Remote Shell (rsh) and TCP/IP [72].

4. **Packaging facility.** The packaging facility provides a standard method of wrapping agents along with the information describing their internal state. Regardless of their structure, the agents have to store state, authentication, agent capabilities and goal information.

5. **Security.** The security facility should ensure that all the components involved in an agent system are protected. There must be a standard method for determining an
agent's owner (identification). Resources must also be safe-guarded against malicious or ill-programmed agents. Security should be an internal attribute of an agent system and not an additional and perhaps auxiliary feature.

2.7 Mobile software agents issues

In this thesis we will attempt to design an agent architecture that addresses the issues of mobility, access control, agent discovery, agent collaboration and communication. Below we provide a concise analysis of these issues together with a review of relevant research work.

Mobility. Mobility is one of the primary characteristics of an agent and as such it must be among the basic services a mobile agent architecture supports. The problem of mobility can be decomposed into a set of three sub-problems. Firstly an agent must have some means of deciding where to go and what to do (navigation). Secondly the agent system must intercept any migration request from an agent and physically move the agent (code and data) to the desired location. Thirdly the agent system must be capable of tracking all the currently active agents so that regardless of their mobility patterns and frequency, the agents are continuously accessible. An agent architecture can only be said to address the issue of agent mobility if it provides the necessary mechanisms to solve all three mobility sub-problems.

Mobility can be classified as strong or weak depending on whether the agent can start execution from the instruction immediately following the request for migration (strong mobility) or it has to start from the beginning (weak mobility). A typical example of systems following the strong mobility model is the Odyssey [46] whereas a typical weak mobility system is TACOMA [61]. In strong mobility the navigation and computation logic of an agent are integrated into a single monolithic program. This has the advantage that complex navigation logic can be produced simply by using the control constructs of a programming language. However strong mobility implies some form of homogeneity and has a rigid structure which cannot be modified dynamically. Typically this type of mobility is followed by language-based systems. Its implementation is done by: a) incorporating in the language a migration request such as go, migrate, etc. and b) extending the interpreter of the language so that the state of the agent can be saved in an intermediate format for transportation over the network.
In weak mobility navigation is separated from computation. Typically the navigation logic is expressed as a data structure in the form of a travel itinerary. The building block of an itinerary is a simple hop which is represented as a pair destination and action. It specifies where the agent has to go and what action it is supposed to perform upon its arrival. Itineraries can be more complex by adding various control constructs, i.e. repetition, selection, etc. In this way weak mobility can emulate the expression power of strong mobility schemes while bypassing the need for having a special notation (language) for navigation. The main advantages of weak mobility are that: a) it assumes heterogeneity rather than homogeneity; b) the navigation structure is visible to the programmer and it could theoretically be modified dynamically; and c) the computation language does not need any extensions or modifications to incorporate mobility commands. On the other hand in weak mobility, the agent can only be executed from the beginning upon its arrival at the destination.

Communication. There are three ways of performing communication between mobile agents. The first approach is method invocation. In this case agents are object entities which communicate with each other by means of invoking methods from their interfaces. Pure procedure calls are essentially the simplest and most primitive form of communication. The second approach is blackboard systems. A blackboard is essentially a common work area which agents can use to exchange information and knowledge (figure 2.8).

![Figure 2.8 Blackboard-based agent communication](image)

In this approach an agent initiates a communication action by placing an information item on the blackboard. Other agents can access the blackboard and retrieve its current
contents. Blackboards can be protected and agents may need to obtain authorisation before they can access a specific blackboard. In general this communication method is suitable for task sharing and result sharing. The third approach is message passing. Agents exchange messages and can establish in this way communication and cooperation mechanisms using defined protocols. Message passing can be used as a basis for implementing various coordination strategies because the contents of the messages are not restricted to simple command and response structures. For example KQML defines a rich set of speech act primitives that can be used to represent complex communication acts and dialogues in a message passing agent system. Another important characteristic of message passing systems is that they can support asynchronous exchange of messages. In the mobile agent domain asynchrony is essential because this feature can result in a high latency tolerant agent system. Blackboard systems naturally support asynchrony as well in contrast to the method invocation which is inherently synchronous. Typically architecture-based agent systems are based on message passing whereby language-based systems and object-oriented architecture-based systems follow the method invocation approach. FF-main [79] is the only example of blackboard based agent system. There are also two Java-based systems: MARS [28] and Jada [32] whose communication system is based on tuple spaces. Tuple spaces, like the Linda language defines, is essentially a blackboard system whose contents are accessed with associative mechanisms.

Cooperation. Franklin [42] attempted to produce a typology of cooperation (figure 2.9). According to this classification multi agent systems are called independent if their member agents are completely independent from each other and pursue their own goals. Cooperative systems on the other hand have explicit cooperative mechanisms. Cooperation in such systems can either be communicative in which case agents use communications protocols and procedures to achieve cooperation, or non-communicative in which case the agents cooperate indirectly through their environment. In this approach agents observe their environment, sense changes caused by other agents and in response to pertinent changes initiate actions which result in yet more changes in their environment. Communicative cooperation agents can either be deliberative if they share a common planning mechanism or negotiation-oriented in which case they are also competitive with each other. Partial Global Planning (PGP) (Durfee Lesser [39]) is an example of deliberative communicative cooperation systems whereby contract nets (Albayrac [2]) is an example of negotiation-oriented cooperation systems.
There are also two other characteristics that can be used to distinguish different cooperation/coordination models: spatial and temporal coupling. Spatially coupled coordination models require the participating entities to share a common namespace. Temporally coupled models require synchronisation of the participating entities. [28] provides a classification of a number of mobile agent systems according to these characteristics.

An important conclusion we can draw is that communication is coupled with cooperation. The communication method an agent systems follow determines at a great extent the cooperation mechanism as well. For example, FF-main [79] has a blackboard-based communication system and as a result its cooperation model is spatially coupled and temporally uncoupled. We can also observe that blackboard and tuple space-based cooperation systems rely on external entities (environment) to provide the service of cooperation.

**Organisation and discovery.** In a multi-agent system (society) the issue of efficient discovery of suitable agents is very important. There are two possible ways of addressing this problem. Firstly we can assume that each agent knows the name and services of every agent that it wants to contact and interact with. This approach, although adequate for systems where the number of agents is small, becomes impossible as the number of member agents of the system increase. The second approach, and indeed the one all existing agent systems follow, is to provide some form of organisation of agents according their contents/services. There are two variants of this: the matchmakers and the brokers.
A matchmaker (or directory or yellow pages) is an agent with special capabilities that has the task of matching incoming requests for services with published services from service provider agents. Figure 2.10 shows the interaction between an information requesting agent (requester), an information provider agent (server) and a matchmaker in KQML primitives. The requester asks the matchmaker for specific services. The matchmaker searches its database of published services and attempts to match the required services with one or more of the published services. If such a match is found then the address and other relevant details are returned to the requester who is then responsible to contact the server directly without the intervention of the matchmaker. This mechanism shows why matchmaker agents are given various aliases. For example a matchmaker can be called yellow pages because humans flip through the real yellow pages to find the contact details of individuals offering specific services the same way requesting agents search the contents of matchmakers to discover suitable server agents.

Brokers have very similar functionality to that of matchmakers (figure 2.11). The main difference is that the broker not only finds suitable agents on behalf of the requester but also contacts these agents (servers), uses their services and returns to the requester the results. In this way the requester does not communicate directly with any server agents. The broker is commissioned to find and use any agents it decides to be relevant. [25] provides a good analysis and comparison of matchmakers and brokers.
Access control. Access control mechanisms are necessary in a multi-agent system to ensure that only authorised agents can access the contents and services of other agents. The issue of access control is tightly coupled with the issue of authentication. Authentication is the process of verifying the true identity of an agent. Without such a process any access control mechanisms fall apart since agents can masquerade and impersonate others. For this reason it is imperative that all the agents in the system are given upon their creation a unique ID that remains constant for their lifetime and cannot be forged or modified. Furthermore the agents usually, by virtue of being the representatives of tasks of a certain human user, are also given certain security attributes according to their user owner.

In the mobile agent domain there are five approaches for access control. The first approach is Access Control Lists (ACL) associated with every agent. In this approach there is exactly one ACL assigned to an agent. Whenever another agent attempts to access this agent then the ACL is searched by the agent system to decide whether the requesting agent has enough authorisation to access this agent. Typically the ACL is a two-column table matching agent identities to different operations. In Java, for example, there is the notion of a guarded object whereby an object is encapsulated in a guardian object that contains an ACL via which access to the protected object is restricted.

The second approach is with the notion of security proxies. In this case agents cannot access other agents directly but only through their security proxies. A security proxy is essentially an object that decides whether to forward any requests it receives to the real target object. Such a decision is taken according to the security policy and data the security proxy implements. IBM Aglets [58] follow this approach. D'Agents [107] use security proxies (resource managers) as well to protect the host resources from malicious
agents. Security proxies can be made dynamic by introducing permits [46, 47]. Agents can be allowed to access certain resources for a certain time or for a specified number of times by presenting electronic tickets to the relevant security proxies. After the agent’s permit has expired the proxy rejects any further access requests.

The third approach is ACLs extended with a few abstractions. In the first approach the ACL matches agent IDs with operations these agents are allowed to perform on other agents. This approach does not scale well as the number of agents increases. For this reason the notions of roles and rights was introduced. According to this scheme agents are given different roles, i.e. there is a table associating agent IDs with roles, and specific operations on specific agents are organised into rights, i.e. there is a table associating a right with a set of operations on a number of agents. The central element of this approach is a third table associating roles to rights. Roles group together certain agents where rights group together several operations on several agents. In this way the access control system is much more manageable (coarse grain) because it associates groups of agents to groups of operations.

The fourth approach is capabilities. Capabilities are a derivative of the ACLs. The only difference is that rights are not assigned to agents from static tables associated with the resources. Instead the agents themselves carry their rights in the form of tokens. The CORBA security specification will incorporate capabilities in future revisions.

The fifth approach is a form of Public Key Infrastructure (PKI) [43]. PKI has gathered momentum in the area of distributed processing during the last decade. PKI has been incorporated in Web browsers. Java provides a number of facilities to assist the programmer to define, store, distribute and use cryptographic keys. PKI can also be implemented by using KQML as a basis for defining several PKI-related protocols such as key distribution, mutual authentication, etc.

In the next section we will attempt to review critically the early years of the mobile agents. Specifically we will examine firstly how distributed computing is done traditionally, secondly we will provide an overview of the most prominent mobile agent frameworks in the early 1990’s and finally we will conclude by providing a concise evaluation of these early systems according to how they address the issues analysed in this section.
2.8 The early years of mobile agents

The traditional approach in doing distributed computing is the Remote Procedure Calling (RPC). The RPC was conceived in the 1970's and since then it has constituted the main communication standard. According to this paradigm, the communication between two computers A and B is the enabling of computer A to call procedures (programs) that reside in computer B and vice versa. If only one of them requests services from the other then this model of communication is called client-server. This name stems from the fact that the computer which is sending requests can be called client and the computer servicing these requests can be called server.

More specifically, in the RPC paradigm one computer requests the execution of a specific procedure which is resident in another computer by sending, via the underlying network, the requested procedure's actual arguments [81]. This greatly resembles the call-by-value execution of functions and/or procedures in all imperative programming languages [105]. The procedure execution request includes essentially the name or Identification Number of the appropriate procedure and the values that substitute the procedure's formal parameters in the procedure call. The computer which is the host of the called sub-routine, after the reception of the request, executes that procedure with the arguments provided and then sends back to the requesting computer the results. This procedure is illustrated in the following diagram (figure 2.12):

![Figure 2.12 RPC based distributed computing](image)

One of the most important objectives of the RPC is the transparency [81]. This means that the process of calling a procedure should look the same to the user regardless of whether the procedure called is local or remote. Naturally though, the fact that the network is used to transport the client requests to the server imposes restrictions and limitations to the whole process. Thus, in real life, because of network inefficiencies and
bottlenecks as well as server shutdowns, the calling of a remote procedure is not really transparent.

As it can be observed, there are two main disadvantages with the RPC approach: firstly, the client computer can only request services or procedures that are implemented in the server computer. This greatly reduces the flexibility of the system since it constrains the number of possible procedures that can be called by the client. Secondly in order for a remote procedure to be called, the request must first be sent by the client and then the acknowledgement (including the results) must be returned from the server to the client. This means that continuing interaction between two computers relies completely on continuing communication [48]. For this reason, unreliable communication can have a fatal effect on the RPC.

The Remote Evaluation paradigm [115] basically states that instead of the client computer requesting the execution of remote procedures by supplying the necessary arguments, the client can supply the procedure itself to be executed at a remote computer. In this case the client computer sends over the network the procedure and the necessary data in order for that procedure to be executed (figure 2.13).

![Figure 2.13 Remote programming](image)

The Remote Programming paradigm has two important advantages compared to the RPC. Firstly, the *performance* of the system is improved because the network is required to carry less messages, hence the usage of the network, which is the bottleneck in all kinds of computer communication, is reduced. Secondly, and more important, the Remote Evaluation paradigm allows the *customisation* of the services a server provides to the user-client. This is possible since this paradigm, by allowing the user-client to send over to the server its own program (procedure), enables the extension of functionality of the server procedures.
2.8.1 Telescript

The Remote Evaluation paradigm can be further enhanced by allowing the transported procedure to begin execution in the client computer and at a certain stage its execution stopped, following which it is sent to the server computer and its execution resumed there from the point it was stopped. The data that accompanies such a procedure when it is transported to a remote computer is essentially its current state [48].

Telescript was the first commercial mobile agent software based on the aforementioned extension of the Remote Evaluation paradigm. The basic building block of the Telescript system is the autonomous process. An autonomous process is essentially a program whose execution is asynchronous and independent to the execution of all other programs. There are two types of autonomous processes: the Agents and the Places [48]. Agents are mobile processes which can travel in the Telescript network gathering information. Places on the other hand are stationary processes which provide services to visiting agents. An agent, because of its mobility can be in different places at different times. Furthermore, a place can be the host of one or more agents. There are three ways by which an agent can communicate with a place or other agents:

1. Travel. An agent can travel from one place to another. In order to do that it must present a ticket which shows the destination of the trip and the means by which this trip will be accomplished. The travel of an agent is implemented by the go instruction in the Telescript language. After the agent has reached the destination place, it can use the services provided by that place. Moreover, it can establish communication with any agent that it is also there.

2. Meeting. A necessary condition for a meeting between two agents to occur is that both of them occupy (reside) the same place. If this is the case then any one of them can call the procedures of the other. They can also exchange information if they are involved in similar tasks. This agent-to-agent communication method is implemented by the meet instruction of the Telescript language. This instruction also allows one agent to refuse a meeting if the terms it has set, such as the start time of the meeting, are not satisfied. If two agents which occupy different places want to communicate then they can either arrange to meet at a certain place (this usually implies that both agents will have to travel) or establish a

3. Connection. This communication method allows two agents to interact if they are at different places, no matter how remote. It is the traditional approach according to
which the two agents communicate by exchanging messages. It is implemented by the `connect` instruction of the Telescript language. This instruction requires the identification of the target agent, the place it occupies and other terms and conditions. Similarly to the meeting procedure, a connection can be refused if certain criteria are not met.

The Telescript Language (TL) is the language used to encode the procedures of the Telescript agents and places. Its syntax and its set of instructions closely resembles that of the C programming language. Although the TL is complete in the sense that entire applications can be written with it, parts of the application such as the interface to databases can be written in C. Its purpose is to allow developers to implement the communicating parts of an application.

The TL is object-oriented, since it is based upon a hierarchy of predefined classes of objects. Each object has attributes and operations. An attribute represents one of the object’s characteristics. An operation on the other hand is a task the object can perform. Both attributes and operations can be private or public. If an attribute or operation is public then it is visible and can be called from another object.

Apart from being object-oriented, the TL is also communication-centric since the instructions which implement the communication methods among agents are central in its design. Furthermore, it exhibits persistency since the current state of every place and agent are stored in non-volatile memory. Finally it must be noted that one of the most important characteristics of the TL is its focus on safety and security. These features will be discussed in a later section.

The Telescript network consists of interconnected Telescript Engines. A Telescript Engine (TE) is a software program which supports the Telescript language and the execution of the places which reside in that engine as well as the agents that visit those places. A TE can support more than one place and more than one agent. The number of places residing in an engine varies greatly and depends on the host computer in which the engine is implemented. Generally though, it can be said that a TE needs to support a multi-user, multi-process environment.

The Telescript Engine accesses the resources of its host computer through three Application Program Interfaces (APIs) as it can be observed from the following diagram (figure 2.14):
The Storage API allows the engine to access the non-volatile memory of the host computer to safely store its places and agents so that they can be restored in case of a computer failure. The Transport API allows the engine to package and transport an agent to another engine via the underlying communication media. Finally the External Applications API enables the part of an application encoded in Telescript to communicate with other parts of the same application which are encoded in another script language such as C.

The purpose of the safety features of a language is to ensure that a program will do what it is intended to do and if it fails, it fails gracefully. Graceful failure means that persistent objects such as places and agents are safe and can be restored.

On the other hand, security features aim to protect the integrity of the system against malicious users. These features allow the Telescript processes to protect themselves against untrusted places or agents by refusing a travel, meeting or connection.

The most important safety and security features are the following [45]:

- **Interpretation.** The Telescript language is an interpreted language. This implies that every process written in Telescript is executed via the Telescript engine. Since the Telescript engine acts as a mediator in the execution of agents and places, it is clear that the engine can check the code of every executable process and refuse executing any part of it which violates any security parameters.

- **Authorities.** The authority of a place or agent is the user or organisation it represents. Places which have the same authority can be grouped into a *region*. Authority is a central security feature in Telescript. An engine refuses to accept and execute an agent which has failed to pass the authorisation procedure, which may demand
cryptographic forms of proof. An agent can refuse a connection or meeting request from another agent if that agent either cannot show a valid authority or its authority is not in a list of acceptable authorities. Places can act in exactly the same way, thus refusing to be visited by agents which carry unacceptable authority. One important feature of the authority system in Telescript is that it does not allow **anonymity**, thus blocking in this way any virus attempting to deceive the system.

- **Permits.** A permit in the Telescript paradigm is data that grants capabilities to agents. There are two type of permits: the first gives the right to execute a specific instruction and the second allows an agent to use a specific resource at a certain amount, for example it can impose a limit on the number of computations an agent can perform or give that agent a maximum lifetime. The permits an agent has are negotiable in the sense that when that agent enters a place, its permits can be restricted. No place has the right to increase the capabilities of a visiting agent. When it exits that place these restrictions are lifted but new ones can be imposed if it enters another place. Thus, generally, the permits provide a method of protecting the Telescript system by limiting the effects of malicious or ill-programmed agents and places. These restrictions are absolutely necessary in an environment where processes execute without any human intervention or control.

In the late 1990's Telescript was re-implemented in Java and was renamed Odyssey [46]. The Telescript's concepts and mechanisms have been maintained completely.

### 2.8.2 Logic Flow

Logic Flow is a model of distributed computation in which the program (logic) flows through a distributed knowledge network. Peter Sapaty [108] defined a model named as WAVE that follows the principles of logic flow. Navigation in WAVE is implemented by means of spatial matching of strings, written in WAVE language, with the network topology. As the WAVE program moves through the network of data nodes, it carries with it operations and intermediate results. When it reaches a network node, arbitrary processing can be performed locally and data may be left at that node to be used by other WAVE programs. Such an arbitrary processing can result in the dynamic splitting of the WAVE program into segments, each of which moves thereafter as an independent
WAVE program in the network space. Thus there is no synchronisation and centralised control of the segments into which a WAVE program has been split.

The network space in the WAVE model consists of data nodes and links which interconnect the nodes. The data nodes are actually active in the sense they can perform local processing and communication with other nodes. This is possible because each node holds a copy of the WAVE language Interpreter (WI) which enables it to parse and execute any WAVE string which arrives at it.

Each node is assigned a unique address via which it can be accessed from another node located anywhere in the network space. Furthermore, all nodes have contents which is an arbitrary string that can indicate that node’s name, purpose etc.

Similarly to the nodes, the links also have contents (a string usually indicating their name). The links in the WAVE model are the only means of representing a relation. This means that if between two nodes A and B there exists the relation R then this is represented by creating a bi-directional (unoriented in the WAVE framework) link between these two nodes and give as contents to the link the name R.

Apart from the contents, a link can also have an orientation (+ or -). The node which is at the positive side of a link is the active part of it and the other is the passive. Moreover, the links can be classified in the following three categories:

1. **Surface links.** These links are used to interconnect adjacent nodes. They can be oriented or unoriented, but must be labelled (non-empty contents).
2. **Tunnel links.** These links are oriented and unlabelled. Their purpose is to connect non-adjacent nodes temporarily. These links are automatically discarded (destroyed after the end of the communication).
3. **Loop links.** These links are abstract and represent local processing in the same node.

Finally, for every WAVE program there are two special nodes in the network space:

- **The Entry node.** This is the node that receives the WAVE program when first injected into the network. This node has an address of 0, carries no contents, has no surface links to other nodes but has a copy of the WI. For this reason, all the WAVE programs are first executed in their entry nodes and then start propagating through the network if this is necessary. When the WAVE program terminates the results are sent back to the entry node of that program.

- **The Terminal node.** The terminal node is the node that accumulates the results from all other nodes in the network. It has no copy of the WI and, similarly to the entry
node, has no contents. It is accessed via a special variable through temporary unlabelled (tunnel) links.

A WAVE program consists of a sequences of constructs called moves. Each move is separated from its neighbour moves by either a period or a comma delimiter. A period delimiter indicates sequential execution whereas a comma delimiter denotes concurrent execution.

When a WAVE program is injected in the WAVE network space via an entry node, its first move, which is called head, is always executed in the entry node. All the following moves known as tails can be executed in any node, including the entry node. After the execution of the head, the head itself is discarded and the first move of the tail becomes the new head.

The basic principle of the navigation system of WAVE is that the execution of the current head determines the propagation of the tail [96]. So if the head specifies a hop from the current node to a new one then the tail will be executed in this new node. If the hop is multicasting or broadcasting then the set of destination nodes specified are called Goal Set (GS) nodes. Each of the GS nodes receives a copy of the tail, of which the first move will become the head and will be executed first in that node. Hence, if the resulting GS includes more than one node, the WAVE string splits itself in as many instances, called branches, as the GS nodes demand. On the other hand, if the GS is empty then no propagation occurs and the branch dies.

The WAVE language [109] is a language which is used to encode all the programs navigating the WAVE network space. Unlike any other programming language, its only internal information unit is a non-nested sequence of elements (scalars) called vector. A vector behaves much the same way as a list in declarative language or an array in an imperative one. A vector’s elements can be accessed by their indices or contents.

The language cannot be considered as typed since there is only one data type, that of a string. A scalar in fact is any combination of letters, digits and special symbols enclosed in a pair of quotation marks. It can be considered to be a vector of one element, since more elements can be appended to it. The empty value "" does not constitute an element in a string, hence whenever encountered it is skipped.

The WAVE language defines three classes of variables:

1. **Nodal Variables.** These variables are prefixed with N and are local to the nodes of the network space. They can be used to store intermediate computation results or code to be injected. They exist as long as the node to which they belong exists.
2. **Frontal variables.** Frontal variables are prefixed with F and are particular to a WAVE program. They move with it in the network, thus allowing the program to carry intermediate results with it. They are destroyed when the WAVE program to which they belong terminates.

3. **Environmental variables.** These variables are prefixed by E and enable a WAVE program to read and/or modify local information in a distributed network. Some of the environmental variables are read-only whereas others permit their alteration. More specifically these variables are the following:
   - **Node Content C:** Stores the content of the current node.
   - **Node Address A:** Stores the address of the current node (read-only).
   - **Predecessor Node Address P:** Stores the address of the node previously visited by the current WAVE program (read-only).
   - **Content of Link L:** Stores the content of the link, the WAVE program followed, in order to reach the current node (read-only).
   - **Sign of Link S:** Stores the orientation of the link, the WAVE program followed, in order to reach the current node (read-only).
   - **Terminal T:** Enables the access of a special input/output terminal, which is used as a common communication point for all WAVE programs in the network space.

1. Apart from the classification of variables into three categories, another central point in the WAVE language is the definition of rules which enclose any arbitrary WAVE strings and determine its control and coordination. There are nine rules altogether which permit, among others, the sequential or parallel execution of branches, the repetition of a set of commands (loop), the creation of an arbitrary network topology and the synchronisation of WAVE branches whenever required.

The WAVE paradigm, as it was mentioned in the beginning of this section, is a typical representative of a family of language-based agent systems whose main functionality is the provision of a proprietary notation to facilitate autonomous navigational behaviour of programs in a virtual semantic network of passive data nodes. Bic and Fucuda [20] developed a similar system called Messengers which belongs in this category and exhibits very similar behaviour to Sapaty’s WAVE.

Echo algorithms, BPEM and Intelligent Email are three systems developed during the 1980’s that can be considered relevant to this category.
Echo algorithms [31] was one of the first approaches to distributed computing by means of propagating self-contained intelligent messages through a network of simple interpretive nodes. The basic idea behind this paradigm is to initiate a wave of messages from some initial nodes into neighbouring nodes until all the nodes of the network are visited. These messages are called explorers and essentially create a forward spanning tree. When a message reaches a node that has been visited by another message then it stops and starts retracing its path to its origin. This results in a second wave of back-propagating messages called echo which collect information about the graph they are traversing which in turn is combined to produce a global solution. Echo algorithms are very useful because they employ asynchronous message passing to explore the properties of arbitrary networks without any a priori knowledge of the network topology.

BPEM (Binary Predicate Execution Model) [19] developed at the University of California, Irvine, is a model designed to support the parallel processing of knowledge in the form of distributed semantic networks. In BPEM the process of answering a query is translated into the process for finding a topological match for a given template in the underlying knowledge net such that each free variable appearing in the query is bound to a node label of the knowledge net.

Intelligent Email [44] is essentially a family of systems whose objective is to transform electronic mail messages into active or intelligent messages. Typically when such a message reaches its destination it is executed and gathers information from the local host or transports itself to another host. The Intelligent Email paradigm is restrictive because it is confined only within the electronic mail domain. However it is relatively flexible because it does not require each participating host to run a special interpreter like WAVE or Messengers.

Lubomir Bic, in an influential paper [20], provides a classification of all these systems according to the degree of navigational autonomy and dynamic composition each one exhibits.

2.8.3 Agent Tcl

The Agent Tcl is a transportable-agent system which is under development at Dartmouth College, USA. The main goals of this system are to provide multiple languages and transport mechanisms for the agents. Furthermore it must ensure
transparency of communication for the end-user as well as security and fault-tolerance of the whole system [52]. Its name stems from the fact that its first implementation (alpha release) only supported the Tcl programming language. Similar to Telescript, the Agent-Tcl system was re-written in Java in the late 1990's and was renamed D'Agents [107].

The architecture of Agent Tcl [52] is depicted in the following figure (figure 2.15):

![Agent-Tcl Architecture Diagram](image)

**Figure 2.15 Agent-Tcl architecture**

As it can be seen, the Agent-Tcl system consists of four levels. The lowest level comprises an API for each supported transport mechanism such as TCP/IP, Email etc. The second level is the server that runs at each network node. Among the tasks a server performs are:

- Keeping track of the agents executing at that site and of the available interpreters.
- Accepting incoming agents and authenticating their identity.
- Allowing the agents to exchange messages.
- Providing the transportation means by which an agent can migrate to another machine.
- Providing access to the nonvolatile memory and restoring the state of the agents in the event of a node failure.

The server is implemented as two cooperating processes. The first process is the *socket watcher* which is listening to a UNIX socket for incoming agents, messages and requests. The second process is the *agent tabler* whose purpose is to keep track of the executing agents and buffer messages between agents until the destination agent receives them.
The third level of the architecture consists of an interpreter for each supported language. The interpreter itself comprises four modules:

- An Interpreter module for a language.
- A Security module that ensures that the agent will not violate any access restrictions.
- A State Capture module, that captures and stores the state of the executing agent.
- A Server API module which interacts with the server to provide agent-migration and inter-agent communication.

Finally the fourth level of the Agent Tcl contains the actual agents.

The Agent Tcl is essentially the Tcl language as developed by John Ousterhout with the following modifications:

- The Tcl core was modified in order to provide facilities for capturing the internal state of an executing agent.
- New commands that allow the agent-migration and communication were added. The most important of them are:
  a) *Agent_send* and *Agent_receive* which are used to send and receive messages between agents.
  b) *Agent_meet* and *Agent_accept* which are used to establish a direct communication channel between two agents.
  c) *Agent_submit* which is used to create a child agent and then move it for execution to another network node.
  d) *Agent_jump* which is used to capture the state of an agent, transport it to another machine and then resume execution at the statement immediately after the agent_jump.
  e) *Agent_begin* which is used to obtain a unique name in the network space in order to be able to be identified and perhaps contacted by other agents.

The security features of Agent Tcl have the following four objectives [53]:

i) Enable the server to authenticate every incoming agent, to impose on it access restrictions (authorisation) and ensure that these restrictions are not violated (enforcement).

ii) Ensure that execution of an agent does not interfere in any way with the executing environment of another.
iii) Protect each agent against malicious actions from the server in which it resides. This means that each server must not be capable of pulling sensitive information out of an agent without the agent’s cooperation.

iv) Ensure that an agent does not consume excessive resources in the network even if it consumes only a few at each machine.

The current implementation of the Agent Tcl achieves the first two of the previous objectives. More specifically, authentication is based on the Pretty Good Privacy (PGP) system. The file to be encrypted, which can be either an agent that will travel to another machine or a message or a request, is sent to the PGP, which runs as a separate process. The PGP process encrypts the file which is then sent to the destination machine where it is decrypted. For example, when an agent issues an agent_jump command then the agent is digitally signed by the current server’s private key and encrypted with the destination server’s public key. Regarding the issue of authorisation, the resources are divided into two categories: indirect resources which can be accessed through other agents, and built-in resources which can be accessed through the language commands. For the indirect resources, the agent which controls the resource is responsible for the access control of that resource. The server attaches to every message from another agent a 5-tuple which contains information about the identity of that agent and the trust it can be given. Then, based on this information, the agent controlling the resource responds appropriately to that message.

For the built-in resources the Agent Tcl system uses Safe Tcl in combination with resource manager agents [53]. In this way, agents are initially executed in the unsafe interpreter of the Safe Tcl from which all ‘dangerous’ commands are removed. If the agent wishes to use a resource it has to either explicitly ask permission from the relevant resource agent, by using the require command, or implicitly by issuing a command that uses the resource.

2.8.4 Tacoma

The TACOMA is a joint project between University of Tromsø and Cornell University [61] whose aim is to provide operating system support for agent-based distributed computation. It is focused on providing flexible yet simple mechanisms that allow the distribution of computations over a network and their remote execution.
The computational unit in TACOMA, as in Agent Tcl and Telescript, is the agent. The agent is essentially defined as being a set of sequentially executed instructions with some initial state. This resembles the definition of a process in UNIX with the difference that not all of the instructions have to be executed at the same place.

The state of the agents in TACOMA is organised into units of data called folders [61]. Each folder has an ASCII name and an agent can have one or more folders. For instance, a CODE folder can contain the source code of an agent, a HOST folder can contain the names of the servers that agent must visit etc. Folders can be grouped together into logical sets called briefcases. Hence, an entire agent can be represented by a briefcase. Consequently, moving the agent to another place in the network means transferring its briefcase to the destination machine. The main characteristic of the briefcase is that it can be moved. But, stationary folders are also required to be used as a permanent data repository. A stationary collection of folders is called a file cabinet in TACOMA.

The basic abstraction found in TACOMA is the meet operation which allows two agents to meet and hence the following general syntax:

```
meet agent briefcase
```

where `agent` is the agent to be met and `briefcase` the briefcase representing the current agent. It must be noted that the host at which the destination agent resides must be specified by the folder HOST in `briefcase`.

For example, if an agent was written in C, was represented by the briefcase TEST and wanted to be compiled then, assuming that the name of the agent which can compile a C program was Compiler, the required command would be:

```
meet Compiler TEST
```

At each network site there are two types of stationary specialised agents: the firewall agent and one or more instances of the `exec` agent. The firewall agent provides a single entry point for guest agents at that site. Hence this agent implements authentication, access control and accounting of the arriving agents. It is the only gateway of that particular site to the rest of the network. The firewall agent, after completing the security check, stores the briefcase of the visiting agent in the nonvolatile memory of the server and notifies one instance of the exec agent that there is a briefcase that just arrived. From that point onwards the firewall agent is free to accept new incoming agents.
The exec agent, which as said before can have two or more replicas active, sets up the executing environment for the guest agent. In this way it can impose more restrictions by limiting the access the guest can have to resources.

The TACOMA API includes folder, briefcase, and file cabinet abstractions as well as abstractions for meeting and executing agents. As mentioned in the previous section, the main abstraction is the meet operation which has the syntax:

\texttt{meet agent briefcase}

whose result is to activate the agent with the briefcase as argument. Apart from this abstraction, there are abstractions that enable the manipulation of data. These provide:

- Creation and deletion of briefcases by using \texttt{bc\_create} and \texttt{bc\_discard}.
- Manipulation of folders within briefcases, for instance: \texttt{folder\_store bc folder data}, which stores data in the \texttt{folder} which is itself in the briefcase \texttt{bc}, \texttt{folder\_fetch bc folder} which fetches the \texttt{folder} from the \texttt{bc} briefcase etc.
- Manipulation of a folder's contents if these are a list of elements, for instance: \texttt{folder\_pop bc folder} which fetches the first element of the \texttt{folder} in \texttt{bc}, \texttt{folder\_push bc folder data} which adds an element to the start of the \texttt{folder} etc.
- Manipulation of file cabinets by using abstractions similar to those manipulating on briefcases, such as \texttt{cabinet\_create name}, \texttt{cabinet\_fetch cabinet folder}, \texttt{cabinet\_store cabinet folder data} etc.

\section*{2.8.5 HTTP-based agents (FF-main)}

The HTTP-based Infrastructure for Mobile Agent (Lingnau et al. [79]) is a project at the Goethe University in Germany. It is designed to provide a low-level infrastructure to support agent mobility and communication through HTTP. The main architectural component of FF-main is an agent server which runs at every host and its task is to provide a supervised execution environment for the agents that visit the specific host. The agent server supports some essential agent operations such as: create an agent, supervise an agent, terminate an agent, etc. Furthermore it is responsible for transporting agents to other hosts and manages the interaction between agents and their users.

Each agent is assigned a unique URL upon its creation. The agent server where an agent was born is called the home server for that agent and it is responsible for keeping
track of the agent’s location. Agents are transported to other hosts as MIME documents by POSTing them to the URL of the destination agent server. Upon its arrival, the agent is parsed by the agent server and if the agent is accepted then the agent server starts the execution of the agent and assigns to it a visitor URL. Furthermore it forwards to the agent’s home server the new URL of the agent.

Agents in FF-main cannot communicate directly. Instead each agent server contains an information space which is accessible by the agents via the CGI methods of GET and POST. Each entry in the information space is assigned an access header. The agent server mediates every access request that the local agents generate in order to ensure that the requesting agent’s credentials satisfy the access criteria of the specific entry.

2.8.6 Evaluation of early agent systems

The presented early agent systems have predominantly followed the language-based design approach. WAVE, Messengers and the rest of the agent systems of the same family are based on proprietary specialised scripting notations. Telescript is an example of a general purpose proprietary language-based agent system. Agent-Tcl is an example of a widely available language-based agent system whereas TACOMA is the only system that attempts to follow the architecture-based (language-neutral) approach. The main benefits of these systems is that firstly they support code mobility in various ways and secondly they also support the asynchronous execution and propagation of the agents.

In general, however, the early language-based agent systems relied on proprietary, complex notations that have complicated semantics and awkward syntax [30]. The result of this approach is that these systems cannot exhibit any interoperability, portability or integration characteristics. Furthermore several of these systems were oriented towards specific applications, for example WAVE and Messengers suited well distributed graph-related problems such as discovery of the topology of an arbitrary network and discovery of the shortest path between two nodes. These characteristics seriously reduced their overall significance and acceptance level by Internet end users.

A second negative point of these systems, with the exception of Telescript, is that they viewed the agent environment as a set of passive data nodes that do not exhibit any autonomous activity. Instead, as Telescript proved, most generic distributed computing
scenarios can be best modelled if these stationary nodes are active and offer certain services to the mobile agents.

Below we will attempt to critically review these systems in more detail according to the navigation, mobility management, communication, organisation, collaboration and access control mechanisms they support:

- **Navigation.** The navigation logic is fully embedded within the computational logic with the exception of TACOMA. Clearly there was the consensus that strong mobility is more powerful than weak mobility supported only by TACOMA. This approach however, because it relies on a certain degree of homogeneity, nowadays comes in direct conflict with the nature of distributed computing environments such as the Internet which are highly heterogeneous.

- **Mobility management.** No mobility management services were offered by these systems with the exception of ff-main. Mobility management, or else the support for the dynamic tracking of the locations of the mobile agents, is necessary to ensure that agents are continuously accessible regardless of their mobility. If such a service is not provided then the functionality and capabilities of the agents are seriously degraded. FF-main provided a mechanism based on the idea of Mobile IP in the domain of mobile hosts. However this mechanism only assisted the discovery of the location of the agents by their users and not by other agents. No automatic mechanism was provided for the latter case. Furthermore in FF-main each server has to be aware of its neighbours to enable mobile agents to decide where to go next.

- **Communication.** Communication was either performed via message passing (Telescript) or data sharing (WAVE, FF-main). In the former case agents can exchange messages synchronously whereas in the latter case agents communicate by leaving data in data nodes that other agents can access and utilise. The data sharing approach imposes the problem of protecting these data and ensuring only specific agents can access specific entries. WAVE and Messengers did not provide any mechanisms for secure data access in the nodes. FF-main append a primitive form of Access Control List (ACL) into each entry in the information space. In general these systems also did not use any higher level protocol for the communication of agents. Furthermore no message-passing related services such as buffering and filtering was provided.
• Organisation. The early systems had mainly focused on providing support for mobility. No agent organisation structures were provided and, as a result, agents were not capable of discovering pertinent peer agents and interact on a service basis. Each agent had to know in advance the names and addresses of the agents it had to communicate with. This approach is very restricting because it assumes that the agents have upon their creation some form of global knowledge of their environment based on which they interact with other agents. Agent-Tcl and Telescript provided some form of local “yellow pages” that agents could use to find relevant local agents. These local structures though were isolated from each other and were not interconnected to provide a comprehensive distributed agent discovery structure.

• Collaboration. In early systems collaboration was implemented either as an exchange of messages (Telescript) or as dedicated entities (nodes) that can store shared data (Messengers, FF-main) acting effectively as passive shared blackboards. No protocols or mechanisms for higher-level collaboration was provided.

• Access control. The issue of access control is completely omitted in Messengers and the other related systems. In TACOMA although a firewall agent was provided which was responsible for maintaining the security of a network site, there was no presentation or analysis of specific security features. Agent-Tcl initially had no security features, although these were later added in an ad-hoc fashion. Still though Agent-Tcl has failed to provide a mechanism for flexible, customisable inter-agent access control, having focused on mainly protecting agents from malicious machines and vice versa. Telescript had by far the most comprehensive security system. However the agents were assigned static ACLs that could only be modified by their owners. No security policies sharing mechanisms were provided. The Telescript security system was focused primarily on providing access control mechanisms for the resources through the Telescript interpreter and the concept of Telescript permits. Finally FF-main completely lacked any access control features with the exception of the access header in the entries of the information space which were static and predetermined.

In the next section we will attempt to provide a comprehensive overview of contemporary mobile software agents systems.
2.9 Modern mobile agent systems

The design and development of modern agent systems has been greatly influenced by the emergence of two standards: Java and CORBA. Java has been used extensively as the platform for the development of language-based agent systems, whereas CORBA [98] offers a comprehensive platform for developing object-oriented, architecture-based agent systems. FIPA [88], the relatively new standard for agent systems is also very relevant but until now has had very little impact on the development of agent systems. In this section these three standards will be outlined. Moreover an overview of a selection of the most prominent agent systems will be presented. Finally we will attempt to evaluate these modern agent systems based on the issues identified in section 2.7.

2.9.1 An overview of Java

The Java programming language [51] developed by SUN Microsystems has two important characteristics that make it suitable for the development of generic client-server applications: it offers a network-based programming environment and platform independence. Various network programming facilities are incorporated into the Java specification for example interfaces to access Berkeley sockets in the Unix operating system, etc. The Java compiler does not translate the Java source code into binary format directly. Instead it transforms the source code into a platform-independent intermediate form called Java byte code. The Java byte code can then be ported and executed directly on all platforms that support Java. The so-called Java virtual machine is used to execute Java byte code on the target platform.

Java is a fully object-oriented language whose syntax is based on the C++ programming language. There are several differences between C++ and Java. The most important ones are that Java provides no pointers structures, no operator overloading, no direct access to memory and no multiple inheritance. On the other hand Java offers automatic garbage collection, multithreading and exception handling.

The Java language and run-time environment (Java virtual machine) provide a number of principles and mechanisms that can facilitate the development and
management of software agents. Specifically the Java services pertinent to software agents are the following:

- The Java security model.
- Support for distributed computing through the mechanism of Remote Method Invocation (RMI) and Java Applets.
- Object mobility through object serialisation.

Java by virtue of being network-based has to provide a far more comprehensive security model than those provided by traditional programming languages. For example, the notion of automatic downloading and execution of Java programs from the network (Java Applets) makes it possible to execute a program of unknown origin and contents in the local machine. This clearly could be disastrous since the downloaded program could be malicious and for instance attempt to erase the contents of the local file system. Indeed Java Applets if unrestricted would provide an excellent platform for the development and deployment of computer viruses at a massive scale.

The Java security model consists of several layers, some of which are embedded within the language itself whereas others are part of the Java run-time environment. The Java class loader forms the first security layer and it is responsible for receiving applets and Java classes from the network. The class loader is a component of the Java virtual machine and its main functionality is to define a name space for each received applet. This name space determines the functions of the Java virtual machine the applet is allowed to access and invoke. The class loader itself is not accessible by any applet whereas the other components of the Java virtual machine can be visible to applets that satisfy certain security requirements, for example, they come from a specific origin, etc. The restricted execution environment assigned to an applet is effectively an implementation of a sandbox as analysed in (Müller [87]). By default an applet's sandbox is very restrictive and prohibits the applet from accessing the network, the local file system or other concurrently executing applets.

After an applet is successfully downloaded from the network and its name space is set the next security component activated is the verifier. The verifier checks the applet's byte code to ensure that: a) the Java language specifications are met and b) the language rules are not violated. Typical programming errors in memory management, illegal data type assignments, etc. are identified by the verifier. In this way it is ensured that an applet's code when executed will not harm the local system.
An applet that successfully passes the stages of the class loader and verifier is allowed to be executed. At this stage a third security component takes over and monitors the execution of the applet, the security manager. Its task is to ensure that the applet can not access any service/functionality of the Java virtual machine which is outside the applet’s sandbox. The security manager acts as a mediator and it is consulted by the Java virtual machine whenever the applet attempts to perform an action that could directly affect the local system. Broadly speaking the security manager is a repository of various access control policies. In a typical scenario these policies permit or prohibit an action depending on whether the requesting Java program is a Java applet or application.

In addition to the above security components the Java language offers a number of further security-related services that enable the Java programmer to: a) authenticate and digitally sign applets and other classes, b) use various encryption techniques, c) define cryptographic keys distribution and management policies, etc.

A second important agent-related Java mechanism is the RMI (Remote Method Invocation). The RMI mechanism allows a Java object to invoke methods from other remote (physically distributed) objects (figure 2.16).

![Java RMI architecture](image)

Figure 2.16 Java RMI architecture

Briefly the first step in implementing the RMI mechanism is for a Java object to publish its interface in the local RMIregistry facility of the local Java virtual machine. By doing so the Java object registers its willingness to accept invocations of its methods by remote objects. The result of the first step is the generation of skeleton code which is stored in the local RMIregistry. The second step is for a remote object to request the
invocation of a method of the previous object via its local RMI registry. This registry is then responsible for locating and communicating with the RMI registry of the target object. The result of this inter-RMI registry communication is that a copy of the target object's interface is downloaded to the local virtual machine of the requesting object. This copy is called client stub. The final step is for the RMI registry to pass a reference of this stub to the requesting object which can subsequently use it and invoke any of its methods directly. The source and target Java virtual machines are responsible in communicating the actual invocation requests and results of the requests. From the viewpoint of the requesting object the procedure is transparent which means that it can invoke a method from the remote object the same way it invokes methods from local objects.

Object serialisation extends the input/output capabilities of Java and supports the transmission of a Java object (class and state) through the network. This mechanism is concerned with the packaging of a Java object's definition and state into a serial, byte-based stream, the transfer of the byte stream to the target machine and finally the unpacking of the byte stream and reconstitution of the original object at the target machine. Object serialisation provides a ready-made facility that Java-based software agents can use to physically migrate from one machine to another (agent mobility). In fact the Java object serialisation can be used to implement strong mobility for agents because an object (agent) can be interrupted at any time during its execution, transformed into a byte sequence, transmitted over the network, reconstituted at the destination machine and execution resumed at the next program instruction after its interrupt (request for migration).

2.9.2 An overview of CORBA

The Common Object Request Broker Architecture (CORBA) created by the Object Management Group (OMG) [95] represents one of the two most important efforts in standardising distributed object computing. The second standard in this area is the Distributed Common Object Model (DCOM) from Microsoft [97]. DCOM in contrast to CORBA is a standard proprietary to Microsoft whereas OMG is a grouping of currently more than 700 companies and organisations. The main aim of the OMG members is to provide a reference architecture and a set of specifications that maximise the portability, reusability and interoperability of software. Effectively the OMG provides an open
discussion forum for the advances in object-oriented technologies. Because of the nature, organisation, members and aims of the OMG, its main output, CORBA, automatically becomes a highly important standard.

The central component of the CORBA system is the Object Request Broker (ORB) (figure 2.17). The ORB's main functionality is to act as a trusted mediator in every interaction between two or more local/remote user objects. The ORB architecture offers a number of important services to the user objects such as communication, security, yellow and white pages, etc. The name *Broker* stems from the fact that when a user object wants to communicate with another object then the ORB intercepts the request, forwards it to the specified destination, performs the operations and return the results to the requesting object. In other words in CORBA objects do not communicate directly with each other but only with the ORB.

![Figure 2.17 CORBA architecture](image)

The Interface Definition Language (IDL) is another important component of the CORBA system. IDL is a declarative notation that objects can use to publicise their interfaces in a standardised representation. The IDL specification does not therefore define any programming constructs but provides the necessary notational structures to describe object interfaces. The CORBA specification makes a clear separation between the definition (description) of an interface and its implementation. In fact the interface descriptions and implementations are stored in different CORBA components (interface repository and implementation repository respectively). This separation gives CORBA
one of its main strategic advantages because in this way the user-defined objects of the CORBA system could be written in different languages, executed in different hardware or operating system platforms but still be able to communicate and interact with each other (via the ORB). Such interoperability can be supported because all the objects regardless of their implementation have standardised, IDL-based interfaces via which any object can find, understand and invoke a method of another object.

Because CORBA is a fully object-oriented architecture, like JAVA, it relies on RMI to implement the most basic act of object-oriented distributed processing which is the invocation of a method of a remote object. The requesting object produces a request message that contains the CORBA ID of the target object. The CORBA ID is a long integer number used to uniquely identify every object in the CORBA system. The requesting object could find the CORBA ID of the target object either statically or dynamically (static and dynamic method invocation). In both cases the ORB locates the target object (skeleton server) from which it finds the location of the actual implementation of the target object. Upon completion of the execution of the target object implementation the ORB returns the results to the invoking object. Here it must be noted that the RMI mechanisms of both Java and CORBA support only synchronous communication between objects.

CORBA is part of a further comprehensive architecture, the Object Management Architecture (OMA) (figure 2.18).

Figure 2.18 OMA architecture
The ORB plays the role of a central software bus that interconnects individual applications (application interfaces), specific application area services (domain interfaces) and general system services (object services and common facilities).

The object services also known as CORBA services provide general services for all CORBA-based applications. These services include naming, persistence, relationships, security, query, object discovery, etc. The common facilities, CORBA facilities, provide horizontal facilities required to support different application categories. The main difference between object services and common facilities is that the first views each individual object on its own whereby the second classifies the objects into groups according to the application category they belong to. Essentially the CORBA facilities are a set of IDL-based interfaces that an object must support in order that it can provide a specific service. For example, in order for an object to provide the service of printing it must support and implement a print interface. The domain interfaces are application domain-specific interfaces. In this way specific application domains such as electronic commerce, knowledge management, etc. can have their own set of interfaces. These interfaces, if implemented, must be available to all objects that belong to this application domain. Finally application interfaces are interfaces defined and used within the context of a single application. Clearly these interfaces cannot be standardised and thus the OMG does not provide any specification for these. An OMG Object Framework [95] is a collection of collaborating individual objects each of which belongs into one of the previously mentioned four object categories.

CORBA is capable of providing direct support for the development and deployment of software agent architectures. The ADERT system is an example of CORBA-based agent interfaces [94]. The OMG underlined the relationship between CORBA and software mobile agents by publishing in 1997 the specification of a Mobile Agent System Interoperability Facility (MASIF). This action also highlighted the importance of providing interoperability among existing and future mobile agent systems. MASIF addresses the interfaces between agent systems in the context of Kinny et al [66]. MASIF essentially defines the interfaces at the agent system level rather than at the agent level. Specifically MASIF proposes standard interfaces in the following four areas [75]:

- Agent management. This includes standardised operations for creating, executing, suspending, resuming as well as terminating agents.
• Agent transfer. This addresses the issue of how an agent is packed, transported and reconstituted at the destination machine.
• Agent and agent system names. The syntax and semantics of agent and agent system names is standardised. Through the proposed naming scheme agents and agent systems can identify and thus interact with each other.
• Agent system type and location syntax. Agent systems are assigned types and agent transfer can only be performed if the destination agent system is of the appropriate type to support and execute the transferred agent. Location syntax standardisation is also required such that agent systems can locate each other.

2.9.3 Foundation for Intelligent Physical Agents (FIPA)\footnote{The information in this section has been extracted and compiled from the official FIPA97 documentation at \url{http://www.fipa.org}}

The Foundation for Intelligent Physical Agents (FIPA) is a non-profit association registered in Geneva, Switzerland. FIPA’s purpose is to promote the success of emerging agent-based applications, services and equipment. The FIPA 97 specification is the first output of the Foundation for Intelligent Physical Agents and specifies the interfaces of the different components in the environment with which an agent can interact, i.e. humans, other agents, non-agent software and the physical world. FIPA produces two kinds of specification:

FIPA 97 defines primarily three basic agent technologies: agent management; agent communication language; and agent/software integration.

The agent management specification defines agent registration, agent message passing, agent lifecycles, and an agent platform (AP). An agent management ontology has been defined to facilitate interoperability between agent platforms using FIPA ACL. FIPA envisages a variety of different agent platforms from single processes containing lightweight agent threads, to fully distributed agent platforms built around proprietary or open middleware standards.

The agent reference model provides the framework within which FIPA agents exist and operate (figure 2.19). The Directory Facilitator (DF), agent Management System (AMS) and Agent Communication Channel (ACC) are specific types of agents which support agent management. The AMS and ACC support inter-agent communication. The
ACC supports interoperability both within and across different platforms. The Internal Platform Message Transport (IPMT) provides a message routing service for agents on a particular platform.

The ACC, AMS, IPMT and DF form what will be termed the agent Platform (AP). Each agent has a Globally Unique Identifier (GUID), also known as agent name, which is a string global over all FIPA domains which labels the agent so that it may be unambiguously distinguished in the agent universe. An agent may be registered at a number of addresses at which it can be contacted.

The Directory Facilitator (DF) provides "yellow pages" services to other agents. The DF is a mandatory, normative agent which is the trusted, benign custodian of an agent directory. Agents may register their services with the DF or query the DF to find out what services are offered by which agents. At least one DF must be resident on each AP (the default DF). The membership of a DF directory defines an agent domain. A domain is a logical space which provides a context within which agents may organise and locate each other. One AP can support multiple domains, one domain can span multiple AP's.

An Agent Management System (AMS) is a mandatory component of the AP. It is an agent which exerts supervisory control over access to and use of the ACC. Only one AMS will exist in a single AP. The AMS is responsible for managing the activities of an AP. These responsibilities include creation of agents, deletion of agents and overseeing the migration of agents to and from platforms. The AMS maintains an index of all the agents which are currently resident on a platform. The index includes an agents GUID and their associated transport address for the AP [18].

![Diagram of FIPA agent management reference model](image-url)
2.9.4 A critical review of contemporary agent systems

The majority of contemporary agent systems is based on Java. There are several reasons for this phenomenon: Java is a widely accepted programming language which comes with a sophisticated programming environment (Java virtual machine) that offers several services such as Java beans, security, serialisation, RMI, etc. Typically an agent is represented as a Java object that implements a specific agent class and interface. The agent interface offers mobility abstraction by the provision of a method `go` which, if invoked, transfers the agent to another location. The Java serialisation interface is used to implement strong mobility. The Java security manager and class loader are configured to perform access control and other security related tasks on behalf of the agents. Communication between agents is synchronous and is implemented by using the Java RMI.

However there is also a small number of agent systems that have followed the agent communication language approach or the object-oriented architecture-based approach. The former category typically creates an environment in which agents can communicate and migrate by using KQML as the communication protocol. The latter category includes CORBA-based agent systems and programming environments. A typical representative of this category provides a CORBA ORB implementation with the appropriate interface to use this ORB through a Java program.

In this section we will review the most prominent contemporary agent systems from the point of the collaboration, navigation, organisation, access control, communication and mobility management facilities they offer. Specifically the systems we will include in our analysis are: Aglets [58], AJANTA [117], ARA [101], Jypsum [76], Jat-Lite [60], Java-to-go [78], Knowbot [63], MAP [67, 84], MARS [28], MOLE [116], Concordia [83, 123] and the CORBA relevant specifications.

The issue of collaboration has in general been neglected. All systems claim that adequate collaborative behaviour can be exhibited via the support for method invocations. AJANTA and Concordia go a step further and define specialised objects called synchronisation objects. These objects can be created and used by the agents to support their coordination. Each such object represents a synchronisation expression which is evaluated every time new data arrives. When the expression yields true then an associated action is taken which typically forwards a specific subset of the accumulated data to a specified destination for further processing. MARS is definitely the most
elaborate system that focuses primarily on providing the service of coordination/collaboration to independent, concurrent Java programs. MARS achieves this by extending Linda's notion of tuple spaces to incorporate reactivity. There are two levels of tuple spaces. The first one is Linda like. The second one operates on top of the first and adds reactivity to the basic tuple space. The approach for using specialised objects effectively belongs in the category of spatially uncoupled systems whereby the second is both spatially and temporally uncoupled. Both approaches however create an environment via which the agents can receive the service of collaboration, therefore they both belong to the collaborative, non-communicative multi-agent system. This category suffers though some important disadvantages: a) the agent's autonomy is unnecessarily reduced since they have to rely on an external environment for the service of collaboration, b) access control is also complicated because agents must protect their collaboration tuples/synchronisation objects from unauthorised access. By separating an agent from its collaboration structure the system is assigned the burden of protecting two entities rather than one, c) environment based collaboration is not efficient because there is the extra communication between the agent and the collaboration structures for delivering/fetching data, d) tuple spaces are not mobile. This means that if an agent moves, its collaborative structures, by virtue of being separate from it, do not follow the agent in its travel. Consequently the tuple spaces must have a way of determining the current location of the agent so that any necessary data can be appropriately forwarded. Furthermore the whole collaboration becomes more inefficient as the target agent moves because agents that produce data for the collaboration point might need to send this data to the location of the collaboration point itself rather than the agent. As a result data might travel to the location of the collaboration which if activated will send the data to the corresponding agent located at the same host as the producers of the data. For these reasons in chapter three of this thesis we describe and analyse simple yet powerful collaborative facilities which are part of the agents themselves.

Most of the aforementioned systems follow the strong mobility model and implement it using the Java serialisation facility. As it will be described in chapter three, strong mobility is less suitable for agent-based computing over the Internet which is inherently a heterogeneous environment. AJANTA and IBM Aglets attempt to emulate weak mobility in Java by equipping their agents with itineraries. These itineraries however are expressed in the form of rigid object hierarchies that can support only minimum dynamic extensibility. Furthermore the navigation logic of each agent is proprietary to that agent.
Other agents cannot copy relevant parts of the navigation logic of a specific agent and use them as off-the-shelf components to plug in and use for their own navigation. Such a feature can be very useful because it views the navigation logic of an agent as a knowledge unit that other agents can utilise as re-usable software components. Another disadvantage of the IBM Aglets and AJANTA is that the interconnection among itinerary items is static and predefined, thus the execution of the action associated with one of these items cannot modify the contents of the other items. Such a feature is also important because it makes the agent’s itinerary fully dynamic and adaptive to the agent’s environment. Java-to-go uses the TACOMA concepts of folders and enables the agent to decide what items to take with it prior to a specific migration. This style of navigation, whereby an agent decides which data will be transported with it during a migration act is beneficial because agents can reduce their size prior to their travel by taking only the essential data items with them. In this way the overall utilised communication bandwidth can be substantially reduced. Chapter three of this thesis provides a flexible, dynamic and re-usable architecture to support the navigation of agents based on the weak mobility model.

Agent discovery is another issue neglected in most of the above systems. MOLE uses an X-500 directory structure for agents. JAT-Lite, Jypsy, AJANTA and D’Agents provide simple yellow pages agents in every host via which an agent can discover relevant local agents by name. Telescript, and the later version Odyssey, provides simple directory structures per Telescript place. CORBA, via its traders specification, provides an extended model whereby trader objects act as matchmakers that receive queries from objects, match these queries against their database of registered services and either return the contact details of relevant objects to the requesting object or propagate the request to other traders for further processing. Thus in CORBA traders can be connected in a hierarchy that a query might travel either because the owner of the query specified a specific trader path to be followed or from the trader’s own volition and policies. Connecting matchmakers together in a hierarchy has been around for several years. In the information retrieval area there is a number of systems that follow this approach [22, 23, 33, 111, 112]. In chapter four of this thesis we will revisit the issue of agent discovery and we will show that there are some significant benefits of adopting a more active organisation structure which participates in several aspects of the computation itself.

Access control in these systems is either provided by customising the security mechanisms of the Java virtual machine or by providing an abstraction layer on top of
the JVM. IBM aglets and AJANTA implement a proxy-based access control mechanism in which agents do not communicate directly with each other but only through security proxies that implement certain access control policies. There is one proxy per agent. Telescript follows the capabilities approach via its notion of permits. D'Agents employs the method of indirection to protect resources from malicious agents via its resource managers. The CORBA security specification is by far the most comprehensive. CORBA defines the interfaces for managing and using security domains, access control policies, user credentials, rights and roles. The access control model of CORBA is centralised and basically is an example of the third category identified in section 2.7. The specification does not define how security domains and security related objects are created, how objects are assigned to domains, etc. Other systems for example JAT-Lite and Java-to-go do not provide any access control mechanisms. In general none of the aforementioned systems offers a distributed, customisable on a per-agent basis, reusable framework for access control. Such a framework is necessary to ensure that the agents are truly autonomous even for access control purposes, agents can define and manage directly and efficiently their access control policies which can be shared among a group of agents to provide for group access control mechanisms. Chapter five of this thesis presents and analyses the design of such a framework.

The communication method adopted by most Java-based agent systems and the CORBA specification is the synchronous remote method invocation. Java-to-go uses the notion of channels for connection-oriented asynchronous communication. JAT-Lite supports message passing asynchronous communication through Agent Message Routers (AMR). These entities act as high level routers of messages expressed in KQML. Arguably the message passing paradigm offers a flexible basis over which complex communication protocols can be established among agents. Furthermore message passing is inherently asynchronous, a characteristic which is vital for agent-based systems because it can hide significant communication latency. The OMG and Sun Microsystems have announced the integration of message passing facilities in CORBA and Java respectively. In chapter four of this thesis we present an agent communication architecture which is message oriented and can also support message passing related services such as message filtering and buffering. Recently we have observed the development of a number of commercial message oriented software such as MQSeries from IBM [57], PIPES from PeerLogic [100], etc. This shows clearly the shift of the research and commercial community towards Message Oriented Middleware (MOM).
Finally mobility management has largely stayed in the background of the agent research community. Software mobility management has several fundamental differences compared to hosts management and as a result mechanisms and algorithms utilised in today's mobile telephony cannot be applied to agent mobility. The OMG's agent specification MASIF simply defines the interfaces for mobility management without offering or attempting to standardise any concrete solution to the problem. AJANTA like MASIF specifies that each agent as it moves could leave forwarding pointers to its new location. These pointers are stored in a local registry which is used by agents to find the locations of other agents. Ff-main uses the concept of a home system for each agent which is responsible in maintaining the most up-to-date location of the agent. Ff-main however does not provide any automatic mechanisms that would help the agents locate each other since the aforementioned method is used only to assist the human operators of the system to locate their agents. FIPA specifies that agents should be responsible in notifying their home system of their whereabouts and that they could also leave forwarding pointers in other systems. In chapter six of this thesis we present and analyse an integrated message passing and mobility management architecture that guarantees continuous access to agents regardless of their mobility frequency and patterns.

2.10 Conclusions

In this chapter we attempted to provide a comprehensive critical analysis of the domain of mobile software agents. We observed that the mobile agent paradigm possesses several important benefits over traditional approaches for distributed computing. The early years of the mobile agents were dominated by systems which were based on proprietary languages with frequently awkward semantics. Moreover these agent systems were rather primitive as they concentrated only on supporting code mobility while neglecting a number of important issues such as security, agent discovery, collaboration, etc. Contemporary agent systems are definitely more complete than their ancestors and provide a range of useful services to the agents themselves. However today's agent software is still based on specific programming languages and as a result agent technology is still hidden and has not reached the typical Internet end user. As the volume of agent systems increases significantly over the years, the issue of interoperability becomes more important and unfortunately with the exception of a few
standardisation efforts this issue has remained largely in the background of agent research. Finally a number of technical criticisms was presented regarding the mechanisms modern agent systems offer to support agent communication, collaboration, discovery, access control, navigation and mobility management. These points of criticism form the set of requirements for the design of the agent architecture in this thesis. An additional requirement of the work in this thesis is to decouple the agent architecture from the idiosyncrasies of a specific language in an attempt to bring agent technology closer to the typical Internet end users. This would maximise the agents’ beneficial impact on society and ensure that agents are not just a hype but a useful concrete software artifact that can enhance significantly the efficiency in general-purpose, distributed computing systems.
Chapter 3

The MASIC Agent Architecture

3.1 Introduction

In the previous two chapters we identified two sets of requirements that a generic agent framework must satisfy. The first set specifies the general issues that determine the adoption level of the agent framework by the industry and individual users. In summary, like any other software, the agent framework must be integrated with the existing technology and distributed applications, and must be designed in such a way that it can interoperate with other agent frameworks. Portability is necessary because of the Internet’s heterogeneous nature. The framework must also be generic so that it can be applied to a range of problem areas (applicability) in each of which it should exhibit clear gains in programming efficiency in contrast to other approaches. Finally the framework should possess a consistent, simple interface via which the user can monitor, program and control the framework’s state and execution (programmability and usability).

The second set of requirements defines the minimal set of issues and services that an agent framework has to address and offer, respectively, in order to support multi-agent societies. These requirements stem either directly from the definition of the autonomous mobile agents or can be inferred from the analogy between human and agent societies. This analogy is natural from the inception of intelligent agents in the early 70’s as a way of modelling and replicating the human behaviour and intelligence in computer software. In this chapter we will attempt to design a novel framework that addresses these issues efficiently.
Specifically section 2 presents the overall conceptual design of the novel framework (agent facilitator and agent system architecture). Section 3 evaluates an early abstract model of the agent facilitator that was produced to satisfy the aforementioned requirements. Finally section 4 discusses the final model of the agent facilitator and its components. Chapters 4 and 5 of this thesis analyse the concrete design of the agent system architecture according to the conceptual analysis of section 2 of this chapter.

3.2 Conceptual Design of the Framework

As was analysed in the previous two chapters, a typical agent framework consists of two components: the agent facilitator and the agent system architecture. The agent facilitator acts as an interface between the agent and the agent system. Its purpose is to provide services and functionality on a per-agent basis. The agent system as a whole provides services and functionality on a per group or society of agents level.

The first decision we were faced with was which design approach to follow in defining the framework. In chapter 2 it was shown that an architecture-based approach often provides better integration, interoperability and portability compared to a language-based approach. However the latter offers better programmability and can still show a good degree of integration if the language the framework is based on is already widely adopted and used. In order to make this decision we took into consideration two important characteristics of the Internet and the existing software landscape. Firstly the main characteristic of the Internet is its heterogeneity and versatility. Secondly there are already a large number of legacy software and multi-agent systems fully developed and deployed on the Internet. Because of these two facts, the issues of integration, portability and interoperability automatically become more significant than programmability. Moreover, an architecture-based agent framework can balance out the loss of programmability by providing a robust, consistent graphical interface via which the users can visually program, monitor and control the framework without requiring to learn a new programming language. The coupling of learning/mastering a programming language with the use/management of agents in a language-based agent framework limits the potential number of Internet users to those who have had extensive programming experience. Because of these reasons it was decided to adopt the architecture-based approach in designing our agent framework.
The next step is to identify the conceptual layers of both the agent facilitator and the agent system. The framework in total has to comply with the second set of requirements identified in chapter 2. The main challenge of this design step is to decide which component of the framework (agent facilitator or agent system or both) should offer which services. The only guidelines followed for these decisions are that: a) the agent facilitator offers services on a per-agent basis whereas the agent system offers services on a per-group basis and b) that an agent society's structure and functionality, as defined in the current literature, follows closely the model of the human nature.

The first requirement is that of information processing and storage. Like any human, an agent needs the ability to process and store information. An agent is to either remember and carry out user-defined plans or possess skills, (i.e. code) and information to offer to other agents on demand. In the first case the agents exhibit active behaviour while performing their user-defined duties whereby in the second case they are passive and only respond to satisfy service requests from other agents. Such requests can be made by both active agents or passive agents answering a service request from yet another agent. The aforementioned activities the agents to process and store proprietary information. These proprietary information and skills define the state of the agent and partly determine its individuality. Therefore the agent facilitator is the component of the framework that must be assigned the task of offering information processing and storage facilities on a per-agent basis. Figure 3.1 shows the agent framework after this decision:

![Figure 3.1. Framework after step 1](image)

The second requirement is communication. A solitary agent needs to communicate with its environment in order to get some input or return results of its processing to its user. In a multi-agent society agents must be able to communicate with each other in order to exchange information and requests for service. Communication can only be
realised if the agents have agreed on a common communication protocol via which they can interact. By definition, this means that communication is a service that needs to be uniform across the multi-agent society. The communication service supports a set of global, unchangeable protocols which do not differ on a per-agent basis. Therefore this service forms the first layer of the agent system. In order for the communication service to be accessible by all agents, each agent needs to be equipped with an interface that encapsulates the services the communication layer of the agent system offers. Figure 3.2 shows the framework design after this decision:

![Framework after step 2]

The third requirement for our framework is the provision of mobility. Mobility is the most important characteristic of software agents and it is the one that gives the agent technology a strategic advantage over other approaches such as distributed objects. Agents can move physically from machine to machine thus bringing the computation closer to the data in order to reduce the communication traffic. Therefore the provision of a mobility service is absolutely essential in agent framework.

The fact that agents are mobile gives rise to the problem of mobility management. Since agents are mobile their mobility may adversely affect their accessibility. The solution is mobility management: a service to be offered to agents on an agent-society scale. It forms the second conceptual layer of the agent system. This layer has to be above the communication layer since the process of maintaining the current location of an agent requires specialised communicating components. Clearly this layer would not be required if we assumed that the agents were static rather than mobile entities.

Mobility is not just the ability to utilise some form of transport provided by the agent system. It includes the capability to determine the route and conditions of travel, which is
termed navigation in agent literature. While on the move, the agent will expect the agent system to ensure its continuous accessibility.

Similar to the communication layer, the mobility management layer must hide its implementation details from the agents, thus the agent facilitator is also equipped with a mobility interface. Figure 3.3 shows the framework after this design decision:

![Figure 3.3. Framework after step 3](image_url)

The fourth requirement is the agent organisation and discovery. Like a human society, an agent society has to have some form of organisation via which agents form groups according to their needs and services. In order for the agent organisation to have any utility there has to be a query layer above it via which the agents can use the agent organisation in order to discover and access other agents which offer pertinent services. By definition the agent organisation and discovery service imposes some logical grouping to the flat set of peer agent facilitators. This grouping is then used by individual agents to discover/access other agents efficiently according to certain search criteria and preferences. Therefore this service should be represented as a layer in the agent system above the communication and mobility management layer. However because agents should be allowed to access other agents directly provided that they know their addresses and possess enough authorisation, this layer should not completely encapsulate the existing layers of the agent system. Figure 3.4 shows the framework after this design decision:
The fifth requirement for our framework is collaboration. As was explained in chapter 2, there are two possible ways of providing this service. Firstly by adding it as a layer into the agent system. The argument for this decision is that collaboration is another service similar to agent organisation and discovery. In frameworks following this approach the agent system is given the task of supporting collaboration among agents by providing them with whiteboards, tuple spaces or other information sharing facilities.

Collaboration is a form of agent organisation whereby more than one agent is involved in performing a task. While it requires various forms of group communication such as broadcast, multicast, poll, etc. these are not significantly different from the communication capabilities required for other forms of activities in agent communities. Therefore it is our position that the agents themselves should be encouraged to collaborate by utilising their communications capabilities. Another argument supporting our position is that the basic act of collaboration is rather simple and easy to represent in any programming language. A collaboration point can be modelled as a tuple \( <\text{producers, synchronisation, action, consumers}> \). There is a set of producers acting concurrently, each of which produces a set of data. Whenever a producer finishes its task it submits it to the collaboration point. The \textit{synchronisation} condition of the point is
evaluated and if it yields false then the newly arrived data are stored temporarily to await the arrival of further data. When enough data has arrived then the synchronisation yields true in which case the stored data is passed to the action for processing. The results are then sent to each member of the consumers set. The definition shows that the act of collaboration involves communication of data, synchronisation of these data and then execution of an action and routing of the results. A collaboration point uses the communication interface to receive data and route the results. Because the collaboration point also contains an action it is important to verify that all the arriving data is authorised to participate in this collaboration point and therefore causes the execution of the action. Therefore it is natural to place the collaboration layer immediately above any security layers of the agent facilitator.

In a typical collaboration point, its action field specifies the invocation of a service of an agent. If the collaboration service is realised in the agent system then this means that there are specialised agent system components that offer this service to the agents. Therefore the collaboration point is not necessarily co-located with the agent whose service is to be invoked when the action field is executed. Thus separating the collaboration points from the agents means that the entities that gather the collaborative data are separate and perhaps physically remote from the agents that perform the processing of the data. So not only would placing the collaboration service in the agent system require the existence of specialised entities performing the data gathering, but it would also increase the communication traffic, since these entities need to pass the gathered, synchronised data to the appropriate action agents. In contrast the approach of appending the collaboration service as another layer of the agent facilitator means that every agent can act as both data gatherer and processor since the data are gathered and synchronised at the place of their processing.

The last but not least requirement for our framework is the access control. Access control occurs at agent and group levels. Individual agents should be able to act autonomously regarding security matters and be allowed to specify and customise their own access control requirements. At the same time it is imperative that a central authority should be capable of specifying and enforcing a common set of ground access control rules to the agents. Without such rules the managing and enforcement of security policies to groups of agents becomes very difficult. For these reasons, our framework represents the service of access control both within the agent facilitator and the agent system itself. The access control layer of the agent system is placed above the existing
layers so that it acts as an umbrella for all the services offered by the agent system. This makes our framework secure because: a) all the layers of the agent system can use the access control layer to set-up and enforce their proprietary security policies and b) each agent can define and manage its own security policy on top of any ground policy specified by the agent system. Figure 3.5 shows the final conceptual architecture of the framework after the design decisions for the collaboration and access control services:

Figure 3.5. The final conceptual model of the agent framework

3.3 The Initial Abstract Computational Model

A cell [4, 5] in the context of our system is a novel representation of a resource or a group of resources. For example, it can represent users, groups of users, a file or a group of files, devices, processes, relationships between users etc. Thus, cells can be used to represent a system at an appropriate level of granularity.
Each cell is uniquely identified by a name, which is assigned to it the moment the cell is created. The body of a cell consists of two parts: the \textit{data} and the \textit{function}. The data part is a private memory area which contains the data necessary to support the functionality of the cell, for example local variables, intermediate results, etc. These items constitute the state of the cell. The data field of a cell is only accessible by the \textit{function} of that cell which is executed every time that cell is \textit{activated}. Hence the function part of a cell, among other tasks it performs, serves as an interface between the cell’s data (and state) and the other cells (figure 3.1).

Each cell has an interface (i.e. the set of commands it can accept and perform), which comprises a standard part and a user-defined part. The standard cell interface supports instructions common to all cells, e.g. self-destruct, rename, etc. The user-defined part depends on the functionality being provided. For example if a cell represents a file, then these instructions may be supported: read, read all, append, overwrite, change title, etc.

Cells can communicate with each other by means of exchanging \textit{tokens}. A \textit{token} is a structured message conveying information that consists of a \textit{header}, a \textit{data field} and a \textit{security field}. The purpose of the \textit{header} is dual: first it provides the information to uniquely identify that token and its context, such as its owner and the computation thread it belongs to, and, second, specifies the token’s destination which, naturally, is just another cell. The Identification part of the header is necessary for the \textit{authentication} of the token, hence this information is assigned to each token by internal secure mechanisms of the proposed system.

The \textit{data} field of the token contains the information that the token conveys. This information includes the \textit{method identifier} of the method to be executed at the destination cell, the arguments or part of the arguments for that method, etc. The data is structured within the field by means of a data definition language such as the ASN.1 notation. This is necessary in order for the recipient cell to be able to perform type checking on the data it received before this data is fed as arguments to the requested method.

The \textit{security} field of the token is reserved for use by the security mechanism of the system. It can contain security information, in encrypted form, that is required by the system in order for the token to be authorised.

The activation rule of this model is a simple string matching rule: a cell is activated when the destination name of a token matches the name of that cell. When a cell is activated, it consumes the information provided by the received token, uses and/or updates the contents of its data field and then produces a new token or tokens. Thus the
functionality of a cell can be described as a *token transformation*. As can be observed, this matching rule is the same as the activation rule in the static Dataflow paradigm [11, 12, 36, 37, 99].

The cell is the only persistent entity of the system. Figure 3.6 shows its simplified internal structure. As can be observed, each cell contains two input token queues: the activation and the authorisation queue. The activation queue receives tokens that are requesting the invocation of a method within that cell. The authorisation queue receives tokens that contain security information. Based on this information it will be decided whether each of the tokens arriving at the activation queue is authorised to invoke the action they require.

The authorisation tokens are the results of the activation of an authorisation graph. If the criteria for a token are matched, then this token is forwarded to the *authorised_token_queue* to be fetched and processed by the *match* process. Otherwise, a token is generated and forwarded to the output queue of the cell in order to be sent to the sender of the unauthorised token informing it of this event.

![Figure 3.6 A simplified view of a cell](image)
When a token successfully passes the authorisation stage, it is then forwarded to the *authorised token queue* (ATQ). This queue acts as a regulator between the *auth* process and the *match* process thus enabling these two processes to operate concurrently until the queue becomes empty. The *match* process consumes tokens from the ATQ, matches them with the tokens held in the Token Store (TS) and, based on its method prototypes table, it determines whether all the data necessary for the activation of the method are present or not. If the token being processed completes the data required for the activation of a method then this token together with the appropriate tokens from the TS are forwarded to the *method activation queue* (MAQ). Otherwise, the token is placed in the TS to wait for a match.

After the matching stage, the next phase of the cell processing is the *exec* process. The *exec* process consumes sequentially the activation requests from the MAQ and activates the requested method. Each method is expressed as an arbitrary graph by means of a *meta language*. This meta language should enable the programmer to connect into a data dependency graph independent modules implemented in any programming language. The interaction between the cell's methods and data is done via a storage API which can be referenced by any of the cell's methods.

Finally, upon termination, the method sends any results produced back to the *exec* process. The *exec* then packages the results into token(s) and places them in the output queue in order to be collected by the server and forwarded to the appropriate cell.

### 3.3.1 Program representation

From the previous section it is clear that the main abstraction of the model is the notion of cell. The cell's methods are functions, through which the user can access and observe/modify the program's script and state. Thus these functions define an interface for the user program interaction. By default typical methods that are included in each program cell upon creation are:

- Functions that manipulate the program's script. For example: insert code, append code, remove code.
- View status.
- Execute.
- Terminate.
- Input. Acts as input when program is executing.
• Output. Acts as output. These two I/O functions can be called by the program itself during execution. They form its I/O interface, and they hide the implementation details from the main program logic.

Cells represent heavyweight, largely static entities. The overhead they impose is large for lightweight, frequently travelling programs (agents). For this reason, small programs that perform simple repetitive tasks over a number of cells are efficiently represented as tokens. Accordingly we shall now extend the concept of tokens with a new variety: the program tokens. A program token (or token agent), in contrast to program cells:

a) Cannot exist and execute independently. A token agent is always executed in the context of a cell it is visiting. It is similar to the concept of applets in JAVA™. The current host cell is the manager of the token and it is responsible for its execution.
b) It generally has smaller size.
c) It migrates frequently.
d) It has no access control of its own, being subject of the access protection mechanisms of the host.

The structure of a program token is as follows (figure 3.7):

As can be observed, it consists of three parts:

1) Navigational Program (NP). The NP is responsible for the navigation of the token inside the network of cells. Upon arrival at a cell, it is activated and determines the next destination(s) of the token, if any.

2) Computational Program (CP). The CP is responsible for performing the required computation at every cell the token visits.

3) Data Sharing. This can be implemented as:

a) Shared variables, which are used for exchanging information. The current host cell of the token implements a semaphore for each shared data structure. Access or modification of any data is done via the appropriate semaphore.
b) Exchange of virtual tokens. The two programs can communicate by sending tokens to each other through the In & Out queue of the host cell.

c) Signals. If NP and CP are different executing processes, then they can exchange signals, provided that they possess the appropriate signal handlers.

Based on the complexity of their different parts, TAs can be classified into the following categories:

1) Simple Tokens. These are messages with a specific predefined destination and data.

2) Routing Tokens. These tokens contain fixed data but the destination field is a NP which is executed at every cell to determine the next hops, if any. In the simplest case, the NP can be:
   - A list of destinations.
   - A routing path of the form: X/Y/Z... This means that the token will go first to cell X. This cell, according to its routing table will perform a transformation of the head of Y/Z producing a new path, for instance Y1/Z. Then the token will go to cell Y1, etc.
   - Another routing path with variables. Assuming that variables start with a $ then the following path:
     
     \[ a/b\$x/\$x/f\$y/\$y \]

     is translated as follows. The Token goes to cell a. There the head (b\$x) is matched and for instance results in instantiating \$x with c. Hence the remaining path becomes:

     \[ c/f\$y/\$y \]

     The token is forwarded to cell c where pattern f\$y is matched yielding for instance \$y=d. Finally the token will be routed to d which is its final destination. A routing path can be further extended with recursion and selection constructs, thus making it adequate to express simple navigational logic.

3) RP Tokens. These tokens have a fixed destination, but their data field, instead of carrying passive data for a remote method invocation, is a program (CP) that will be executed at the destination. This is similar to the remote programming concept.

a) Agent Token (TA). These token are fully fledged agents since they comprise both a NP and a CP.

\footnote{An example of such style is found in WAVE \[108\]. We decided not to research this issue further.}
These different types are distinguished by a flag in the token header. This flag allows the host cell, upon reception of a token, to identify its type and thus process it accordingly.

From the above analysis, it is clear that each of the two program representations has advantages and drawbacks. Selecting one is a matter of efficiency and user choice. In general a program consists of a set of heavyweight entities (lengthy, complex algorithms) together with a set of lightweight entities that perform simple repetitive tasks. Such a program in this system is represented as a set of cells and token agents (figure 3.8):

![Figure 3.8 Program representation](image)

Existing systems with similar capabilities consider the cells (static objects) as higher entities and the tokens (mobile objects) as lower order entities that serve the cells by retrieving and filtering information. So the user is forced to organise the program into cells and tokens statically, thus not taking into consideration any dynamic conditions which might make this organisation inefficient.

By contrast, our model states that both entities are equal. Furthermore, just as a cell creates and sends tokens, a token agent can request the creation or one of more cells. Effectively a token could create a cell, copy its contents into it and then destroy itself. Such a procedure allows a token to be transformed into a cell and vice versa. A possible reason for such a decision is, for example, that a token has become large in size and it is not efficient any more to remain a token. This capability enables the programmer to use the logic that controls dynamically the representation and organisation of the program.

### 3.3.2 Program cooperation

Programs can co-ordinate their activities by means of synchronisation points. Each cell can define and use an arbitrary number of synchronisation points in their token store. A synchronisation point (SP) can be defined as the following tuple:
< ID, Attributes, Expression, PendingData >

where:

ID: An SP is identified by a name. Programs can use a synchronisation point by referencing its name.

Attributes: SPs possess certain characteristics, some of which are:

Lifetime: An SP can be temporary, which means that it will be destroyed when the program which has created it terminates. SPs can also be permanent, in which case their lifetime is independent of the lifetime of their creator. The lifetime of a temporary SP can be defined in terms of: the number of invocations, real time, etc.

Scope: An SP can be global or local. A global SP is visible by the whole network of cells. Any cell or token agent residing in a cell, can access such an SP. On the other hand, a local SP is only visible by the program that has created it. In a more general scenario, one of the attributes is actually an ACL defining a more complicated access control policy.

Colour flag: An SP can participate simultaneously in many synchronisations. Each synchronisation is independent from the others, hence their data must be distinguished and stored separately from the data of other invocations. This is achieved by colouring each datum.

Expression: This is the main part of an SP. The expression, in PROLOG terms, is just a predicate whose body is of the form: \(<condition,action>\). Both parts are arbitrary PROLOG compound goals. The semantics is that if the condition part of the SP expression succeeds then the action part is executed.

PendingData: This is a temporary storage for the data of the different synchronisations.

3.3.3 Evaluation of the abstract model

The abstract model presented in this section represents the first attempt to define an agent facilitator (cell) that follows the specifications defined in section 3.2. The presented model has the following characteristics:

- Efficient program representation. The model defines two types of agents, the lightweight (tokens) and the heavyweight (cells). The existence of these two program representations enables the programmer to define computations as an arbitrary mixture of tokens and cells. Therefore, in contrast to existing approaches that provide a single program representation, the programmer in this model can
classify and map efficiently his computation on a set of collaborating cells and tokens.

- Mapping to the conceptual model of the agent facilitator. The model follows closely the conceptual architecture of the agent facilitator as defined in section 3.2. The cell has a communication interface (input and output queue) via which it can communicate with other cells. A newly arrived token passes first the access control layer, then the synchronisation layer (token store and match process) and finally it performs an action on the cell's state by means of invoking the cell's program (method).

- Absence of navigation and mobility logic. The current cell architecture does not specify a component to store and manage its navigation logic. Furthermore the mobility interface is not defined. Therefore the cell's architecture has to be modified such that its state contains the navigation logic and its functionality uses the service of mobility.

- Asymmetric security. The cell only enforces access control for its input. In this way it can control which agent can access which services. This facility is not adequate. For example the owner of the cell might want to specify a security rule which prohibits the cell from using the service of other specific agents because either these services are too expensive or the specified agents belong to a malicious or not trustworthy user. This rule is by definition an output access control rule and cannot be either represented or enforced by the cell's input access control facility. Therefore in contrast to existing approaches the cell's architecture has to be extended to include an output access control facility. The arguments for not having output access control are that:

1. output control rules can be represented as input control rules of the destination agents. This argument is wrong because it assumes that if user A does not wish to communicate with user B then user B does not want to offer any services to user A.
2. the skills (programs) residing within the cell are created by the user owner of the cell and therefore he can embed his output control rules within the programs themselves. However this form of output control is inefficient because these rules have to be present in every program the owner inserts in the cell and unmanageable because if the output control rules change then all the existing programs within the cell have to be updated to reflect these updates.
Multithreading. The current architecture of the cell is multithreaded. The cell consists of three concurrent threads (authorisation, matching, execution) that exchange data (tokens) in a sequential fashion. The cell's internal queues act as token flow throttles to control the flow of tokens from one thread to the other. Each stage in this pipeline is responsible for processing each token according to its current state which is the lock table for the auth thread and token store for the match thread and updating their state if the token carries a command to do so. If the token does not carry a command to modify/access the state of the current stage then it is forwarded to the next stage. If the token is a token agent then it carries a program and hence such a token cannot be inspected by the auth and match process as to whether its code requires access to their respective states. Such tokens are executed in the exec stage. However because of this fact, there is a serious limitation in this multithreaded design. For example, it is possible that an executing program in the exec stage needs to access and modify the state of the access controller or the token store. In this case the auth or match thread, respectively, must switch to this request. Race conditions can occur because the executing program in the exec stage can specify a modification to the access controller. It is possible though that before its request gets satisfied, the access controller has already authorised and forwarded other tokens from the input queue, which would have been rejected if the state of the access controller had been updated first by the program in the exec stage. These conditions must not occur because they impose security risks. They can be avoided by making the design of the cell single-threaded.

Based on the above points of criticism, the final functionality of the cell (agent facilitator) can be described as follows: the cell is a single process that processes the incoming messages one at a time. Each message passes several stages sequentially until its request or program (token agent) is executed. The navigator component provides a repository for the user-defined tasks and navigation logic whereby the Information Store stores the current skills and information about the agent. In the remainder of this chapter the structure and functionality of the cell's main components (i.e. tokens, token store, IS, Navigator) will be presented. For ease of reading the description of the input and output access controller will be presented with the design of the security policies layer of the agent system in chapter five.
3.4 The tokens

The tokens are messages exchanged between cells. The types and structure of every type of token defines the high level communication protocol among cells. From the existing literature on communications protocols, a typical message contains a header and a data field. The header contains information about the message, i.e. its ID, its owner agent, its type, etc. The data field contains the information the source agent wants to convey to the destination agent. As shown in section 3.3 the tokens in our framework are extended to incorporate a security field which can contain security information the access controller of a cell uses to determine whether this token has enough authorisation or not. This security information has to be kept as a separate field because special operations might be required to be performed on it. For example, the source agent might wish to apply some form of encryption to this information because of its confidential and sensitive nature. Furthermore, the input access controller of a cell should be the only entity in the cell which is allowed to decode and inspect the contents of this field. The contents of the security field of a token must not be passed as input to a program of the Information Store because they might be copied or altered. At this point it must be emphasised that the functionality of a cell is predefined and constant and hence it can be trusted, whereas this trust must not be assumed of the programs the owner of the cell have incorporated in its Information Store. Because of these issues, the security information of a token must be kept and treated as a separate field within the token.

To design the structure of a token, recall that our framework in general must be adaptable by a wide range of users. By definition, since the framework is architecture-based the tokens in this framework must be defined in a declarative notation. The chosen notation must be simple, flexible and widely available. All these constraints are satisfied by the XML [80]. It is a simple, yet powerful tag-based notation emerging as the de facto standard for data modelling and interchange over the Internet. Using this notation a generic token can be represented as a simple XML text file which contains three tags, each mapping to one field of the token. Because the aforementioned three fields, header, data, security, make up a token the three XML tags representing these fields are embedded as sub-elements within a main element called token in the XML file. Figure 3.9 shows the generic token in XML:
By applying the same technique of mapping sub-fields of the token into sub-elements of the XML file we can represent any token in XML. Furthermore by using the XML DTD (Document Type Definition) facility the framework can define and use different types of tokens each of which have different structure and elements. If there are more than one type of token then each token needs to have an attribute to indicate its type so that the receiving agent knows how to process it. The use of the Document Object Model (DOM) facility gives us the added benefit of type checking. If we assume that each agent knows the DOM model of all types of tokens then when it receives a token, it firstly identifies its type and then matches its structure against the DOM model for that type. If the token does not have the right structure then it can be rejected. This facility gets even more important if we assume that there is an interface via which the users can interact with the framework, i.e. send/receive tokens to agents, etc. In this scenario when the user attempts to send a token to an agent, the interface can have the DOM model of the corresponding type and thus do type checking on the token before the token’s departure. This can save communication because an error-containing token will be detected and rejected before it travels unnecessarily to the destination agent.

The necessary fields for the header are the following: a) the type of the token, b) the source and destination address of the token. The destination address is necessary so that the communication system can route the token to the appropriate destination. The source address shows the originator of the token and is important in order for the input access controller of the destination agent to determine the authorisation criteria for this token, c) its ID via which the source agent can recognise any replies to this token, d) two flags to indicate whether this token requires any reply and/or synchronisation (more on this flag
in section 3.6) and e) any action to be taken if an error condition occurs during its processing (i.e. the token is rejected for security reasons by the destination agent). This sub-field provides the flexibility to specify alternative destinations for a token in case it can be processed in the current agent. The presented set of sub-fields for the header is not exhaustive. It serves only to identify the fields that are necessary for the operation of the framework in its current form. The type field allows for future extensions of the header of the token.

The tokens can be classified into two categories according to the type of information they carry in their data field. The token can be a simple invocation token in which case its data field has to specify the service it wishes to invoke at the destination agent together with the necessary data. The supplied argument values are tagged with the name of the corresponding argument of the specific service as specified in the service's prototype. Furthermore the token can be a token agent in which case its data field carries the actual code of the program to be executed at the destination agent. In this case it also needs to specify the language the program is written in so that the receiving agent can find out which interpreter to run in order to execute this program.

In the case that the token contains a program then the token’s data field, apart from the program text, can contain other data items the program might need during its execution. These data items are naturally represented as tagged values. In XML these tags are mapped into elements or attributes. This set of data items comprises a state that the program wishes to use when it is executed. When a cell receives such a token then its responsibility is to execute this program if the whole token is authorised. The main process of the cell forks the appropriate interpreter to run the token’s program. During its execution the program is given a simple interface via which it can access and use the services of the cell. This interface also allows the program to access/modify the contents of the data field of the token in which it resides. The cell process remains alive and receives requests/send replies from/to the token program process via a simple inter-process communication mechanism such as pipes. Among the services the program can access is the mobility interface of the cell, thus it can request its migration to another cell. When the cell receives such a request, then it terminates the execution of the token program, sets the destination field of the token to contain the new address, appends its own address in the source field of the token and finally places this new token in its output queue to be forwarded by the communication system. At the destination cell, the token
program is restarted from the beginning (weak mobility). It then uses its state (data items in the token's data field) to determine how to proceed.

The services visible to a token agent can vary and hence we can have a number of variations from the previous case. For example if the service of mobility is not accessible by the token program then this means that the token cannot request its migration to another cell. Therefore the token program will be executed only once in a single cell. Such tokens can be called Remote Program Execution tokens (RPET) and in fact can be used to perform simple remote programming. Such variants can be distinguished by appending a separate type attribute for the token's data field.

3.5 The Information Store

The Information Store (IS) holds the agent's proprietary information and skills. Agents, if they have enough authorisation, can request access to the contents of the IS of another agent. The design of the IS must satisfy the general requirements of the framework. Specifically there are three issues pertinent to the design of the IS:

a) integration. Because the agent's facilitator state does not depend on any specific language, therefore its state, part of which is the IS, must be able to incorporate skills and information represented in different languages.

b) usability. It must be straightforward for the users of the framework to view, manage and use the contents of the IS of the agents, and

c) programming efficiency. This issue is not only related with the program representation and execution schemes of the framework but also with the information representation strategies. Efficiency in the context of information representation means that agents should adopt a uniform information representation scheme that enables them not only to exchange information but to share information as well. In the previous section we showed that an XML-based message passing can be used for efficient exchange of information and commands among agents. The key in addressing the second issue (information sharing) is the notion of re-usability. One of the reasons for the wide adoption of object-oriented systems is that object-orientation offers an integrated, elegant mechanism for information sharing between objects, the notion of inheritance. In this approach the definition of new objects is based on the contents of the existing ones. Therefore the concept of re-usability is inherent in
object-oriented systems and thus readily available for object-oriented, language-based agent frameworks. On the other hand the incorporation of re-usability features in an architecture-based framework is not straightforward.

The key idea in solving this problem is to view each item in the IS of an agent as an independent black box that can be copied from one agent to another, assuming that the enforced security policies are satisfied. Each such black box has a name for identification purposes, some contents, an interface via which these contents can be used/managed and an access control policy that its potential users must satisfy in order to be allowed to access it. The notion of an independent black box maintains the principle of encapsulation in object-oriented systems since its contents are accessible only via its interface. The access control policy is a necessary component of a black box because it is a movable entity which is not aware of either the characteristics of its current host cell or the characteristics of the agents attempting to access it. The only protection such an entity can have is, upon its creation, to be hardwired with an access control policy. It should be possible though for some users to be allowed to reprogram a black box’s security policy at a later stage.

From the above analysis it is clear that the IS becomes a repository of a number of black boxes. The IS must provide an interface via which other agents can access the current black boxes, create new ones or request that a specific black box is copied or moved to another agent. The command of copying black boxes from one cell to another is probably the most important of the whole IS interface since it enables agents to use the contents of other agents as the building blocks of their own contents. Thus it is possible that a user can create an agent which is initially empty. Gradually though the agent can start having some specific state by copying black box components from other agents, assuming it is allowed to do so.

According to the contents and functionality, the black boxes can naturally be classified into three categories:

- **Data.** The contents of a data box are a set of passive data items. These contents can be expressed in XML for the same reasons as we used with a token’s contents. The box’s interface contains a number of interpreted programs which, when invoked, are provided with a standard interface from the cell via which they can access/modify the contents of the black box. This interface also allows the executing method to invoke another method of the same box. For every method the black box needs to contain the method’s name, the language the method is
written in, the actual code and the names of the arguments the method requires as input. The executing method is not allowed to invoke the method of another black box because this would destroy the independence characteristic of the black boxes. If it was allowed then it would be possible for an agent to copy a black box only to find out that in order for it to work it required access to another black box which was not present.

- Program. A program box is structurally identical to a data box. There are only two differences: a) the content of the program box is the code of the actual program and b) by default a program box comes with the method *execute* which if invoked, makes the host cell responsible for executing the program of the box. The *execute* method’s arguments are effectively the arguments the program of the box requires as input whenever it is invoked. The user that creates a program box can also specify more methods which, if invoked, show or modify parts of the program’s code.

- Agent/intelligent. An intelligent black box is a natural extension to the notion of a program box. The contents of this box are again a program together with a set of data items in XML format that comprise its state. The main difference between a program box and an intelligent/agent box is that when the program of an intelligent box is executed then it is given an interface via which it can access its state (XML elements) and can also access other black boxes in the same IS. An intelligent black box is thus a box that can utilise the contents of the IS it resides in, in order to perform its task. Furthermore the interface provided to an intelligent black box allows it to request its migration to another cell. The difference between a token agent and an agent black box is that the former is executed upon its arrival at the destination cell whereby the latter upon its arrival is placed in the IS of the destination cell. A token from an authorised agent is required to re-activate the newly arrived intelligent black box by invoking its *execute* method. Because of their interaction with the contents of the IS of the current host cell, intelligent black boxes can on the one hand perform their tasks perhaps more efficiently than program boxes but on the other hand might not be able to complete their task if they are placed in a cell that does not contain the necessary black boxes.
3.6 The Matching Store

As was explained in chapter two, the collaboration service is an important characteristic/utility that an agent framework should support. In section 3.2 it was shown that our framework provides an important new abstraction: collaboration points within the agents themselves. The Matching Store (MS) component of the cell is responsible in maintaining and managing an arbitrary set of collaboration points. The basic structure and functionality of a collaboration point was shown in section 3.3. An important difference between the proposed framework and other existing systems is that our notion of collaboration point is more general and powerful than the collaborative facilities in other systems. Firstly most of the existing agent frameworks provide only basic synchronisation facilities which enable a number of agents to synchronise their execution. The implementation of these synchronisation points is by means of a semaphore which is given an initial value and each of the synchronising agents, after it reaches a specific point in its program, decrement this value. When the semaphore reaches zero, then the agents are allowed to continue their execution. Such facilities can only support synchronisation of programs which does not include any data exchange and processing. Secondly Aglets and Concordia do attempt to provide more generic collaboration facilities. For this service they define independent specialised agents called synchronisation objects which offer the necessary functionality to support the collaboration among other agents. The synchronisation objects contain a set of synchronisation items each of which has a name, a synchronisation expression and an associated action. The collaborating agents must know the address of the synchronisation object and the name of the synchronisation item in order to perform the action of collaboration. The synchronisation expression of a synchronisation item contains the names of all the agents that must send data to this item in order for its associated action to be executed. Typically the action part of the synchronisation item just specifies a method of an agent to be executed using the gathered data as input.

Our definition of collaboration points is more generic and can incorporate the functionality of the existing approaches as well as support more collaboration patterns. Specifically, as it was shown in section 3.3 the collaboration points in this framework form a conceptual layer all incoming tokens in a cell have to go through. The new collaboration patterns supported by this framework and their importance are discussed below:
- **Dataflow-style.** The dataflow model of computation provides an elegant way of representing programs as asynchronous dataflow graphs [13, 15]. In such a graph the nodes represent single, low-level instructions and the arcs indicate the flow of data among these nodes. The functionality of dataflow graphs is based on a simple node activation rule that inherently contains the notion of synchronisation [82, 89]. More specifically it is defined that a node in the graph is activated and the relevant instruction executed if and only if all necessary data for the instruction is available. This rule is important because tokens in a dataflow graph are synchronised automatically according to their contents and the ID of the program they belong to. This matches exactly our framework’s approach to place the collaboration phase before the execution phase of a token’s processing in every cell. The dataflow rule can be applied at a higher level in our framework for tokens carrying a single service request (agent method invocation in object-oriented terms). With this technique tokens from different agents can go through an implicit, automatic collaboration act that the programmer does not need to define explicitly.

The dataflow-style of collaboration is implemented as follows in our framework: when a token arrives at a cell and passes the authorisation phase it is then submitted to the Matching Store of the cell. The Matching Store examines the data field of the token and identifies the name of the method the token wishes to invoke. If the specified method exists then the Matching Store uses its prototype signature to determine the names of the arguments that method requires as input. The token then is checked as to whether its data field carries all the necessary arguments. If it does then the token is forwarded to the execution stage. If it does not then the Matching Store creates automatically a dataflow collaboration point. The contents of this collaboration point are as follows: a) its synchronisation expression contains the required argument names for that method in an AND boolean expression. This represents the fact that in order for this point to be executed it has to have received values for all the specified arguments, b) its action field contains just the name of the method to be invoked, c) its name contains the ID of the program this collaboration point participates in. This ID is part of the header of every token, and d) the already arrived token is placed in the collaboration point’s storage. When a new token belonging to the same program arrives at the cell wishing to invoke the same method then the Matching Store checks whether the arguments the new token carries in its data field together with the ones already present (from the previous token) satisfy the synchronisation expression. If they do the Matching Store passes the
invocation request to the execution phase. The results of the invocation are sent to the union of the destinations the collaborating tokens have specified. Otherwise, if the present arguments are not sufficient then the new token is stored with the old one to await the arrival of more tokens. The presented novel collaboration mechanism is simple but suffers from one problem. Since all tokens pass through the collaboration layer, a cell cannot distinguish between tokens that carry partial data because they wish to participate in a collaboration act from the tokens that carry partial data because of an error. To solve this problem we define a synchronisation flag as a necessary sub-field of every token's header. If the flag is set then this indicates the token's willingness to go through the collaboration phase. Otherwise the token indicates that it is independent from every other. In the latter case, if the token is found to contain less that the required data for the method invocation it specifies, it is discarded and an appropriate error message is sent to its originator.

- **User defined.** The data flow style of collaboration can be applied when the programmer wishes to define a single, existing method of a cell to be invoked as a result of a collaboration act. There are cases though where the programmer wishes to specify more complex actions to be performed as a result of the activation of a collaboration point. For this reason we define the user-defined collaboration pattern. The structure of such a collaboration point is identical to that of a dataflow collaboration point with the difference that the action field of the point now contains a user-defined interpreted program to be executed. The synchronisation expression contains an AND boolean expression of all the arguments required as input by the program specified in the action field. As before, when all these necessary arguments have been assigned values by incoming tokens then the program is executed. The cell provides an interface to this program via which the cell’s IS contents can be accessed/used/modified. The name field of such a collaboration point contains a user-defined string that uniquely identifies it in the local Matching Store. In order to participate in the activation of a user defined collaboration point, a token specifies in its data field the name of the point rather than the name of a cell’s method (in the dataflow style). The existence of user-defined collaboration points gives the collaboration layer two strategic advantages over other models: a) in other approaches if the user wanted to define a collaboration point in the context of our framework, then he would have to first incorporate the program within the IS of the agent and then define a synchronisation point whose action invokes the
The aforementioned program. In our framework, this scenario is implemented in a single integrated step, and b) After the activation has occurred the programmer will have to explicitly remove both its program from the agent and its synchronisation point from the appropriate synchronisation object. In our approach this procedure takes place automatically in a single agent where the user-defined collaboration point was defined.

In conclusion, the notion of user-defined collaboration points can be used to efficiently create, use and discard collaboration points that invoke arbitrary user-defined programs. The efficiency stems from the fact that: a) the gathering of data and execution of action occur within the agent whose services the program will utilise (minimum communication) and b) both the creation and deletion of such a point do not involve any change in the IS of the agent which is used for storing more permanent information items.

- Agent collaboration points. The aforementioned patterns of collaboration can be used by service request tokens. However, as shown in section 3.4 the tokens can also be the carriers of lightweight agents (token agents). Token agents cannot use the previously described collaboration patterns because they carry a program rather than a single invocation request. In this case it is not possible for these programs to define in advance which method (service) of the cell they wish to invoke. The decision and invocation of appropriate cell services takes place at run-time only. The question that naturally arises here is how to specify that a number of arbitrary programs (token agents) wish to collaborate and what the possible semantics of such an act are. Existing approaches claim that the only semantics that can be given for such collaboration is that of simple synchronisation. According to this approach a prespecified number of programs access a semaphore entity when they reach a certain stage in their execution. Every participating program accesses the semaphore and reduces or increases its value. Then the program checks if the new value of the semaphore is equal to the prespecified value. If not then the program remains blocked until enough participating programs arrive such that the semaphore's value reaches the threshold. In this case the programs are awakened and can continue their concurrent execution. From this description it can be observed that the action associated with this synchronisation is that the participating programs can resume their concurrent execution.
The approach followed in this framework is a natural extension of this. An agent collaboration point synchronises the execution of a number of token agents, referenced by name in the point’s synchronisation field, and results in their execution according to a prespecified sequence in the point’s action field. Our approach is more generic and flexible because: a) the synchronisation condition specified can be an arbitrary Boolean expression over the names of token agents. Thus the only option the existing systems give which is an AND expression of specified agents becomes just a special case in our approach, and b) the action specified can define the sequence (or randomness) according to which token agents will be executed in the cell after they are synchronised. This enables the programmer to define the order with which his token agents will interact and thus change the state of the current host cell. In this way the programmer can ensure the integrity of the semantics of his program.

Token agents can indicate that they are willing to participate in a collaboration by setting their synchronisation flag. It is also necessary that the state of the token agents specifies the name of the collaboration point they wish to activate. The token agents are responsible for defining the value of this attribute (represented naturally as an XML element) prior to their migration to a new cell.

Because of the reasons mentioned in sections 3.4 and 3.5 all forms of collaboration points can be efficiently represented in XML. The type of a collaboration point is indicated by an XML attribute. Each field of a collaboration point is represented by an appropriate XML tag (element). Each of the three forms of collaboration in total is represented as an XML file that follows a different XML DTD. The synchronisation expression of the collaboration points is a simple string representing the appropriate boolean expression. The action part is either a string (if it specifies order of execution of the token agents or if it specifies the cell service to be invoked in the case of dataflow-style points) or the text representation of the program to be executed in the case of user-defined collaboration points.
3.7 Agent Navigator

One of the primary characteristics of mobile agents is their ability to migrate autonomously from host to host [64]. Thus, support for agent mobility is a fundamental requirement of every agent infrastructure. Agent mobility has been addressed by various systems [46, 52, 58, 101, 117, 123]. There are two distinct design approaches for agent mobility based on which there is the separation of strong from weak mobility.

Strong mobility is implemented by incorporating within an interpreted programming language the most primitive navigational action of migrating to a new single physical location. The agent in this case is a single monolithic program whose computational logic is fully integrated with the navigational logic. When the interpreter encounters the navigational command, the execution of the agent is interrupted, the whole state of the agent’s execution thread is captured and subsequently transmitted to the remote destination the agent specified [46]. Upon arrival the system resumes the execution thread thus the agent resumes execution from the point of interrupt.

Strong mobility schemes have two important drawbacks: a) Language dependency. They are usually based on a new, proprietary language that the programmer is required to learn in order to program agents. [108, 121], and b) Homogeneity. Being based on a single, global language for agents, these schemes assume that the language’s execution system is distributed and made available in all the hosts of the network. The homogeneity assumption though does not hold for the Internet which is inherently a heterogeneous environment.

The alternative approach to strong mobility is weak mobility. This approach is based on the separation of navigation from computation. The navigation logic of an agent is typically encapsulated in an itinerary object which contains a set of 3-tuples <destination, action, next>. Each such tuple defines where the agent must go (destination), the method of the agent (action) that must be executed upon its arrival and the next tuple to be executed after this current one (next). Tuples of this form can be hardwired with each other to form navigational patterns, such as sequential traversal, parallel traversal, spawning agent replicas that perform a specific task and return the results to the parent agent, etc. These patterns can be embedded within each other thus allowing the creation of more complex regular navigational expressions or itineraries. Existing systems implement each navigational pattern as an object therefore a regular navigational expression is represented as a rigid object hierarchy. As a result an agent
travel plan is inherently static and predefined by the agent programmer during the agent’s creation phase. In a continuously active and changing environment though, like the Internet, it is not sufficient to provide agents with static travel plans. The AJANTA system has no support for dynamic modification of itineraries whereas IBM’s Aglets argue that they implement dynamic travel plans but in reality they only support the addition/removal of a plan item at the bottom/end of the itinerary. Furthermore Concordia is based on the assumption that a change of plan takes place only in exceptional cases [83, 123].

In this section the agent Navigator is presented and analysed. The agent Navigator is an active entity embedded within an agent and it supports the creation and dynamic modification of the agent’s itinerary.

3.7.1 Agent Navigator: An overview

Like the computation logic, the navigation plans are equally an important part of an agent’s state. Therefore in contrast to the Concordia approach they should be embedded within the agent itself. The agent Navigator is an active component of an agent and its main functionality is to a) store and manage the agent’s navigational data, and b) implement the agent’s navigation plans. The agent Navigator consists of three basic elements: the plan items, the Navigator State (NS) and the Application Programming Interface(API) as shown in the following figure (figure 3.10).

For the following analysis, it is assumed that the agent consists of two concurrently active parts: the agent Navigator which is responsible for implementing the agent's tasks and the communication process which is responsible for receiving messages from other

![Figure 3.10 The agent Navigator](image_url)
agents/users. These messages are either kept in a queue to be processed after the end of the currently executing navigation plan if they contain requests from other agents, or sent as interrupts to the Navigator if they contain navigation control commands such as pause, resume or terminate.

An agent’s Navigator contains one or more interconnected plan items. A plan item is an autonomous, pseudo object whose contents are expressed in a declarative notation. It carries an identity tag that uniquely identifies it in the local Navigator. The fact that the plan item is represented as a named set of declarations supports the cross-referencing of plans or components of plans. For instance, in contrast to existing approaches, the executing plan item can reference by name other plan items within the local Navigator and use/modify their components (run-time modification of travel plans), create new plans, delete existing ones (if they are not referenced currently by any other plan items), etc.

The functionality of a plan item is similar to that of a simple_itinerary in Aglets. Each plan item specifies an action to be executed at a specific destination as well as the next plan item(s) to be executed after the current one. The action field of a plan item, rather than referencing a method of the agent (existing approaches), embodies instead a program written in any interpreted language for which a secure interpreter is available. The plan item becomes in this way autonomous since it does not depend on the contents (methods) of the agent it resides in. This autonomy is useful because it enables the copying of plan items among agents without altering their semantics or functionality.

The plan items in this architecture contain more fields. For example the field data stores the results produced from the execution of the action. This provides an added benefit over the existing systems because the plan items can directly exchange and share results simply by referencing each other’s name and field name. Therefore the plan items do not rely on the local agent to provide any proprietary storage facilities that could be shared by them. Section 3.7.2 discusses the plan item’s structure more extensively.

The Navigator State is a private information store of the Navigator. It contains certain permanent information characterising the Navigator itself such as the ID of its owner, its version, etc. This information can only be viewed by other agents or executing plan items but it cannot be modified. The main data item of this component is the plan stack which is used to store a number of regular navigational expressions (as defined in section 1) that must be executed in a last in-first out (LIFO) fashion. More details of the plan stack are also presented in section 3.7.2.
The agent Navigator implements an API through which other active components of the agent and in general other agents can access/use the agent's navigation logic and data. The API contains a set of methods that enable the currently active plan item to a) access/modify the components of other plan items in the same Navigator; b) define/use synchronisation points in the agent's token store, presented in section 3.6; and c) use the communication facility of the agent to interact and use the services of other agents. Similarly, this interface enables the agent itself to a) access/modify its navigation plans (plan items); and b) control the execution of the Navigator process at run-time if it has received one or more navigation control commands from the network. Moreover, the interface also offers methods that enable other agents to copy, move or create plan items in the local Navigator assuming that such requests are authorised by the access control policy the agent enforces.

### 3.7.2 Plan Items

As already mentioned, a plan item represents the pre-specified action or actions that the Navigator will have to execute on a specified location, and then hop to another host or terminate the agent's trip. It comprises the following fields:

i. The **name** of the plan item, used for cross-reference and reusability purposes as described below.

ii. A set of **attributes**, where the plan item keeps some information for private use i.e. the plan's item ID, the owner's ID of the plan item.

iii. A **data** field, where the plan item keeps data acquired during its execution.

iv. The **location** field, which holds the host name, where the agent has to go in order to execute the plan item.

v. The **action** method, that holds the series of actions that the plan item has to execute on the specified location.

vi. The **next** field, which holds a pointer to the next plan item(s) the Navigator executes next.

The benefit of individually naming each plan item is that every plan item can access individual components of another plan item. This way, if a user wants to reuse a plan item in a different Navigator he can cross-reference an already pre-defined one. Furthermore, a component of a plan item can be linked to the same component of another plan item. Hence, if the user/agent changes one plan item, then automatically all plan
items referencing this plan item will implicitly be updated. The deletion method of a referenced plan item is similar to the UNIX directory deletion operation. A reference counter is used and the plan item will not be deleted unless the counter is zero.

Given that plan items are reusable components, once defined, they can act as pseudo-objects and be copied or accessed by other plan items or Navigators. Let us assume that the user creates a plan item—called "stores_check"—to check if a specific product exists in a shop's warehouse. The product ID is defined as a variable and stored in the attributes field. Because of this the programmer can define a new plan item that utilises the existing computational logic of the first plan item, to look for another product at another location.

Another advantage is that the action field can be implemented in different programming languages for each different plan item, thus giving a language-neutral approach. Consequently, the action field will hold the language's name the code is written in and the actual code that has to be executed upon arrival at the required location. It is the programmer's responsibility to ensure that the required compilers/interpreters will be available on the location of execution. Moreover, the plan item being a dynamic entity can modify components of itself or of other plan items during execution of its action. The programmer can use the API to access and/or change other plan items. For example, at the aforementioned example, plan item "stores_check" looks for a product at the stores of a shop and the item is not in stock. Let us also assume that there is another plan item—called "supplier_check"—which checks if a supplier has in stock a product. During its execution, the action of the first plan item ("stores_check") can change the value of its next field to point to the second plan item ("supplier_check"), since the requested product is not in stock. Therefore the plans in this architecture are adaptive and thus more efficient than static plans supported by other systems.

In the simplest case the next field of a plan item points to a single plan item. However, it can also represent navigation patterns. The set of patterns that are utilised in this model was identified by the AJANTA and Aglets models, and are the following:

* **sequence**  $\text{SQ} = (\text{plan\_item\_1, plan\_item\_2, ..., plan\_item\_N})$. The agent has to execute each plan item in sequence.

* **set**  $\text{SET} = (\text{plan\_item\_1, plan\_item\_2, ..., plan\_item\_N})$. The agent has to traverse through all plan items in an undetermined order.
• **split** \( \text{SPLIT} = (\text{plan\_item}_1, \text{plan\_item}_2, \ldots, \text{plan\_item}_N) \). The agent creates \( N \) replicas, each of which executes one of the specified plan items. Upon their birth the children agents act as autonomous agents and their behaviour is independent of their parent's behaviour. In this navigation pattern the children agents are not expected to report any results back to their parent.

• **join** \( \text{JOIN} = (\text{plan\_item}_1, \text{plan\_item}_2, \ldots, \text{plan\_item}_N; \text{SP\_ID}) \). The agent creates children of the plan item and dispatches them concurrently at the specified destinations. It has to provide them with a synchronisation point ID so the results can be collected (described in more detail in Section 3.7.3).

Additionally, each *next* field can include regular expressions in its context. A regular expression will be a combination of the aforementioned patterns. For example, let us assume that the *next* field of \( \text{plan\_item}_1 \) includes: \( \text{next} = \text{SQ(} \text{plan\_item}_2, \text{SET(} \text{plan\_item}_3, \text{plan\_item}_4) \text{)} \). In this case, the Navigator has to execute first \( \text{plan\_item}_2 \) and then the pattern \( \text{SET(} \text{plan\_item}_3, \text{plan\_item}_4) \). In order to do that, it copies the whole regular expression to its Navigator State and then executes \( \text{plan\_item}_2 \). Therefore, an important difference between the presented architecture and the existing approaches is that regular navigation expressions are represented as strings which is more efficient and flexible than the object hierarchy representation adopted by the IBM Aglets and Concordia.

The problem that arises here is that one or more of the plan items referenced in the above expression can, in turn, contain another expression in their *next* fields. For example, if the *next* field of \( \text{plan\_item}_2 \) has another regular expression -- \( \text{SET(} \text{plan\_item}_5, \text{plan\_item}_6) \) -- then the Navigator will have to execute this expression without losing the rest of the travel plan. In order to maintain consistency, the Navigator keeps a stack of the regular expressions that have to be executed. The Navigator's State before and after the *next* field of \( \text{plan\_item}_2 \) is depicted in figure 3.11. We can see that the next plan item for execution will be either \( \text{plan\_item}_5 \) or \( \text{plan\_item}_6 \), then plan\_item\_6 or plan\_item\_5 respectively and then \( \text{SET(} 3, 4) \).
Since \( \textit{next} \) is language-neutral, we can introduce breakpoints and give the user more control over the execution of the travel plan. Specifically we implement "PAUSE" as a keyword which if it appears in the \( \textit{next} \) field of the current plan item, the Navigator execution is paused. Until a "RESUME" or "TERMINATE" message has been received, the agent has to wait\(^2\). Thus, by inserting breakpoints, we can create collaborating agents that execute a computation in parallel and co-operate by pausing, exchanging results and resuming. For instance, imagine two agents which collaborate. The first one – after executing his action – has paused and has to wait for some results from the second one. When the second finishes with its execution sends the result and a "RESUME" command to the first one, and then both continue their execution.

A problem that can appear is the asynchrony between the two agents. That is, the results and the command "RESUME" from the second agent arrive before the first one has paused. This situation can be solved by introducing a stack that keeps "RESUME" or "TERMINATE" messages arriving at the input queue of each agent.

An additional advantage of our model is that a plan item can be easily represented with XML as shown in figure 3.12. Hence, by representing a plan item with a human

---

\(^2\) The commands PAUSE, RESUME and TERMINATE belong in the set of standard methods of every cell
recognisable form in standard notation, applications can be created, where the user can
directly generate and inspect his own travel plans.

3.7.3 The Split/Join procedure

One of the most important and frequently appearing navigational patterns is the split-
join procedure. In a typical split-join scenario the agent creates \( n \) replicas each of which
is assigned a specific task to perform. The replicas behave as normal agents but at the
end of their task, the results produced have to be reported back to the parent agent. The
parent agent processes the results and decides what further task to execute next (if any).

From the above it is clear that an important part of the split-join process is the
synchronisation and processing of the results from the children. In MASIC agents are
equipped with a Token Store facility which can be used in this context as well. Briefly, a
Token Store is a collection of Synchronisation Points (SP). An synchronisation point in
the context of navigation is defined as a tuple \(<ID, expression, action, store>\) where:

- **ID** is a string uniquely identifying this synchronisation point in the local token store
- **Expression** A Boolean algebra expression over the names of children agents. For
example if an SP's expression field contains the string “A and B or C” then the **action** of
the SP will be executed if either the results from both children A, B are received or only
the results from child C.
- **Action** The action field of the SP contains a program written in an interpreted language
which takes as input the results from the children agents and produces as output the
navigation pattern the agent Navigator has to execute next.
- **Store** Whenever the results from an agent arrive indicating the **ID** of this SP, then the
new result's agent name together with the contents of the **store** are checked as to whether
they satisfy the **expression**. If the **expression** yields false then the new results are placed
in the **store** awaiting more results to arrive.

The proposed agent Navigator supports the split-join pattern, like other related
systems. In our system this procedure is implemented as follows:

**Step 1:** The currently executing plan item creates via the API \( n \) new plan items with
names \( plan_1, plan_2, ..., plan_n \).

**Step 2:** It also creates a synchronisation point with a specific ID.

**Step 3:** The parent's next field contains the string \( JOIN(plan_1, plan_2, 
..., plan_n:SP\_ID) \) where \( SP\_ID \) is the ID of the synchronisation point.
Step 4: The $n$ agents are created. In each of them there is a field in the Navigator State indicating where the end results should be submitted. Specifically, it contains a tuple of the form $<\text{parent\_agent\_address}, SP\_ID, results\_tag>$ where the $\text{parent\_agent\_address}$ is the address of the parent agent, $SP\_ID$ is the ID of the synchronisation point to be invoked at the token store of the parent agent and $results\_tag$ is the tag that has to be appended to the results and will be used as input to the expression of the synchronisation point.

Step 5: The child agents execute the appropriate plan items.

Step 6: At the end of execution they submit their results to the parent agent invoking the synchronisation point $SP\_ID$ and appending the data with the $results\_tag$.

Step 7: After the expression of the $SP\_ID$ yields true the action is executed to produce the new navigation pattern the parent agent wishes to execute next.

In conclusion, the advantage of using the external token store (which is still part of the agent's state) is that plan items can define and use generic synchronisation points while the Navigator's structure remains simple without the need to incorporate any extra synchronisation facilities. Furthermore, the synchronisation points defined in this way are persistent and can be used repeatedly by different plan items.

3.7.4 Conclusions

The support for agent mobility is one of the primary services that an agent architecture has to offer. Existing systems offering weak mobility support only the definition of static, precompiled travel plans that cannot be modified dynamically. In this section a novel agent Navigator facility was presented. Its main components, the plan items, are autonomous, named lists of declarations that a) can cross-reference each other thus allowing for efficient itinerary representation and b) can be used as re-usable components that can be copied to other agents (navigation knowledge sharing). The plan items are offered an API via which they can access and modify the contents of other local plan items at run-time. Furthermore it was shown that breakpoints are introduced to enable the user to control the execution of his agent dynamically. In conclusion the presented agent Navigator can be used to efficiently represent re-usable, dynamic travel plans which can be created and inspected by the user directly through the use of an XML viewer.
3.8 Discussion

In this chapter two main objectives were achieved. Firstly the overall conceptual model of a novel language-neutral agent framework was designed. Secondly the internal architecture of the agent facilitator (cell) was presented and analysed. The main requirements for both designs were set in chapter two and will form the basis for the evaluation of the designs.

Two important requirements of the proposed framework as a whole were integration and interoperability. These requirements have been taken into account in several parts of our design:

- Firstly the framework itself does not depend on any proprietary programming language or environment.
- Secondly the declarative parts of the various components of the agent facilitator are represented in the standard notation of XML.
- Thirdly the active parts (programs) of the agent facilitator were assumed to be represented in any interpreted language for which there is a safe interpreter. This does not pose any serious restriction since there is a number of such interpreters widely available such as Tcl, Java, etc.
- Fourthly the framework as a whole supports language-independent weak mobility rather than language-dependent strong mobility.

The usability/programmability of the framework is also robust because of the presented agent facilitator architecture. Specifically because there is no underlying restriction for the interpreted programming languages used to create and insert items in the IS of a cell and also because of the XML representation of the cell’s declarative parts, it is straightforward to create a GUI via which end-users can interact with the framework. The process of creating a cell should be a matter of simply filling an online form. The process of updating the state of a cell should be a matter of selecting already existing programs from libraries and incorporating them in the IS of the cell. Tokens can also be created by online forms. Furthermore since a cell is represented in XML it should be possible for the user to view directly the contents of the cell and perhaps invoke a specific service by utilising the GUI. The GUI can also include a graphical copy paste method that enables the user to select a component of a cell’s state and paste it to another agent, assuming that the source and destination access control policies are enforced.
Another important metric for our framework was re-usability. Distributed object systems achieve re-usability through inheritance. Architecture-based frameworks, being language-neutral, do not immediately support any inheritance or indeed any other re-usability mechanism. Our proposed framework solves this problem by viewing the contents of agents as independent, copiable components. New agents can utilise and copy (if they are permitted) the components of existing agents.

Hierarchical structuring is also important for program representation and execution. Our framework provides a range of structuring facilities through the agent facilitator. Firstly, in contrast with existing frameworks, the agent facilitator incorporates a hierarchy of characteristics. Cells are rather heavyweight agents, tokens can be the carriers of lightweight agents (token agents) and intelligent black boxes contain programs that interact with the local host cell and can jump at their will to another cell. The programmer can use any mixture of this rich repertoire of agents to represent his program efficiently. Cells can represent persistent agents that do not migrate frequently, token agents can represent temporary highly-mobile sub-programs that visit a number of hosts and perform simple, perhaps repetitive, tasks and intelligent black boxes can be used to represent persistent services that utilise the contents of their host cell. Secondly, each agent facilitator has onboard collaborative facilities that the programmer can use to efficiently co-ordinate the activities of his multi-agent program. These collaboration points reveal the co-ordinated structure of the invocation of simple cell services. Collaboration points can also be structured into a distributed dataflow-style graph which can act as the control mechanism of a set of multiple-type agents representing a single user program.

The presented novel agent facilitator also provides some additional benefits to our framework in comparison to other systems regarding access control. Firstly in our cell there is support for both coarse-grain access control via the access controller of the cell and fine-grain access control because of the fact that the components of the IS can have their own access control policies. The access control in total is symmetric and enables the user to define both input and output control policies. Furthermore the cell itself has a trusted, predefined functionality and acts as a trusted mediator in the interaction between incoming tokens and internal active or passive components.

However the aforementioned benefits do come at a price. Our proposed framework suffers from two main disadvantages: firstly the lack of static type checking. Being an architecture-based framework, it offers no embedded mechanism for type checking of the
arguments passed as input for the invocation of a cell's service (method). Because each cell is expressed in XML, the framework can still implement a primitive type system by assigning to every argument of a method a type label in the form of an XML tag. These labels could then be checked against the labels of the argument values that the incoming tokens contain. However this constitutes dynamic type checking with all its known disadvantages.

The second disadvantage is related to the procedure of executing active components within a cell, for example a method. As was presented earlier, in this case the executing program is given an interface via which it can access the contents of the data box it resides in. Every access request is received by the main process of the cell which consequently retrieves the requested data item and returns it to the executing method via an interprocess communication mechanism such as pipes. This procedure is necessary so that the main process of the cell ensures that every requested access is allowed according to the access control policies in place. But it is slower than the access to data by an executing method within an object in object-oriented languages. Although communicating across pipes is less fast, the advantage is that different procedures do not have to share the same address space and hence certain security risks are avoided.
Chapter 4

Mobility management and agent discovery

4.1 Introduction

In a distributed mobile agent environment, the issue of locating and tracking of mobile agents and thus ensuring continuous access is of paramount importance. The first part of this chapter outlines the conceptual (high level) architecture of a novel, integrated addressing and routing system (Mobile Agents Routing Services Architecture MARSA) which provides a structured decentralised solution based on the notion of lightweight software routing/addressing nodes, the arbitrers [1, 5]. The system supports the creation of overlapping physical and logical hierarchies which define multiple access paths for each agent. Each such path is coupled with a distributed access control mechanism and can be customised on a per agent basis. Furthermore, specialised management agents (MARSA Management Agents) support the dynamic updating of the system's naming hierarchies.

Apart from mobility management, a multi-agent system must also enable the agents to discover efficiently other agents based on the services they offer or their characteristics (agent discovery). In the second part of this chapter it is argued that the agent discovery can be coupled with several aspects of the computation such as access control and customisation resulting in a better sharing, use and management of the information held by the agent discovery system. A novel architecture is presented in which discovery messages and discovery paths are mutable, active entities, which interact with each other as peers making the organisation system dynamic in nature. Queries traversing the system can be reformulated while at the same time the system itself can change depending on the nature and volume of the query traffic. Furthermore it is shown that the organisation nodes of
the proposed architecture can serve as re-usable components for building more complex, composite nodes from existing ones.

### 4.2 Mobility management background

Since the notion of software mobility is relatively new, a plethora of naming and addressing schemes available that could be used to resolve the issue of mobile agent tracking become inadequate because the concept of mobility is not taken into account in their design. Typical examples of such systems are: the Internet’s Domain Name System (DNS) [85] and the X.500 Directory Service [103].

Furthermore this issue, strangely enough, has not been given enough attention by the various state-of-the-art mobile agent systems or frameworks. Typical examples of such systems are: Odyssey [46], Aglet [58], ARA [101], Agent-Tcl [52], TACOMA [61]. Most of these systems either completely neglect the problem or provide an oversimplified inadequate scheme. Flat schemes do not scale well and strictly hierarchical ones suffer from serious inefficiencies. Furthermore, several models for locating mobile hosts such as [14, 73] can not be applied in the tracking of mobile agents [120].

Nevertheless, there is a number of proposals that attempt to resolve the problem in a structured manner. These systems are: the Globe location service [119, 120], Emerald [62] and the Location Independent Invocation (LII) [21]. Emerald is based on building chains of forwarding pointers, a concept which does not scale well to global networks, whereby the LII assumes that the update-to-lookup ratio is small, an assumption that clearly should not be the basis of a naming system in a mobile agent world. Globe on the other hand, is based on a simple physical hierarchy of dedicated servers which can not be agent-customised or tackle any agent privacy or security issues. Furthermore, it offers little extensibility and can only accept very few optimisations.

Agents in the context of the system proposed in this chapter are user or system entities representing data or programs that can communicate with each other by means of exchanging structured messages (tokens) [4]. Each agent is assigned an input and output queue. The agent reads its input queue for incoming messages and places in the output queue any outgoing messages. The system is assigned with the task of removing the pending tokens from the output queues of the agents and placing them at input queue of
the appropriate target agents. Each agent is given upon its birth exactly one Global Unique Identification (GUID) which depends on the location of its creation and remains constant throughout the lifetime of the agent. Each agent has a home system agent which is assigned to store the current address of the agent. The purpose of the GUID is to identify the home entity of an agent, which in turn guarantees to maintain the current location of the agent. Thus the GUID provides the primary way of locating a mobile agent.

4.2.1 Overview of the proposed system

In this section, an overview of the novel architecture of the system is given. A more detailed analysis can be found in [5].

The system is realised as a network of servers. Each server comprises a set of system level entities, the arbiters, and a MARSA Management Agent (MMA) agent. The functionality of a server is partitioned into a number of arbiters. Each arbiter represents a virtual place in Odyssey terms [46]. In this context every arbiter enforces its own security policies and provides a set of proprietary services to the agents visiting its virtual place (local agents). The main functionality of an arbiter is to act as a message-passing node that receives all the outgoing messages from the agents that visit its place and route them to the appropriate destination. Similarly the arbiter receives all incoming messages and delivers them to the appropriate local agent(s). The purpose of the MMA is to manage the set of arbiters in the local physical machine (server), improve the efficiency and speed of the GUID resolution system by registering all foreign agents currently visiting the local machine and support the dynamic reconfiguration of the servers hierarchy.

The servers considered in our system are not dedicated into merely implementing the naming system. Each server, however, operates in "dual" mode, acting as both a GUID resolution node and as a venue for visiting and creating agents. Agents can be created in any of the servers of the system, thus each server is the home machine for a number of agents. Upon its creation, an agent is given an input and output queue and is assigned an arbiter, called its "Home" arbiter". Home arbiters are responsible to keep a minimum state information for each agent they serve as also to keep track the location of the agents they serve. Similarly, when an agent moves and resides in some foreign system, it is again provided with an input and output queue and assigned to one of the local arbiters of the
foreign system, the “Foreign” arbitrer. The Foreign arbitrer is responsible for the forwarding of messages to and from the visiting agent and to inform the Home arbitrer of the agent about its current whereabouts. This is done by letting the visiting agent reveal its GUID and probably other details to the Foreign arbitrer through a registration procedure.

The GUID assigned to an agent is a structured string which can resemble an IP address. The servers in our system are organised into a hierarchy that represents their physical organisation. A message whose destination field is the GUID of an agent, travels through the hierarchy of servers until the current location of the target agent is retrieved.

As a result, in contrast with existing hierarchical naming systems, the GUID of different agents can vary in length, depending on which level in the server hierarchy, the home servers of the agents are located. For example, if we assume the tree topology of Figure 4.1, an agent created in server 2, has the GUID 1.2.arbl.id1 whereby the agent in server 4 has the GUID 1.3.4.arb2.id2

The immediate consequence of this organisation is that resolution of shorter GUIDs can be done in less steps than the resolution of longer ones. This property does not pose any serious problem because this difference is only experienced at most once when one agent tries to locate another. Thereafter, the source agent can use one of the provided shortcuts, which are discussed later, for any subsequent attempts to locate the same agent, thus bypassing the physical hierarchical resolution of the GUID.
4.2.2 Functional Specification of arbitrers

The conceptual architecture of an arbitrer (see Figure 4.2) comprises the following layers:

1. **Naming and arbitration layer.** The functionalities provided by this layer allow him to: a) remove pending messages from the output queues of the agents it serves, according to some arbitration policy, and route them towards their specified destinations, b) receive incoming messages destined for the agents it serves and distribute them accordingly and c) participate in the structured resolution of GUID addresses. In the most typical case, the arbitration policy can be a priority based scheme whereby agents with higher priority are served first and more frequently.

2. **Mobility management layer.** This layer gives the arbitrer the functionality to act as home router for agents. It implements algorithms for maintaining the most up-to-date location of its agents, possibly perform analysis on their travel patterns that can be used for possible probabilistic tracking of Mobile Agent etc.

3. **Logical layer.** This layer gives the ability to arbitrers to be connected into an arbitrary logical network. Each arbitrer can be assigned to represent a concept or relation. When an agent is created, it submits a profile of its contents or services that it can provide. This profile is injected into the logical network and it can be registered in a number of logically relevant physically distributed arbitrers [111]. Each of the arbitrers that accepts the agent profile becomes a logical home router for that agent and maintains a
pointer to the agent’s physical home router. The logical home routers of an agent act as aliases of the agent’s GUID, thus allowing the agent to be located and accessed by agents which are not aware of its GUID. The logical layer enables the arbitrer to route messages whose destination is a logical address rather than physical address (GUID). The structure of the logical address can be thought of as a list of strings separated with a delimiter, like for example a URL address in the WWW.

4. Services layer. This layer consists of the services the arbitrer offers to agents which are assigned to it. One important service can be message filtering. Each of the agents being served by this arbitrer can customise this filtering. For example an agent can specify that it doesn’t wish to accept any messages from a specific agent or with specific content. The provision of a services layer to an arbitrer can transform it from a routing/naming node to an information centre for its assigned agents. Thus the arbitrers apart from partitioning the naming and routing service can also be the means of partitioning and distributing to groups of agents the general services the physical server offers to the agents currently visiting it.

The arbitrer interface comprises three parts. The first part is the set of commands that the local MMA can use to manage and enquire the arbitrer. The second part is the user interface which is used by the agents in order to invoke/customise the various MARSA user services. These services include the setup and customisation of filters, buffers for connection-oriented communication, mobility-related requests, etc. Finally the third part is the set of commands that other peer arbitrers can invoke in order to support the migration of agents as well as the resolution of the GUIDs.

4.2.3 Access paths

In general, each arbitrer is identified by a GUID. After all, an arbitrer is just a system owned agent. Furthermore, an arbitrer is also registered and identified with an ID in the MMA agent of the server it resides in.

The problem of locating a mobile agent in this system is transformed into the problem of locating the physical Home arbitrer of that agent. There are four ways of performing this:

1. Via the full GUID, hence following the normal hierarchical GUID resolution method (indirect physical addressing).
2. By giving an address of the form:

```
serverIP.arbitrerID
```

With the second method, the message is sent directly to the server whose IP address is `serverIP`. That target server’s MMA agent matches the given `arbitrerID` with the list of local arbitrer IDs and forwards the message to the appropriate arbitrer. In a similar manner an agent can be accessed by using an address of the form:

```
serverIP.arbitrerID.agentID
```

where `agentID` is the ID with which the agent is known locally to the arbitrer that serves it. It has to be unique only within the specific arbitrer (*direct physical addressing*).

In this way, a message can be delivered to an agent with only two pattern matching actions: Firstly to find the appropriate arbitrer within the specified server (matching performed by the server’s MMA agent) and secondly to find the specific agent among the set of agents served by the specified arbitrer. This method bypasses the arbitrer physical hierarchy and views the system as a simple flat set of distributed arbitrers.

3. By using one of the logical addresses of the agent. (*logical addressing*).

4. By using a *hybrid* address. This scheme is a combination of indirect and direct addressing. Such an address has the following format:

```
DirectPart.IndirectPart
```

The first part is of the form presented in method 2 whereby the second part is of the form of method 1 or 3. With this scheme any node of the logical or physical hierarchy of arbitrers can be used as a starting point in resolving a hybrid address (multiple entry points). Parts of the hierarchical resolution of GUIDs or logical addresses can be bypassed while others can be maintained.

Access paths do not only provide shortcuts which can improve the efficiency and speed of locating a mobile agent but also provide a way of tightly coupling the addressing and tracking mechanism with privacy and general security requirements. More specifically, each agent can specify what security requirements a message has to satisfy depending on the originator of the message and the path it follows in order to reach the agent’s home router. Each path can be given different security requirements.

The agent can distribute the security requirements for a path as lightweight Access Control Lists (ACL) into the arbitrers that comprise that path. In this way, a message destined for that specific agent and following the specific path is scrutinised by each intermediate arbitrer to ensure that it satisfies the specified conditions before it is routed.
to the next arbitrer. The union of the security constraints of each arbitrer participating in an access path for an agent constitute the security policy that agent implements on that path. Hence the system supports secure access paths on a per agent basis. A typical application of this security system is to protect the privacy of the mobile agents. A mobile agent could decide not to be contacted by certain other agents when it is not currently in its home machine. Thus, for example, access requests for a specific agent can be denied, if the requests have passed through specific paths.

4.2.4 The MARSA Management Agents

The Marsa Management Agent (MMA) plays three important roles in this system:
Firstly the MMA participates in the resolution of the GUIDs and keeps track of the current visitor agents in the local physical machine. MMAs that reside in different machines can communicate with each other in order to perform the address resolution. Each MMA listens to a specific, well-known port in the local server and can access directly the network independently of the arbitrers.

It also acts as a central logical node that can be contacted by user agents to update dynamically the configuration of the local MARSA system. For instance the MMA agent can create new local arbitrers that implement different policies and offer different services. Moreover the network of MMAs makes the whole hierarchy of the GUIDs dynamic in nature since they can support the run-time addition of arbitrers and physical machines. An existing arbitrer can also be removed provided that there are no registered agents for which this is their home arbitrer. This is a reasonable restriction which is present in all major file management systems such as UNIX, MS-DOS, etc.

The third task for each MMA is to store and manage various policies and services that are common to all the arbitrers of the local machine. If a user agents, assuming that it possesses the necessary authorisation, modifies these policies/services, then the MMA is responsible in communicating these updates to all the local arbitrers. The arbitrers themselves have a predefined, trusted functionality which guarantees that they will enforce the common policies and provide the common services to their registered agents. An immediate application of supporting common policies among arbitrers is access control. For example the administrator of the local MARSA system might deem necessary to forbid any agents of a specific user from entering the local machine. The MMA provides a
simple interface via which the administrator agent can set and enforce this policy in all the local arbiters without the need for contacting each one of them individually.

The MMA interface commands can be grouped into four categories: a) create/remove arbiters/machines, b) maintain a record of the visiting agents in the local machine, c) perform name (GUID) resolution, d) manage the common policies/services. The first group of commands is used by authorised agents to request the creation/deletion of arbiters. It is also used by the MMAs of unregistered remote machines that wish to register and join the MARSA system. Such MMAs must first contact a registered MMA in order to be authenticated and be given a GUID. After this process these new MMAs can create their own, local, customised arbiters. The second category is used by the local arbiters to report the arrival/departure of a visiting agent. This information is important and must be stored by the local MMA as this is related to the GUID resolution functionality of the MMA. For example if a local agent wishes to send a message to a remote agent then the arbiter responsible for forwarding this message contacts its local MMA to discover the home network address of the destination agent. It is possible though that this agent currently is visiting the same machine where the originator agent is. Since the MMA is aware of the current visitor agents then it is guaranteed that the message will be sent to the destination agent without contacting the home arbiter of that agent. This is efficient because the act of communication was completed without any network communication. Clearly the MMA is the most suitable entity of the MARSA system to store this kind of information because such scenarios can occur with any of the local arbiters. The third category of commands is used by remote MMAs to request the resolution of GUIDs. For every such message the MMA uses its knowledge of the network addresses of its children machines and forwards the message to the appropriate MMA. If the requested GUID belongs to one of the local arbiters then the MMA places this message to the input queue of the appropriate arbiter. This category is also used by the local arbiters to request the resolution of GUIDs their agents wish to communicate with. Finally the fourth category is used by authorised agents that wish to view/update the active common policies/services of the local machine.
4.2.5 Evaluation

In this section, we presented a new infrastructure that enables the location tracking and transparent routing of Mobile Agents (MARSA). The mechanisms presented can be integrated with security and privacy protection mechanisms and enhanced with the provision of logical hierarchies to support dynamic discovery of agents based on the services they can provide to other agents. Our proposed scheme has theoretically the following properties:

- Guaranteed continuous access to the mobile agents.
- Realisation of logical addresses for each agent to support its dynamic discovery.
- Tight integration of the addressing mechanism with a general-purpose security mechanism which can be customised on a per agent basis.
- Provision of multiple entry points in the logical and physical hierarchies which can be used as a mechanism to bypass parts of the naming hierarchy.
- Integration of the addressing scheme with a decentralised routing mechanism based on the concept of Arbitrers and distributed caching.
- Arbitrers can be agent customised and offer higher level services to the agents they serve such as filtering of messages.
- The arbitrers have a layered structure which is open to future extensions.

However, the MARSA system is still in its infancy. There is no prototype implementation or simulation results to demonstrate its performance. This section presented only the initial conceptual (high level) architecture of the system. Therefore various important communication-related issues such as buffer management, communication deadlock and starvation, etc. have not been addressed. Clearly more research is required to transform this initial design into a feasible, efficient, general-purpose communication system for mobile agents.

4.3 Introduction to agent organisation & discovery

An agent community or society can be thought of as a group of specialised agents collaborating with each other in order to perform multiple, user-defined tasks. Similarly to the human society organisation, each agent offers specialised services that other members of the community can invoke and utilise. In such a system it is important for the agents to
be equipped with a discovery facility that enables them to publish their services and
discover other agents whose services are pertinent to their tasks. Therefore the agent
discovery issue falls into the broader category of information retrieval and resource
discovery.

Existing multi-agent or distributed objects systems such as Odyssey [46], Agent-Tcl
[52], Aglets [58], ARA [101], WAVE [108], Mole [116], CORBA [98] either ignore the
need for agent discovery completely or use a distributed database which acts as a
centralised agent directory. The agents can register a profile of the services they offer to
the database. Each such profile contains the address of an agent and a description of the
services it offers, the latter using some fixed notation. The database is simply a large
collection of profiles which agents can query to discover whether there are any agents
offering specific services.

From the aforementioned systems, CORBA is the only one to provide a
comprehensive framework for agent discovery. The CORBA framework is based on the
notion of specialised discovery objects called traders, which can be created by objects that
carry administrative privileges. User objects can access the traders to register the services
they offer to other user objects. Trader objects can be connected with each other by links.
An inter-trader link defines a unidirectional path between two traders. It is used and
managed by the source trader as the target trader is not made aware of the existence of
the link. Each trader can receive queries directly by user objects and according to certain
policies specified either in the trader or the query can utilise one or more of its existing
links in order to forward the query for further processing. When the query’s search criteria
are satisfied then a message is sent to the query originator object. The message contains
the CORBA name and the interface of the object that has been found. Policies specified in
the query can determine the number of objects returned as answers to the query as well as
their ranking according to certain criteria.

The literature on information retrieval and resource discovery [23, 33, 85, 112, 22, 54,
55, 110] describes solutions similar to those introduced with CORBA. Typically there is a
group of organisation nodes that form a hierarchy and are distributed over a number of
physical servers [23, 55, 71]. Resources, usually documents, register a summary of their
contents (in the form of an SQL query or a Small Prolog database) with the back end of
the organisation system [77]. The profile indicates what queries the specific resource (i.e.,
document) will satisfy. In the front end, the system accepts user queries in the same
notation that the profiles are expressed in and attempts to match the query against the current contents of the profile database [68]. Some of the systems require the user to guide them during the query-processing phase [23]. Others introduce automation in either the gathering and distribution of profiles or the processing of the queries [112]. In the first case the system deploys a number of profile gathering agents which visit the resources and create their profiles [68]. These profiles are then submitted to the leaves of the hierarchy of organisation nodes. Each node processes the profile and possibly forwards the profile in full or abbreviated form to the nodes one layer up the hierarchy. In this way, the nodes of layer n can utilise the profiles of the nodes in layer n+1, process them and then forward aggregated profiles to the nodes of layer n−1 [111]. In the second case the queries are automatically forwarded down the hierarchy until they reach the server where the resulting document is physically located [111].

In general the hierarchy of profiles/organisation information is typically managed/updated manually by an administrator. The administrator can perform its tasks by dispatching specialised agents that traverse the set of documents/resources pertinent to a specific knowledge domain and report back any changes that need to be reflected to the central organisation hierarchy [68]. Although this approach introduces a degree of automation in the process of information gathering, it suffers from the disadvantage that the agents need to have an a-priori, complete knowledge about their domain.

In general, the discovery of agents or resources follows the following simplified model (figure 4.3):

![Figure 4.3 Current approach to agent discovery](image)
According to this model, resource agents first register themselves with the organisation system (step 1). In step 2 a user agent contacts the organisation system in order to discover whether there is any agent offering certain services. The organisation system processes the query and sends the reply to the query originator (step 3). Finally, in step 4 the user agent that originated the query uses the results returned by the organisation system to directly contact the resource agent that has been found. In this simplified model of interaction, it is clear that the process of agent discovery (steps 1 to 3) is separated from the computation process (step 4). The main disadvantage of this approach is that an important component of the computation, namely access control, takes place only in step 4. The organisation system is based on the assumption that all the information it holds is public, hence it offers no access control services for its registered resource agents. This means that the user agent can acquire information about a resource agent which it has no sufficient authority to access. As a result, processing time and communication bandwidth is wasted for both agents.

The agent discovery process and access control process should therefore be properly integrated. Another advantage of integration would be the ability to use the infrastructure of the organisation system to organise, distribute and manage access control information. The organisation system would then be an active mediator between user and resource agents, with the process of agent discovery being concurrent with the access control process. The organisation system should ensure that a query does not proceed to the next phase of processing unless the owner(s) and the characteristics of the query satisfy certain authorisation criteria and constraints, respectively. At first glance, this approach seems to introduce additional latency as the interaction between any given user and resource agent has to be mediated by the organisation system. However, the impact of the extra latency is unlikely to be significant since the indirect interaction only takes place when a user agent attempts to discover and access a resource agent for the first time. At this stage (see step 4 in figure 4.3), the resource agent would additionally return the part of the interface that this user agent has the authority to access. No further assistance of the organisation system should be required from this point on. To summarise, in the existing approaches the search criteria of a query determine the interfaces the query can retrieve and use:

\[
\text{Interfaces} = f(\text{query})
\]

The new approach described in this chapter is based on the following novel equation:

\[
\text{Interfaces} = f(\text{query}, \text{security_policies})
\]
4.4 Agent organisation as name management

In general, an organisation/agent discovery system is essentially a name management system. In this section we will attempt to substantiate this claim by defining a generic abstract model of a name management system and observing any similarities between its structure and functionality and those of a generic organisation/agent discovery system.

Names are symbolic representations of abstract concepts or real objects. A namespace provides a context within which a name has a specific meaning (semantics). A name management system is a group of namespaces, possibly connected with each other and distributed over an arbitrary number of physical servers. Each namespace can be represented as a function \( f \) which takes a name into the value (meaning) of that name within the current context:

\[
\text{value} = f(\text{name})
\]

A name can be given different semantics in different contexts. This means that in general if \( f \) and \( g \) represent two different contexts then:

\[
f(\text{name}) \neq g(\text{name})
\]

A context can be thought of as a pair:

\[
<\text{namespace\_name}, \{<\text{name}, \text{value}>\}>
\]

where \( \text{namespace\_name} \) is the name uniquely identifying the specific context within the group of contexts of a particular name management system and \( \{<\text{name}, \text{value}>\} \) is a group of names each of which is assigned specific semantic value. Following the same notation the name management system itself can be represented as a pair:

\[
<\text{namesystem\_name}, \{<\text{namespace\_name,namespace\_address}>\}>
\]

where \( \text{namesystem\_name} \) is the name uniquely identifying the specific name system within a group of name management systems and \( \{<\text{namespace\_name,namespace\_address}>\} \) is a group of namespace names each of which is assigned specific namespace addresses. Therefore a name management system is essentially a mapping function from the group of names to the group of semantic values. The group of values represents the group of objects that have a physical manifestation, such as a data file, a program, a device, a user, etc., and operations that can be performed on them. Up to this point, the names in different contexts and the contexts themselves have been treated as disjoint, independent entities. Only two operations can be defined in such name management systems:

a) definition of a name within an existing context;
b) definition of a new context within the name management system.

However a generic name management system should be able to address at least two more phenomena:

a) the relationship between two categories of names and
b) the relationship between the semantics of two names in two different contexts.

The first case is important because naturally one category of names might be a subcategory of another category. For this reason a context should be allowed to register itself with a name in another context.

The second case is equally important because it incorporates the notion of re-usability. The semantics of a name in a context can be defined to be equivalent with the semantics of another name in a different context. Expressing equivalence relations among semantics of names results in a more centralised and thus more efficient management of a group of names. For instance if the semantics of $n$ names are associated with the semantics of a specific name then the updating of the semantics of this name will result in the automatic implicit updating of the semantics of all the $n$ names that "point" to it. There are numerous real world applications that have demonstrated the usefulness of such a scheme. For example in the World Wide Web, a company might create a simple Web page via which it offers various services to its customers. It can then register links to this central Web page in other pertinent pages or search engines. If the company decides to modify its services then it needs to modify only its central page.

So far the structure of the semantic space has remained somewhat abstract. The semantic space has to consist of two separate entities:

a) The external namespace. All the names defined in this namespace have externally defined semantics. This means that evaluation of such a name results in an action to be executed by the underlying operating system. Essentially the operating system acts as a local external namespace for the segment of the name management system which is installed in this physical machine. For example if the operating system is UNIX then the name `mkdir my_dir` in the context of this OS will create a new directory called `my_dir`. Clearly the external namespace forms the interface of the local segment of the name management system to the real world. Because the physical machines which the name management system is distributed over are assumed to run different operating systems it can be concluded that the external namespaces are local and variable. For example the
name is in the UNIX external namespace makes sense and it will result in a valid action, but in the context of MSDOS it would yield an error.

b) The constant namespace. Because the external namespace depends on the physical location of the name management system, we can not use this notion to create mappings between names and semantics that remain constant throughout the system. For instance, the value of name 2+3 may be obtained arithmetically, giving the result of 5, which is unchangeable and location-independent. The name management system has to ensure that such information remains constant within the system and does not depend on the physical location or will of possible malicious users. Figure 4.4 shows a diagrammatic view of the novel name management model.

![Diagram of the abstract name management model](image)

Figure 4.4 The abstract name management model

The main functionality of this model is to be able to find the semantics of a name, submitted as a query by a user. Because the contexts can be interconnected and because the semantics of the name in a context can be linked to the semantics of other names in other contexts, the process of finding the semantics of a name consists of several stages. At each stage the system finds the semantics of a name. If these semantics depend on the semantics of another name in another context then the system continues the evaluation process with the new name. Otherwise if the semantics of the current name are fully determined then:
a) if these semantics are an action (external namespace) then this action is performed and the evaluation process terminates, otherwise

b) if the semantics are a global constant then this constant is returned as an answer to the query. The names whose semantics in a specific context are defined either in the external namespace or the constant namespace are called atom names in that context because they constitute the points at which the name evaluation process terminates.

The existing organisation and agent discovery systems [23, 85, 98, 111] follow this model closely. They treat an external namespace as the group of resources the system manages, the contexts are represented as organisation nodes within which resources can link up, the organisation nodes are interlinked, typically forming a hierarchy. Because of this interlinking, the process of evaluating a name passes through several stages as well until the process reaches a context that is a leaf of the hierarchy.

However, the model presented in this chapter offers two benefits over the existing models:

a) centralised control of name semantics. This facility is useful when there are names which have to be assigned a global, permanent semantics. For such names the provision of a centralised registry to store global mappings is necessary. Existing systems do not offer such a facility. There are two possible reasons for this: a) as suggested by section 4.3 of this chapter the current mobile agent systems are primarily focused on providing code mobility and have neglected the issue of agent discovery altogether or have provided the simple and easily implementable passive "yellow pages", b) the names of resources is of the form of a hierarchical path that is guaranteed to be unique for every resource. In this way such systems claim implicitly that one or more resources can have the same name provided that they have registered this name in different name contexts. Clearly though this approach leaves open the possibility for the unauthorised and possibly malicious exploitation of registered names (including trademarks) of business services and products.

b) Evaluation of names is linked to actions. Existing systems are passive and assign passive information as semantics to names. In contrast the presented abstract model extends the notion of semantics of a name to include actions. Hence in this model, evaluation of a name can yield an action which will affect the external environment. The repeated evaluation of the same name could yield different results subject to the
constraints and rules of the external environment. For example all operating systems forbid the creation of a file with a specific name if another file already exists with the same name in the same directory. Therefore if the evaluation of a name in the proposed organisation system results in the creation of a file then repeated evaluation of the same name will yield an error message from the underlying operating system.

4.5 Requirements of a concrete agent discovery system

The abstract model presented in the previous section extends the generic model that many current systems are based on. However there are a number of more concrete requirements that a generic organisation/agent discovery system has to satisfy. In this section we will attempt to produce these requirements and match them against the current organisation systems in an attempt to prove that the existing approaches suffer from some further important disadvantages.

Adaptivity. In existing systems the organisation of resources remains constant over time. Typically the resources are organised into a hierarchy which is designed to be as generic as possible. The hierarchy is rigid in that it can not accept major structural modifications. In CORBA, for instance, user objects can not create their own traders or trader hierarchies. On the other hand, the organisation and service-discovery structure of an agent society must be adaptive. It should be able to change to reflect the current availability of services and service requirements of the society's members. Individual agents should be allowed to create their own organisational structures to reflect their own contents and services. These individual organisational structures could then form group organisational structures, etc.

A necessary condition for creating an adaptive organisation system is that there is no ontological connection between the agents and the organisation nodes. Agents should be allowed to register/unregister themselves from organisation nodes at will. The organisation nodes should be kept separate from the resource agents. In this way, any changes in the organisation structure will not have any side effects on the state of the resource agents themselves.

Interaction. In all existing systems the queries and organisation links are passive entities. A query defines a group of requirements. A link offers a group of services. The
organisation node where the link resides matches the query's requirements against the link's services and then the service interface is returned to the query originator. In this chapter we argue that links and queries should be active peer entities that interact with each other. This interaction offers certain advantages:

(i) A query can modify the state of a link and leave a message. This may be useful for other queries of the same or different program thread that attempt to travel through that link. For example a query could inform other related queries that it has passed down that link so that the others can avoid following the same route (avoid duplicate queries),

(ii) The link can modify the state of the query and make the query itself more specific. Certain attributes of the query might change, for example a certain number of credits can be removed from the query's wallet (in the case of electronic billing). The link could in general incorporate new information into the state of the query which might be useful for the further links that the query will follow. Note that this query-link interaction could be mediated by the organisation node for security purposes and in order to make the links of the organisation system sensitive to the nature and volume of the query traffic.

**Reusability.** In existing systems the links, which are the basic building blocks of the system, are not autonomous entities. Furthermore, the organisation nodes themselves cannot be used as building blocks to create new, composite organisation nodes. The reusability, and hence the efficiency, of the system can be improved by considering links as pseudo-objects. Apart from the benefit of encapsulation, the links will then become reusable components which can be copied from one node to another. Additionally, the existing organisation nodes can be used as re-usable components at a coarse-grain level to create new aggregated organisation nodes. Applying the concept of reusability to the organisation system can significantly reduce the complexity and speed up structural alteration of the system.

**Integration of agent discovery and access control.** The benefits of such integration have been analysed in section 4.3.

**Organisation nodes as high-level service providers to agents.** The organisation nodes of the system could offer high-level services to their agents: e.g., access to legacy systems, a group mailbox for the members, etc. In this way an organisation node could be
transformed from a mere service publisher to a central management node of a community of agents sharing the same interests.

Specialisation. An efficient organisation system usually requires specialised nodes. Each organisation node should represent a concept or characteristic that its members (resource agents) should possess. This is important because user agents can be made aware of these specialised nodes so that they may direct any queries to pertinent nodes rather than flooding the system with broadcast queries or searching through a number of nodes sequentially. To enforce specialisation, each organisation node should implement a registration protocol that resource agents have to follow before they are allowed to publish their services at that node. The registration protocol should ensure that the services a candidate resource agent wishes to publish satisfy the topic or characteristics of the particular organisation node. Existing systems in the area of resource discovery follow this principle but CORBA does not ensure that each trader is a repository of specialised interfaces (services).

4.6 Overview of the proposed architecture

In sections 4.3 and 4.4 we presented our view that organisation and agent discovery models are essentially name management systems. An abstract name management system was suggested and shown to have certain benefits over existing models. Furthermore, some requirements for an efficient, dynamic organisation system were delineated, which are not fully met by the existing systems. In this section, a concrete architecture for agent organisation/discovery is introduced. We shall discuss its compliance to the abstract model of section 4.4 and the extent to which it addresses the requirements laid out in section 4.5.

The novel features of the proposed architecture are as follows:

- Agents are organised into groups. Each group is represented as an agent called group manager (GM), thus both members of a group and the group itself are on the same level of representation. A group manager, described later in this section, acts as a proxy node performing aliasing of the names of the agents it includes.
- Groups can be organised into further groups by gathering their group manager agents. In this way, the agents' organisation can be of arbitrary depth and complexity.
• There is no ontological connection between the agent members of a group and the
  group itself, meaning that groups can be created and deleted to organise arbitrary
  groups of agents without destroying the existing members and bringing any new ones
  into being.
• The name of the group manager agent defines the group it manages as a set expression
  on groups. Other groups can be referenced in a regular expression by using the name of
  their group managers. For example, if A, B, C, D are GM names, then a GM with a
  name of:

\[(A \cap B) \cup (C \cap D)\]

represents the union of intersections. Thus new groups of agents can be created from
existing ones.

In terms of implementation, a newly created agent is assigned a unique identifier (tag).
A message whose destination field is equal to the tag of an agent, directly matches it and
activates the agent (after the message is authorised). In traditional systems, such as UNIX,
the tag of an object is the only means by which it can be represented. For instance, a
UNIX file would have as name a path starting from the root of the tree hierarchy
(absolute path) which uniquely identifies the file within the UNIX file system. In other
words, this path defines a unique access path to the object. Although further access paths
can be added by creating symbolic links to the object, those links are separate objects,
while the access path should be part of the original object state.

In the architecture described in this chapter, agents can belong to more than one group
and hence can have more than one access path. An access path acts as an alias of the
agent’s unique ID (address), which identifies an additional route, as a list of other agent
IDs that can be aliases as well, via which that agent is accessible. For each access path, an
agent can have a different lock expression locking it [7, 10]. Furthermore, the agent itself
can control which users can access it through which path. Any agent can customise the
security requirements for a specific path based on whether this path is trusted or not.
Existing systems neglect the importance of the route a message follows in order to reach
its destination and thus their access control policies take into consideration only the
originator of the message [98]. By contrast, in this architecture, an agent can define who
can perform which operation via which access path. In this way, for example, an agent
could loosen its security requirements for messages that have reached it via an access path that the agent knows is trusted and thus secure.

To clearly demonstrate this integration of access control and discovery, a simplified description of the structure and functionality of a group manager is given below.

The primary function of a group manager is to maintain and update a list of aliases. There is one or more aliases for every agent that belongs to this group. Each row of the alias table consists of five elements:

1. **Alias.** This is the pseudonym by which an agent, registered with this group manager, is known. It is set by the agent upon registration with this group manager.

2. **Address pattern.** Any message whose destination matches the alias field of a row of the alias table will have its destination field appended with the corresponding address pattern. This pattern can be the true identifier of the agent or a new alias, thus allowing for multiple indirection. In the most general case, the address pattern can be a general procedure (an executable program) that it is invoked to determine the new destination of the matched message. This program takes as input the characteristics of the query and generates as output the address pattern for the query.

3. The **security field** contains the identifiers of the lock agents that need to be contacted to check whether the message possesses enough authorisation to proceed to the next destination. The lock agents are engaged only if both the owner of the message and the access path it has followed to reach this group manager match the corresponding ownership path and access path fields, respectively. The lock agents specified in the security field are generally the input nodes of a complex authorisation graph. The results produced by this graph will be sent to the message's next destination agent, whose lock table will determine whether the message is authorised to activate that agent or not [7].

The structure of the alias table, as described above, leads us to the following important observations:

a) Every time a group manager redirects a message to a new destination, it concurrently activates the authorisation graph associated with that destination. Thus messages that are generated during the execution of a program proceed in parallel with the messages that are produced by authorisation graphs.
b) As part of an agent's registration procedure, the group manager is given the authorisation graph to be evaluated for every message addressed to the agent. This authorisation graph can depend on the ownership path of the message and the access path it has followed [10]. This means that an agent, upon its inclusion in a group, can define the security requirements for all categories of incoming messages that are received via the group. Group managers thus integrate security with routing. Both addressing and access validation are cascaded through a chain of group managers en route, which allows the address and security information to be translated incrementally. Accordingly, the destination address is structured into a slash-separated (by convention) list of patterns with the head of the list being processed at every group manager along the message route.

4.7 The links

From the previous section it is clear that the group managers form a network of control nodes of this massively-aliasing system. The primary function of a group manager is to maintain and update a list of links, which contain the security and routing information. In the process of registering itself with a group manager, an agent creates a link per each pseudonym it wishes itself to be known by.

![Diagram of a link]

Figure 4.5 The structure of a link

A novel view of a link is that of a lightweight object which can only exist as a part of a group manager environment. Representing the link as an object-like entity offers the
advantages of data encapsulation and security that can be applied to each individual link. Furthermore, requirement three (re-usability) of section 4.5 is satisfied because links can be used as components that can be “copied and pasted” from one group manager to another. The composition of the link is shown in figure 4.5.

1. **Name** is a pseudonym of the agent that has created this link. The name can either be a single lexical unit or a sequence of units. In this way the resource agents can use more than one keywords to describe their published services.

2. **Destination.** Any message whose destination field matches the *name* part of a link will have this field replaced by the corresponding address pattern which is stored in the *destination* field of the link (see the structure of the alias table above). For each destination specified there is a corresponding security field, too.

3. **Constraints.** Each link can contain a constraint base which contains a set of constraints that each message (agent) must satisfy in order to be permitted to travel via this link. These may, depending on the application, have the meaning of size restrictions, minimum payment credits, etc. The constraints can be passive and expressed in a declarative notation such as XML. In this case the group manager is responsible for reading and matching them against the characteristics of the query. A constraint can also be represented by an executable program, which is triggered by message arrival. The group manager executes the “constraint program” and gives it an interface via which it can read and/or modify the contents of the information base of both the link and the message. Details of this interface will be presented in the next section. The constraint program and indeed any other active part of the link can be written in an interpreted language for which the group manager has a safe interpreter. Examples of such languages are Java, safe-Tcl, etc. As was mentioned earlier, the group manager mediates this interaction to ensure that any vital, constant characteristics of both the link and the query are not altered accidentally or maliciously.

4. **Information base.** This component of the link acts as a general purpose information repository. Some items of this repository can be made public whilst others can be accessible only by specific agents. More specifically the information base of a link is defined as a set of *information items*. Each information item is defined as a tuple: $\langle \text{name}, \text{value}, \text{type}, \text{access\_key} \rangle$ where *name* is a string uniquely identifying this item in the information base, *value* is the content that this item has, *type* defines whether this
item is read only and \textit{access\_key} is a string used for access purposes. If the \textit{access\_key} is present then if a query wishes to access this item, it should also provide the required key. The provision of access key is a simple method of controlling access in the information base at a fine grain level. It is necessary as a basic protection against a malicious query destroying information intended for another query. Because links are \textit{copyable} objects, it is important to represent their properties in a machine-independent language, such as XML, which is a \textit{de facto} standard for data interchange in the WWW.

Each link possesses a standard interface comprising operations that incoming messages can perform on the link. One of the most important commands of this interface is the copy command. If an agent invokes this command then the link as a whole can be copied to another group manager. For integrity and security reasons the owner of the link should be able to specify which parts of the interface are visible to which agents. To achieve this an Access Control List (ACL) is incorporated as part of the state of the link. This is a simple table that defines which operations are visible to which agents.

The links according to their destination field can be classified into two categories: a) static, in which case the destination field contains a fixed, static destination pattern and b) dynamic, in which case the destination field contains an executable program.

Figure 4.6 shows a typical link in XML form:

```xml
<?xml version="1.0"?>
<link name="test">
<destination type="dynamic" language="perl">
  code="/usr/bin/my_link"  
</destination>
<constraints type="static">
  query_owner=nick
  query_topic=research
</constraints>
<information_base>
  link_owner,css1na,r,null#
  msg_100,hello,w,null#
  msg_210,query100passed,w,abcd#
</information_base>
<ACL>
  null
</ACL>
</link>
```

Figure 4.6 An XML link
4.8 Query and its interface

The structure of a query is a simplified version of the link structure (figure 4.7). Each link has an ID assigned to it by the query-originator agent. The query specifies a destination which is the name of a link or a list of links to be followed. Similar to a link, a query also possesses constraints and an information base that have the same semantics as the corresponding components of a link. Finally a query can specify an action to be performed when it reaches its final destination (target resource agent).

![Diagram of query structure]

Figure 4.7 The structure of a query

The action field of a query can either specify the invocation of a function of the resource agent for specific arguments, or a program it should execute. Figure 4.8 shows the XML representation of a typical query.

```xml
<?xml version="1.0"?>
<query ID="my_query_1234">
  <destination>
    travel_link
  </destination>
  <constraints type="static">
    link_owner=css1na
    link_topic=asian_travel
  </constraints>
  <information_base>
    query_owner, css2as, r, null#
    query_history, surrey_travel_link, r, null#
  </information_base>
  <action type="static">
    f, x=10, y=20
  </action>
</query>
```

Figure 4.8 An XML query
The constraint fields of both queries and links can contain executable programs. Such programs are executed by the group manager which gives them access to the information base fields of both the query and the link. To this end, the program is provided with two primitives:

a) get(info_base, item_name, access_key). The first argument determines which of the two information bases the program requires access to, the second argument specifies the information item to be accessed and the third argument provides the access key required to access this item.

b) set(info_base, item_name, value, access_key). This command allows the program to set the value of the information item item_name in the information base info_base using the access_key.

A program execution has the nature of a transaction in that if the program fails, the set requests are rolled back. The following novel algorithm is used to process arriving queries:

1. Fetch the query destination.
2. If the destination is a list of links, use the head of the list.
3. Match the content against the names of the current links.
4. For every match do:
   a. Start a transaction
   b. Execute the constraints of the query.
   c. Execute the constraints of the link.
   d. Commit the transaction
   e. Execute the destination field of the link to determine the query's next destination.
   f. Append the result as the head of the query's destination field.
   g. Route the query to the next hop.
5. Proceed to the next element of the destination list (if any).

This method nicely follows one of the main concepts of the proposed architecture: the peer interaction between a query and a link. Furthermore, by using transaction discipline, it ensures that the state of the query and the link does not change if their matching fails. The modifications the query and the link perform on each other's information bases persist only if their interaction succeeds.
As shown above, a query is allowed to match, and travel through, more than one link in any given group manager (i.e., to multicast). The information base of the query can include an item to indicate the policy the agent-creator of the query wishes to follow. For instance it could specify unicasting of the query or a maximum number of links to be followed in every group manager. In a similar fashion, the group manager could implement its own policy on this issue. If the two policies conflict then the more restrictive one is obeyed. The proposed architecture follows the CORBA specification of traders at this point.

4.9 Access paths coupled with customised security

The links as presented in the previous section are re-usable components. The proposed architecture also supports interlocking links. This is an important characteristic of the abstract model presented in section 4.4 and as a result it becomes one of the design requirements of the concrete architecture.

The interlocking links enable the resource agents to create automatic access paths. An access path is a chain of links of arbitrary length, i.e. a link set where members point at one another in a specific order. A query traveling along this path only needs to activate the first link. After that, the query is forwarded automatically from one link to another. The links of a path can belong to physically-distributed group managers.

Note that in this architecture every group manager is an independent specialised agent. Therefore, it has an address that queries can use to access it directly. Consequently an access path does not have to be traversed from the start. A query with sufficient authority can travel along a path by using any link of it as a starting point. Path segments can thus be utilised as re-usable components, too.

As a simple example, let us assume that there is a group manager A which contains a link x that points to a resource agent Ag1. The Ag1 is the creator and owner of the link x. Now let us assume that the security requirements that a query must satisfy in order to travel through the link x are $S_{x,Ag1}$. Furthermore let us assume that there is another group manager B in which the agent Ag1 wishes to create a link y that points to the link x. To do this the agent Ag1 sends a request to group manager A invoking the function
connect_link. This function is included in every group-manager interface and has the following parameters:

\[ \text{Connect\_link}(\text{new\_link}, \text{current\_link}, \text{group}, \text{state}, \text{security\_required}) \]

where \text{new\_link} is the name of the new link to be created in the group manager \text{group} whose contents are defined in the parameter \text{state}, its destination field is defined to point to the existing link \text{current\_link} and the security requirements that a query has to satisfy in order to be forwarded from the \text{new\_link} to the \text{current\_link} are specified in the parameter \text{security\_required}.

On receiving this request, the group manager, according to the policies it implements, might modify the contents of the \text{security\_required} parameter in order to incorporate its own authorisation requirements. The next and final step is to forward this request to the group manager \text{B} where the new link will be created. In order for this to succeed the owner of the existing link must have authorisation to create new links at the destination group manager \text{B}.

Here it must be noted that the group manager is only allowed to incorporate more security requirements to the existing ones and not reduce them. This is achieved by ensuring that any implementation of a group manager has certain, predefined, fixed functionality. This is achieved by having a dedicated, specialised agent in our architecture, called creator, which is the only authorised entity in the architecture to create new group managers. The creator is publicly available and can be contacted by any user agent to create simple or composite group managers. In the latter case the creator receives as input from the user agent the set theoretic expression the new composite group manager must satisfy. It is then responsible in getting the contents of the relevant group managers and combining them appropriately to produce the requested new composite group manager.

Fake group managers created by malicious user agents can be detected because each valid group manager could use in its communications a secret key that can be shared among a group of valid group managers. An alternative is that the creator agent acts as repository of the addresses of valid group managers in the local system. Every group manager that attempts to contact another group manager for the first time should contact the registry to ensure that the destination group manager is valid and vice versa.

If we represent the contents of the \text{security\_required} variable as \( S_{y,2} \) and the security requirements the group manager \text{B} has added as \( S_{y,2} \) then it can be said that the security...
requirements for the path y->x->Ag1 are \( S_a \cup S_{yx} \cup S_{y,x2} \cup S_{x,ag1} \) where \( S_a \) are the authorisation requirements a query has to satisfy in order to access the group manager \( A \).

The above equation can be generalised as:

\[
S_{\text{path}} = S_{\text{agent}} \cup S_{\text{groups}}
\]

where \( S_{\text{path}} \) are the total authorisation requirements for an arbitrary path, \( S_{\text{agent}} \) are the security requirements that the end-of-path resource agent sets and \( S_{\text{groups}} \) are the security requirements added by the participating group managers. This equation is very important because it allows: a) the resource agents to create and customise the access control requirements for access paths to themselves and b) the group managers can impose a set of minimum security requirements that a query has to satisfy. The first characteristic is innovative because it couples agent discovery with access control and the second point is important because it makes the model symmetric.

### 4.10 Operations on group managers

As was discussed above, group managers form a conceptual network where resource agents may be associated with more than one node simultaneously. When a resource agent is registered with more than one group manager, this corresponds to a resource that falls under more than one category. Discovering multiply-registered agents in the existing systems usually involves a poll of all potential categories and a merge of the information found there. In the distributed case, those may result in massive communication. To remedy this, we propose aggregate organisation nodes that contain collated information obtained from several group managers. Since those are little more than sets of links, set algebra can be utilised to define the aggregated nodes, as follows:

- **a)** Union. A link will become a member of the union group manager if and only if it appears in at least one of the participating group managers.
- **b)** Intersection. A link will become a member of the intersection group manager if and only if it appears in all the participating group managers.

A natural question that might arise at this point is why the set complement operator, being one of the primitive operators in set algebra, is not represented and used by the proposed system. The set complement can not be defined in this system because this would require the knowledge of the universe of discourse of each group manager. If it
was known in advance that the universe of discourse of a group manager is a set of \( n \) specific links, then assuming that the group manager contains \( m \), where \( m \leq n \), links, the complement of this group manager could be defined as a new group manager containing the remaining \( n-m \) links. In the proposed system though group managers do not have a universe of discourse and thus the result of taking the complement of a group manager is undefined. Moreover it is neither practical nor efficient to assume that the universe of discourse of a group manager is the union of all the links in all the group managers. In such case the complement group manager would have to be notified by any content changes in all existing group managers thus making the process of updating the complement group managers unscalable.

In order to maintain the semantics of the union and intersection operations it is necessary to ensure that no resource agent can register directly with a composite group manager. By using these primitive operations composite group managers can be built to represent set expressions among a number of simple group managers to support two fundamental data-organisation primitives: generalisation and specialisation.

In order to create composite groups there must be a set of rules which determine whether a link is equal to another link. This is necessary for intersections, but also for unions, where a link must occur no more than once.

First of all, not all components of a link are important for determining the link identity. The information base refers more to the link state than function. Also, two links differing only in the access-control and constraint components should be considered identical for the same reason. On the other hand, the link name and destination are part of its identity and have to be matched exactly. Since, nevertheless, all of the link parts need to be represented in the aggregate-node link, those non-identity-related items have to be combined in some manner. The access-control information and the constraint base allow simple combining, which could be either conservative (i.e. the more restrictive item wins) or liberal (i.e., the less restrictive one does). This however, does not have to work the same for constraints and access control in that the link with more security can have weaker constraints, making it impossible to choose both items consistently. Indeed, the weaker constraints could be weak because there is a tough security limitation added to them on the same link. In this circumstance, the aggregate link can only refer to both source links.
Another issue is data consistency. As aggregate nodes totally depend on the group managers they are intended to represent, any link update in the latter should cause the corresponding update in the former. To minimise additional data traffic between simple and composite nodes, which could be significant, we limit the aggregate node to reside on the same physical machine as the nodes they represent.

Finally, remember that links can be interlocked. In order to enable the decision-making process at the starting point of an interlocked chain, the eventual interface of the resource agent the chain is leading to is propagated back up the chain. Whenever an aggregate link is created, its attributes reflect the respective interfaces of all the destination agents associated with the source chain-links. To analyse this further the following facts and definitions must be taken into account.

The links in the proposed architecture can be classified into three categories according to the type of their destination field. The first category are the fixed links whose destination field is a string representing the new address the queries will have to be forwarded to if they match this link. The second category are the multi-links whose destination field is a table containing a set of possible destination addresses. In this category, if a query matches the link then the query is forwarded undeterministically to one of the destinations specified. Our architecture requires the existence of such links because they provide an efficient representation of groups of agents registering themselves with a single link to a group manager and thus enjoy common access paths and therefore common common constraints, information and access control information. These agents could either offer the same services in which case the multi-link represents replicated services or offer different services. In the latter case the multi-link acts as a registered agent company. In general the existence of multi-links offers a way to form groups of agents sharing the same interests/services as well as the same access paths and policies. The third category are the dynamic links whose destination field is an executable program that determines at run time the new destination for the query that matches this link. The dynamic links are the logical extension to the multi-links. Multi-links are essentially static and explicit. But there can be situations in which the new destination where a query has to be forwarded depends on a number of factors such as the query itself, real-time availability information, etc. In these cases it is much more efficient to have a program that determines dynamically the new destination of the query rather than having a large multi-link enumerating pairs of conditions and destinations.
The links in the proposed architecture can either point directly to a resource agent or point to yet another link. The first type are called terminal links (in accordance with the abstract model presented in section 4.4) and the second type are called indirect links. Because of the existence of indirect links it is possible that even if two links point to different destinations, their final destination is the same resource agent. This phenomenon can happen because two links can participate in two different access paths which converge at a certain point and thus are connected to the same end resource agent.

Because access paths are lists of links and because links can be of different types, it can be said that in general an access path is a list of links of different types. The main characteristic of an access path from the point of view of the user agents (these can be called the clients of the path) is the resource agent the path is leading to together with the set of services the resource agent offers to queries incoming from this path. We can define a parameter for every path called path_services which is equal to the pair <Resource, Interface>. The path_services parameter is an important piece of information that must be made available to all links participating in a path as read-only items in their information base. This parameter can be used by queries to avoid travelling through a long path to perform a specific action only to discover that the end resource agent does not offer the required function or that the invocation of the function requires different syntax. An access path has to be associated with a deterministic value of the path_services parameter in order to have significance and provide real assistance to the queries. For this reason the last link of an access path must always be a terminal link. This is also important in order to ensure that an access path does not contain any loops. Access paths with loops should not be present in this architecture because they could create livelock situations for queries thus seriously degrading the performance of the system. In the simplest case an access path consists of $n$ fixed, indirect links and its final link is terminal. Depending on the type of its terminal link we have the following scenarios: 1) The terminal link is a fixed link. In this case the path_services parameter is equal to the address of the end resource agent coupled with the interface this agent offers to queries travelling through this path. 2) The terminal link is a multi-link. In this case the path_services parameter is equal to the union of path_services parameters defined for each destination the multi-link specifies. 3) The terminal link is a dynamic link. In this case the possible destinations are not specified and thus the path_services parameter of such path are set to the address of the group manager this link appears in coupled with the name of the link. This pair forms the only alternative
deterministic information that could be associated with such path. A dynamic link is always considered terminal because it defines the point beyond which the access path becomes undeterministic. Therefore it can be said that an access path either terminates on a proper terminal link (fixed or multi-link) or at the first undeterministic point (dynamic link). By using the aforementioned simple scenario, the path_services parameter can be evaluated for more complex paths whose links are not fixed but they also contain multi-links. For example let us assume that figure 4.9 shows a path where the circles indicate indirect multi-links, the dots fixed indirect links and the squares fixed terminal links. The question is what is the value of path_services parameter of multi-link X.

![Image of complex paths](image)

**Figure 4.9 An example of complex paths**

At the terminal links the value of path_services is A, B, C and D respectively. Link Y is a multi-link and thus its parameter is equal to the union of the parameters of all its “branches” which is A,B. Therefore the value of the link X is the union of the parameters of all its branches which is A,B,C,D.

In conclusion the proposed architecture creates two layers of indirection above the actual resource agents: the simple and the composite group managers. From the previous discussion it should be clear that the query traffic is going downwards (composite groups -> simple groups -> resource agents) whereby the update traffic is directed upwards (resource agents -> simple groups -> composite groups). Therefore it is possible that a query might pass through a composite group manager before this manager receives an existing update from its component simple groups. This is a period during which the system can be in inconsistent state. But because the update traffic a) moves opposite the
query traffic, b) is minimised (only certain updates are required to be broadcast) and c) is localised, the system is guaranteed to reach the consistent state quickly after the occurrence of an update.

4.11 Top level view of a group manager

The top-level architecture of a group manager is depicted in figure 4.10. The following components are included:

1. **Agent links.** The agent-link layer is the minimum functionality a group manager has to provide. This layer includes the group of links the member agents have created. This repository of links acts as a routing table that redirects incoming queries towards the appropriate destinations.

2. **Service links.** These links are a group of links not created by the member agents. When followed, they provide different services that the group manager offers to its members. These links can be created during the creation phase of the group manager. For example, these could be: email, access to specific WWW sites etc. The provision of the services can be hierarchical. For instance, only certain member agents could be allowed to access the WWW whereas the email service could be available to all the members. The provision of service links makes a group manager act as an information service centre for its members, rather than a mere query router.

3. **Synchronisation links.** Member agents of the group manager can define and use synchronisation links (SL). The purpose of these links is to act as independent
synchronisation points among possibly different computation threads in the form of queries traversing the network of agents. The synchronisation and agent links are structurally identical with only one difference: An SL additionally possesses temporary storage for queries. An SL is defined as a tuple:

\[ \langle \text{Name, Attributes, Condition, Action, matching-store} \rangle \]

where,

**Name:** This is the name uniquely identifying this SL in the current group manager.

**Attributes:** An SL can be temporary or permanent, visible by all computation threads or only by a specific one, etc.

**Condition:** When a query's destination field matches the name of an SL in a group manager, then it is checked whether the new query together with stored companions satisfy the condition. If it does then the **Action** is performed. The action typically is the passing of the existing queries to one or more agent links for processing and forwarding. Otherwise, the new query is stored temporarily in the **matching-store** awaiting the arrival of further queries. The synchronisation links can be used to efficiently synchronise the traversal of the organisation structure by a number of agents (queries). For example a synchronisation link can be used to ensure that two queries can continue their travel if both arrive at the same group manager. The concept of synchronisation points, their structure and functionality is discussed in more detail in [20].

4. **Agent support.** Since the constraint field of the queries can contain a program that is executed in every group manager the query goes through, it can be said that a query in the presented architecture can be a mobile agent. The current implementation of the group managers supports only the sequential processing of queries. The group manager structure can be extended to support the concurrent execution of queries. In this case, a group manager can act as a mobile agent venue in which agents execute and communicate with other locally resident agents. Such group managers must implement the **agent support** layer which comprises: a) a registration and local agent discovery facility, b) local inter-agent communication facility and c) the appropriate interpreter for each agent, assuming that agents are coded in interpreted script languages. Agents visiting a group manager can make use of its two lower functional layers.
5. **State.** The state of the group manager is a repository of the group manager attributes, policies and interdependencies with composite group managers. Each group manager maintains some attributes which remain constant throughout its lifetime such as its address, owner, topic, etc. It also maintains and implements a number of policies regarding, for instance, the maximum number of links a query is allowed to follow, its own access control list, etc. Furthermore each group manager has to know the addresses of either a) the composite group managers it participates in, if it is a simple group manager or b) the simple group managers it depends on, if it is a composite group manager. In the latter case it must also know the set-theoretic expression its contents must satisfy. This information is needed so that the group manager may a) notify any composite group managers of any changes in its contents, b) accept notifications of updates in the contents of relevant simple group managers, c) ensure that after it receives an update, it recalculates its contents to reflect that change while maintaining the validity of the set-theoretic expression it implements. It must be noted that the update messages can also inform the group manager of the removal of one of its components and vice versa. The state of the group manager being essentially a set of declarative statements, it can be efficiently represented in XML similarly to the individual links and queries.

### 4.12 The name registry

The final component of the agent organisation/discovery architecture presented in this chapter is the name registry. The name registry is a specialised agent which implements the notion of constant namespace described in the abstract model of section 4.4. There is one name registry in every physical server implementing the proposed architecture. This distributed set of name registers implements, and ensures the integrity of, the constant namespace. Since names are used to identify links in every group manager, providing global control for names means that the name registry effectively controls the naming of the links in the group managers. Each name registry agent provides a local portal to the complete name registry. The main data structure each name registry agent maintains is a database which consists of tuples of the following form:

\[
< \text{name}, \text{owner}, \{ \langle \text{context, constraints, semantics} \rangle \} >
\]
Each tuple defines the global characteristics of a registered name. *Name* is the string representation of a name. In the name registry every name is associated with a set of global characteristics. Each tuple of this set defines the *semantics* the *name* must have if used in the group manager context and the prospective user of the name satisfies the associated *constraints*. The *owner* field shows which agent is the registered owner of the name.

The *context* field can either indicate the address of a specific group manager, or the topic (category), or a set of group managers/topics for each of which the corresponding *semantics* should be given to the *name* if it is registered in any of the specified group managers/topics.

The *constraints* field contains a set of declarative statements that the characteristics of the prospective user of the registered name must satisfy in order to be allowed to use this name. These constraints can include the type and characteristics of services the user wishes to associate with a name in the group manager, the characteristics of the user itself, etc.

The *semantics* field includes any base semantics that must be given to the specific name when used in specific contexts. For example the name *microsoft_corporation* can be registered and defined so that whenever this name is used to represent a link in any group manager then the destination field of the link should include the address of the Microsoft corporation. This means that any query that matches this link will be forwarded to the address of the Microsoft corporation representative agent. The semantics associated with a name can be *fixed* in which case no user of the name can add any extra semantics to the name whenever it uses it. Alternatively, the semantics are called *base semantics* and any user of the name can add his own semantics to them.

Each name registry agent implements an interface via which user agents can register and access name declarations. There are two main operations that each name registry agent performs:

1. *register_name*. A user agent can invoke this function in order to register a specific name. The request must contain the set of tuples (of the above form) that the user wishes to associate with this name. The name registry agent, upon receiving such a request, contacts other name registries in order to ensure that the name is not already registered. If the operation is successful then the new name's declaration is inserted into the database.
2. **Search_name.** Group managers must ensure that no agent can register a link with them using *illegally* names from the name registry. Group managers use this operation whenever a resource agent attempts to register a link with them. Because of the importance of the name registry, each name registry agent must have a local well-known address so that the local group managers can access it.

### 4.13 Evaluation

The described architecture [6, 9] is based on the abstract model developed in section 4.4. Thus the requirements of the abstract model are satisfied. Specifically, the semantics of a link can depend on the semantics of another link in another group manager (notion of indirect links and multi-links); group managers can register themselves with other group managers (in other words, group managers can behave as resource agents towards other group managers); the external namespace is implemented with the notion of dynamic links whose destination field can contain a program that accesses and modifies the external file system; the constant namespace is implemented with the notion of name registry agents distributed in different physical machines.

Furthermore, the architecture satisfies the requirements of section 4.5. Agents can create their own organisational structure by accessing the group manager creator agent, there is no ontological connection between resource and group manager agents therefore links can be created/deleted without affecting the existence of the resource agents, access paths can be created and modified at run time via the interface of the individual links (*adaptivity*).

Queries and links are considered as peers entities that interact with each other via the group manager entity. The information base of a query can be used to show the history and any intermediate results. The information base of a link can change based on the number and nature of the queries that have travelled through this link. As a result both access paths and queries can be mutated because of this interaction. The group manager ensures that certain vital characteristics of both are maintained intact for integrity and security purposes. (*interaction*).

The access paths are coupled with access control. The destination field of a link not only points to a new destination but also incorporates the authorisation criteria the query
has to possess or acquire in order to be accepted by the agent the link points to. As it was shown, the access control requirements of a path also depends on the access control policies of the participating group managers. Thus the model integrates access control with discovery and treats the organisation nodes (group managers) and resource agents as peer entities. (security)

The concept of re-usability is present in four levels: a) individual links are re-usable components that can be copied among group managers, b) resource agents can form small groups and register themselves at a single point in the group manager structure, thus enjoying common access paths and hence common discovery and access control policies, c) existing access paths can be used to build new access paths thus re-using the characteristics (constraints, information and access control policies) of the existing paths, d) existing simple group managers can be used to create new aggregated organisation nodes (composite group managers).

The concept of service links as described in section 4.11 enables the group managers to offer a set of high level services to their members. Each group manager can “come” hardwired with various services from its creation. In this way the group manager can become an information-services centre for the resource agents registering with him. In fact the group manager can become a virtual market place which uses its onboard facilities (service links) to attract customers (resource agents) to register with it and buy (access) his services. To achieve that the group manager can register with other group managers for marketing purposes. Thus there is the architectural flexibility to transform the group managers from simple query routers to electronic business centers (high level services for agents).

Finally every group manager represents a category which is specified upon its creation and remains intact throughout its lifetime as one of its permanent attributes in the state component. Each group manager implements a registration protocol that determines whether a particular resource agent can register a specific set of services according to the relevance of the services to the category or topic the group manager represents. The presented architecture does not give any specification for this protocol because this depends on the topic of the group manager and the strictness its owner wishes to enforce. Instead this architecture supplies the group manager with the prototype of a method whose implementation depends entirely on the owner of the group manager. (specialisation).
The *access control mechanism* being coupled with access paths offers several further benefits: a) from the point of view of the resource agents there is now one more attribute in this system to assist them to do fine/coarse grain access control policies. Specifically the resource agents can now attach specific capabilities to pairs of user agents and access paths. The access path itself becomes a factor that determines the security requirements a user agent has to satisfy in order for his request to be authorised by the target resource agent. Furthermore the unauthorised requests can be blocked before they reach the target agent therefore the problem of *denial of service* attacks at an agent level can be avoided assuming that the resource agent has not made its address public thus enabling other agents to contact it directly. b) because the access control requirements for a path are distributed to its component links this means that the access control information of a path can be shared between a number of paths (security knowledge sharing). c) user agent queries need not acquire all their credentials from the beginning of their travel. Each hop within an access path specifies the credentials the query needs to have in order to go just to the next destination. Therefore the problem of determining all the needed credentials before the trip starts (in a CORBA fashion) is eliminated. Furthermore, for this reason the size of the query is smaller. The security risk is also minimised because if the security information the query possesses (credentials) is compromised then the damage is restricted since the query only carries enough authorisation for the next hop and not for the whole journey. d) the access control mechanism can allow both *synchronous* and *asynchronous* security. As discussed earlier the computation thread the query represents can be split in an authorisation thread and an activation thread. These threads proceed asynchronously and they get synchronised at the next hop. Alternatively the links of a path could require a key (password) that the query needs to have in order to be authorised by the destination agent. In this case the security policy is called synchronous because the computation thread is not split as above.

The described architecture possesses inherent mechanisms which guarantee that the system will reach a *consistent state* automatically after a change in its state occurs. Existing organisation models fail to provide such mechanisms. In summary the consistency mechanisms of this organisation system as follows: A) Each link registers its parent links (the name and group manager of every link pointing to that link); This information is available because in the context of this organisation system the process of interlocking links is destination based rather than source based. In this way if a link is destroyed
(removed) then all its parent links are notified and are either removed too or "re-wired" to point to another link or resource agent. Such a decision can be taken based on the value of a specific user-defined constant stored in the information base of the link. B) Similarly with the links, simple group managers record the IDs of all the composite group managers which depend on them. This is necessary because if the contents of a simple group manager change then these changes need be propagated to all pertinent composite group managers such that their contents remain consistent with those of the participating simple group managers.

Ease of implementation inherent in this model is due to the fact that the declarative parts of all the components of the architecture (links, queries, name registry, group managers) can be represented in the standard XML notation. The functional parts are arbitrary programs written in interpreted languages. Because XML is quite simple, a prototype implementation of the architecture is straightforward.

The architecture is general purpose and thus it can be applied to organise arbitrary entities passive or active. In fact the architecture at its current form has direct application to the important area of hypermedia since its main concept of treating links as separate objects from the data objects conforms and naturally extends the main principle behind hypermedia.

4.14 Implementation issues

1. **Scalability.** The provision of automatic consistency mechanisms within the architecture implies that the update traffic increases with the number of composite group managers created. This traffic also depends on the frequency of other user actions, such creation and deletion of resource agents, group managers and links. If the number of users of the system is high then, in combination with the aforementioned factors, efficiency may degrade. To counter that, in a realistic implementation the users (and therefore their proprietary agents) must be restricted in the number of simple and composite group managers they are allowed to create. These restrictions could be enforced at a coarse grain level on groups of users rather than individuals.

2. **Livelock.** As mentioned before, livelock occurs when a user query travels across the organisation structure indefinitely without reaching a final destination. This
phenomenon could occur in our system either because a) the query follows a specific access path which is cyclic or b) the query's destination pattern is inherently cyclic. Case (b) may happen due to the group managers in our system not being organised in an acyclic hierarchy. In fact, the case (a) livelock can not happen because the access paths in this architecture are built bottom up. An access path construction starts from the terminal links first which in turn can be re-linked to new links. In this way, it is ensured that an access path always terminates on one or more resource agents. Case (b) livelock can be prevented by registering in the information base of every query the IDs of all the links this query has travelled through.

3. Duplication. Duplication is the phenomenon whereby the same query arrives at the same resource agent more than once. Duplication could appear in this architecture if a query specified a multicasting policy which means that it wishes to be replicated and traverse more than one links concurrently. If the only types of links in our model were the fixed and multi-links, then the group manager where the multicasting were requested could check the path_services parameters of all the relevant links and determine whether they present a duplication hazard. Since there are also dynamic links, this approach is not guaranteed to be successful. To address the issue of duplication, the following decisions have been taken: a) if a query wishes to travel through a multi-link then it is forwarded only to one of the destinations the link points to. This rule can be relaxed if all the destinations of the multi-link are different resource agents. In a similar manner only one destination is used from a multi-link appearing in a composite group manager. In general if the rule of following one link in every group manager was enforced then no duplication can occur. However duplication has also a positive aspect in this architecture. Let us remember that even if the path_services parameter for two links is the same this does not mean that the actual access paths are the same. This means that the constraints and access control policies of the two access paths can be completely different. Therefore a query could be successful in travelling along one path but unsuccessful in travelling through the other. If duplication was permitted then it could be guaranteed that a query would reach the eventual resource agent if its characteristics satisfied the requirements of at least one access path to it. If there was no duplication and the access path to be followed was decided non-deterministically by the group manager then the system would be unfair. The solution to this problem is to allow the group manager to record
in the query's information base alternative links it could follow from this group manager in case at some stage later it is not allowed to proceed any further. So for example if the query can not satisfy the constraints of the required link in one of the next group managers then the current group manager could read the query's information base and if there is a list of alternative links (from other group managers) then the query is sent to the first alternative in the list. This procedure is similar to depth-first searching.

4.15 Conclusions

An agent organisation system is proposed which is based on a set of requirements derived from the general application area for mobile agents. The system has the advantages of usability, extensibility and integrated security mechanism. Existing organisation nodes (Group Managers) can be re-used to produce new more complex ones. Moreover the system is dynamic in nature because queries and links are active entities which interact with each other as peers. The properties of the system have been discussed and its features evaluated identifying the main advantages over the existing models as well as the key issues of its implementation.
Chapter 5

The MASIC Access Control Architecture

5.1 Introduction

The mobile software agent paradigm provides a generic, customisable foundation for the development of high performance distributed applications. An efficient, general-purpose access control mechanism is required to support the development of a wide range of applications. This is achievable if the design of the access control system is based on the principles of simplicity, programmability (customisation) and reusability. However existing mobile agent architectures either neglect this issue at all or offer centralised schemes that do not support adaptive access control on a per-agent basis and do not address the issues of secure knowledge sharing and reusing. In this chapter a novel, distributed access control architecture is presented based on the concept of distributed, active authorisation entities (lock cells) any combination of which can be referenced by an agent to provide input and/or output access control. It is demonstrated how these lock cells can be used to implement security domains and how they can be combined to create composite lock cells.

An agent can be involved concurrently in multiple computations either as a client or as a server. Access control must ensure that an entity possesses enough authorisation to invoke a specific operation on a specific target entity. The following issues should be addressed:

1. **Integration with the computational model** Any computation that roams across a server network presents a security hazard to its recipient.

2. **Generality** The access-control architecture should be general-purpose so that it can meet the security requirements of a range of applications.
3. **Flexibility** Applications should be able to create and dynamically modify their own access control policies and mechanisms.

4. **Distribution** Centralised access control cannot support customisation on a per-entity basis since it would have to act across a number of administrative domains.

5. **Security-knowledge sharing and reuse** Parts of the access control structure of an entity can be shared with other entities, possibly for subsequent extension. In this way access policies can be applied to groups of entities rather than individuals (creating “security domains”).

The existing access control approaches do not possess most of the aforementioned properties. Security in general is either neglected, as in [20, 61, 108], or is based on complex centralised architectures [51, 106] that contain hidden security holes [34].

The prevailing methodology of inter-agent access control is built around the notions of *capabilities* and *Access Control List* (ACL) [46, 98]. Each agent carries certain capabilities assigned to it by the principal (user) who owns the agent. Access to an individual agent is controlled by a single central ACL in every host. This mechanism, although simple, is inherently static and passive. An agent usually has to carry all the privileges of its principal in the course of its distributed execution. The ACLs are static entities, which can be customised only by an administrator. In CORBA, for example, the security administration interfaces are hidden from the application objects. Furthermore there is no concept of security-knowledge sharing and reuse. Implementing security domains on these systems requires the introduction of further entities (security domain managers in CORBA), thus making the system more complex. In summary, this access control mechanism suits static distributed applications whose access control requirements are known before their deployment and remain largely constant throughout their lifetime.

On the other hand, the world of mobile agents is anything but static. The contents and functionality as well as the environment of an executing agent can change dramatically during its lifetime. The principal, or indeed the agent itself, should be able to update any access control schemes to support and protect the changing functionality.

In this chapter a distributed access-control architecture for mobile agents based on the abstract access control model described in [5] is presented. Section 5.2 gives an overview of the abstract model. In sections 5.3, 5.4, 5.5 and 5.6, the main entities and mechanisms of the architecture are described. Section 5.7 illustrates an object-oriented model of the architecture. Finally, section 5.8 has the conclusions.
5.2 Overview of abstract model

The proposed access control architecture is based on a novel abstract model as shown in figure 5.1. The access control model is an integrated component of the agent architecture presented in this thesis (MASIC). A cell in the context of this system is a trusted agent-wrapper encapsulating untrusted user-defined functionality. Users can create wrappers and then use them as containers of passive data entities and active program entities. Each cell advertises an interface via which other cells can access the contained user-defined functionality through synchronous or asynchronous message passing. An important service each cell offers is a synchronisation store, which supports the collaboration of cells for invoking a function of the target cell. Cells can provide partial invocation data (i.e. function name and arguments) and the target cell has the ability to combine these into a single complete function invocation. This form of synchronisation/collaboration is exploited in the authentication mechanism of the concrete model. Now let us turn to the access-control methodology being proposed.

![Figure 5.1 The abstract access control model](image)

**Locks and keys.** Cells are [locked](#) with a [lock expression](#). Each cell maintains a [lock table](#). This table assigns different authorisation criteria (lock expressions) to different users or groups of users. Messages are provided with a [key](#). The purpose of the lock expression in a cell is to restrict the access to that cell only to those messages that possess a key that fits that lock expression. The locking mechanism is based upon the concepts of [lock cells](#) and [locked cells](#). A cell is said to be [locked](#) if it references by name one or more lock cells. In its simplest form, every lock is just an independent name-registration centre maintaining a list of User Identifiers (UIDs). The registration procedure and
requirements that a user must follow and satisfy, respectively, in order for his UID to be included in the lock’s list can differ dramatically between locks.

The successful registration of a user \( x \) with a lock \( l \) is equivalent to giving \( x \) a personal key for \( l \). The possession of a personal key for a lock \( l \) by a user grants him the capability of being partly or fully authorised to access any cell that is locked by the single lock \( l \) or a combination of locks that include \( l \), respectively. The functionality of a lock cell resembles that of an Access Control List (ACL). The main difference being that, an ACL, since it actually represents a column of the Access Matrix, specifies which subjects (users) have what access to an object and thus it is bound to a specific object, whereas a lock cell is an independent list of registered subjects that can be referenced by any object, in combination with other locks, to provide partial access control to that object.

**Lock expressions.** Whereas individual locks control access by managing key sets that open them, there is a further requirement of combining locks for the purposes of authorising access on a combination of conditions. Traditionally such restrictions have been implemented by introducing multiple membership of users (agents) in access groups. This is a highly centralised solution whereby the appropriate grading of authority is kept consistent by the central administrator of the system. It is also prone to management errors; for instance, a user may be removed from a group that accesses a low-security resource while leaving him in another group that has access to a high-security one of the same nature. In such systems, it is impossible to express the implicative relation between access privileges (i.e. if \( x \) is authorised to access \( A \), he is automatically authorised to access \( B \), but not vice versa). Access calculation can still be done by some database, new privileges worked out and then the access relation as a whole updated to the new state. In this chapter we are proposing the following fully decentralised solution.

A **lock expression** is a Boolean expression on locks. It follows the standard set theoretical notation:

- \( l_1 \cup l_2 \) union of locks. The result is a lock that can be opened by any key that unlocks \( l_1 \) or \( l_2 \).
- \( l_1 \cap l_2 \) intersection of locks. The result is a lock that can be opened by any key that unlocks both \( l_1 \) and \( l_2 \).
- \( l' \) lock complement. The result is a lock that can be opened by any key that does not open \( l \).
This algebra is homomorphic to the set algebra on key sets, which means that the set of keys that unlocks a composite lock can be worked out from the set of keys that open the corresponding primary locks by interpreting the lock formula as a set expression. A full isomorphism is not required since there may be more keys that open a given lock than any lock expression may suggest. This opens up a possibility of having master keys, security control keys, etc.

### 5.3 A generic authorisation scheme

The general mechanism just described is based on the assumption that there is always a single owner of a token. However, it cannot represent accurately certain scenarios that are likely to arise in a distributed processing environment. For example, let us assume that user \(x\) has sent a token to cell \(A\) that belongs to user \(y\), in order to invoke a specific method on that cell. Let us also assume that the method being invoked requires some data from another cell and for this reason a token is emitted from cell \(A\) to cell \(B\). Immediately there is a question of who should be the owner of the token sent from \(A\) to \(B\). Is it the user \(x\), who was responsible for invoking the method of cell \(A\), or the user \(y\) who is the owner of the method that sent the token to \(B\)?

All existing systems avoid tackling this problem by declaring that every object has a single owner. This position can lead to security problems. For instance, if the token in our example is tagged with the UID of \(x\) and is authorised to access cell \(B\) then instantly a method belonging to someone else (\(y\) in our example) acquires the capabilities of user \(x\). Alternatively, if the token is given the UID of \(y\) and \(y\) is authorised to access cell \(B\), then the facts that firstly it is \(x\) who initiated the execution thread and secondly that the results of the method invoked in \(B\) could be going back to \(x\) are hidden from the cell \(B\). If \(B\) was aware of this information it might have rejected the token it had received from \(A\).

A novel solution to this problem is to observe that the ownership of a token can be described accurately by a path of UIDs. For instance, assuming that the symbol '||' is used to separate UIDs, the ownership of the token in the previous example is:

\[x || y\]

This path, which is similar to the hierarchical pathnames encountered in UNIX and Windows, shows that the token has been emitted from a cell belonging to user \(y\) which was activated by a cell belonging to user \(x\). This notation enables more accurate
authorisation since it presents to the target cell the token’s exact ownership ‘path’. Hence, the lock tables of the cells, instead of assigning lock expressions to individual or group UIDs, can associate authorisation criteria with ownership paths by providing an ownership path pattern. For example, if a lock table contains the row $a \parallel b$, $l_1 \cap l_2$ then any token coming from a cell belonging to user $b$ which has been activated by a token coming from a cell that belongs to user $a$, must satisfy both locks $l_1$ and $l_2$ in order to be authorised. In the general case, ownership paths are matched against a conventional regular expression that uses UID symbols, the concatenation symbol as well as the standard Kleene closure, alternation and “wild card”. Similarly, the lists maintained by the lock cells can register regular expressions for activation paths rather than individual UIDs.

Note that the responsibility for assigning the appropriate ownership path to tokens is given solely to internal mechanisms of the system. The reason being that ownership of a token is the primary authorisation vehicle. To give a cell the ability to ‘set’ the owner field of a token would render the system vulnerable to security attacks. The authorisation process can depend, and indeed in the real life it will, on more complex and even more sensitive information. Consequently, the authorisation criteria appearing in a cell’s lock table can be much more complex and extensive than the lock expressions described in the previous section.

The overall authorisation mechanism is depicted in figure 5.2.

![Figure 5.2 Overall security mechanism](chart)

When a source cell wants to send a token to a target cell, it concurrently emits tokens containing security information to the input nodes of the appropriate authorisation graph. The target cell, upon receipt of the token from the source is not activated automatically but it awaits for verification of authorisation. Upon termination of the activity in the authorisation graph, its final node(s) send the results of the authorisation process to the
target cell. The target cell evaluates these results according to the criteria of its lock table for that user and if they are matched the token is given authorisation to activate the cell.

5.4 A security example

Let us assume that a cell $A$ represents a bank account held jointly by users $A$ and $B$. Every transaction, such as withdrawal from the account, needs to be authorised by both holders before it can be executed. In such a situation, cell $A$ must be equipped with a lock that needs the combination of two keys for unlocking. One of these keys is given to user $A$ and the other to $B$. The security mechanism of this example is illustrated in figure 3. Three cells are created, forming a simple dataflow graph. When $A$ needs to perform a transaction (say a withdrawal of £500), it sends two identical tokens: one to the bank and another one to $B$ for approval. The tokens carry the account number, the transaction number in $A$’s books and the amount to withdraw. Agent $B$ uses its authorisation logic (possibly comparing the amount with agreed spending limits, etc.) to figure out whether the transaction should be authorised. It then sends a similar token to the bank placing the authorised withdrawal limit in the data field (in this example the full amount £500). The bank receives two tokens, one from $A$ and one from $A/B$, for which the appropriate locking structure has been set in the table. Note that neither key was divulged to the other party and there was no chance of an agent external to the transaction to interfere in any way. Even causing a deadlock by sending a token from a third party to agent $B$, which might be forwarded to the BANK in error and never matched with a token from $A$ is totally impossible: the BANK access control patterns will reject it even before the business code has a chance to analyse its content.

The figure 5.3 shows agent $B$ initiating another transaction concurrently with the first one. The second transaction is a request to withdraw £300 from the account which is authorised up to a level of £200 pounds. It is clear to see that these transactions are independent as they use different tags. Since the BANK agent is sequential (as is any agent in MASIC (chapter 3)) synchronised tokens will be processed one after another.
BANK

Synchronisation

Tag = Account/TransactionNumber
Synch Expression = "Receive 2 tokens"
Action = "Get minimum of amounts in token’s data fields"

Access Control

\[ A/B \rightarrow L_B \]
\[ B/A \rightarrow L_A \]
\[ A \rightarrow L_A \]
\[ B \rightarrow L_B \]

Agent A

A#5643/A101#-£500#
B#5643/B202#-£200#

Agent B

A/B#5643/A101#-£500#
B#5643/B202#-£300#

Figure 5.3 Security protocol example

As this example illustrates, our approach to security has some strategic advantages:

- It is user-defined. A user can use the primary mechanisms, described in the beginning of this section, to create arbitrary dataflow graphs that impose a customised security scheme to a cell or a group of cells (security programming).
- A security dataflow graph is completely distributed as its nodes can reside in different servers.
- Security mechanisms are decoupled from the code and hence can be installed, examined and validated irrespective of the program. This adds extra reliability to the scheme.
- The cells of a security graph operate asynchronously.
5.5 The concrete access control model

The abstract model described in the previous section is based on two assumptions:

- The system is deployed as a network of dedicated trusted servers. In other words it is assumed that there are no malicious hosts in the system.
- There are safe interpreters for every language a user can use to encode his programs and insert them into the cells. These interpreters prevent an executing cell from accessing directly the underlying operating system.

Both assumptions are quite realistic. There is already a number of widely available safe interpreters such as JAVA, Safe-Tcl, Python, etc.

Figure 5.4 shows the proposed concrete access control model, which clarifies and extends the abstract model of figure 5.1. The access control architecture is divided into two layers: The access control functionality embedded within the cell and the set of simple and composite lock cells. As can be observed from figure 5.4, cells can reference a combination of simple and composite lock cells for input and/or output access control. Furthermore, the cells can access and manage the contents of the lock cells via a standard user and administrator interface that all lock cells implement. The simple lock cells are aware of the composite lock cells that depend on them so that they can update the...
contents of the appropriate composite lock cells when a change occurs in their contents (automatic consistency). As it was mentioned earlier, the composite lock cells need to be co-located with the simple lock cells they are based on. In this way the update traffic is localised and the period of inconsistency of a composite lock cell when a change occurs in one of its constituent simple lock-cells is minimised. Below an analysis of these two layers is provided:

5.6 Security features of cell

The access-control related functionality of a cell serves two purposes:

a) to provide and enforce input and output access control to its encapsulated user functionality;

b) to participate in the authentication procedure of incoming and outgoing messages.

Authentication is a necessary prerequisite for authorisation. Any access control scheme has to be based on a reliable, secure authentication mechanism.

The authentication mechanism of this system is based on the notion of ownership graph. Each message exchanged between cells has its ownership graph in the form of a string appended to the header. The purpose of the ownership graph is to describe the principals of the message and the relationship between them. There are two possible relationships between the principals of a message. The first one is collaboration. A collaboration relationship among \( n \) principals indicates that these principals provided partial inputs, which were combined to form a complete function invocation. The second relationship is nested invocation. Two principals \( A \) and \( B \) form such a relationship if a cell that belongs to \( A \) invoked a function from a cell that belongs to \( B \). Any message that the invoked function sends during its execution (further nesting of invocations) will be appended to the ownership graph showing that this message is the result of principal \( A \) invoking a function that belongs to principal \( B \) which in turn requests the invocation of another function at another cell. This relationship is similar to the composite delegation scheme in CORBA.

The collaboration relationship between two principals \( A \) and \( B \) is represented by the symbol \( \oplus \). The nested invocation is represented with the symbol \( \uparrow \). For example the string \( (A,B,C)/D/E \) represents an ownership graph of a message that was produced as a
result of principals A, B and C collaborating to invoke a function belonging to principal D which in turn invoked a function belonging to principal E.

Each cell is responsible for ensuring that every outgoing message contains the correct ownership graph. The user-defined functions of the cell do not have access to, and as a result cannot modify, the ownership graph. The only entities of the cell that can read and modify the ownership graph are the Input Access Controller (IAC) and the Output Access Controller (OAC) respectively. There are two simple rules which govern the transformations of ownership graphs:

a) the reply message a cell sends in response to a message with graph $G$ possesses the same graph $G$ as the request,

b) any message that a cell sends while executing a function that was invoked by a message with graph $G$, will have a graph of the form $G/H$ where $H$ is the ownership graph of the cell itself.

The novelty of the described scheme lies in the fact that the ownership graph of a message is more informative that the simple list of principals utilised in existing systems. The fact that a new relationship (collaboration) is represented in the graph means that a cell can enforce more complex access control schemes where, for example, certain collaborations are allowed and the others are rejected.

Furthermore, more complex security protocols can be implemented by accepting messages, where untrusted principals provide partial input, but rejecting others where the same untrusted principals participate in a simple nested invocation. For example, a simple graph of the form $(A,B)/C$ means that the results of the processing in $C$ are guaranteed to be returned to both $A$ and $B$ whereby a graph of the form $A/B/C$ indicates that the results of the processing will be returned to $A$ after being further processed in $B$. If we assume that principals $A$ and $C$ are trusted but principal $B$ is not, then the target cell could accept the former graph since it guarantees the integrity of the results sent back to $A$. The latter graph could be rejected because the results from $C$ may be modified maliciously by $B$ before they are sent back to $A$, or not sent to $A$ at all.

From the above analysis it is clear that the ownership graph of a message must form the basis for the authorisation of messages in cells. The IAC and OAC modules decide whether an incoming request is authorised to access the cell and whether an outgoing message is permitted to be sent, respectively. The existence of the OAC is another novelty of the proposed architecture. Access control can now be source based instead of target based which is offered by existing approaches. The OAC allows the principal of a
cell to forbid any communication attempt with untrusted principals or cells. In the case where the "no trust" is mutual (bi-directional) between source principal and target principal, this scheme stops messages at their source and thus improves error localisation. As both IAC and OAC can be decoupled from the business code, the maintainability of the application is improved. By contrast, the CORBA standard defines a client-based part in secure invocations. However, its scheme is centralised and oriented towards providing a required "quality of protection" which does not include the capability of rejecting the transmission of messages based on their destination.

The IAC contains the tables of figure 5.5:

<table>
<thead>
<tr>
<th>Message type</th>
<th>Authorisation expression</th>
<th>Security info</th>
<th>Security value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access path</td>
<td>Security value</td>
<td>Privileges</td>
<td></td>
</tr>
<tr>
<td>Privilages</td>
<td>Operations</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.5 Data structure of the input access controller

When a message arrives, its characteristics (i.e. ownership graph and other message attributes) are matched against the first table to determine what authorisation expression (lock expression) the message has to satisfy in order to be granted access to the cell. The IAC may utilise security-information caches or contact the relevant lock cells directly to acquire items missing in the cache. If the authorisation expression is satisfied then the message is assigned a security label. This label, together with the access path [6] that the message has followed to reach the agent determine its privileges. Finally it is checked whether the function that this message requests to invoke is permitted, based on its privileges. The IAC can also reference an input-guarding lock cell. The referenced lock cell acts in this way as an input security proxy of the cell. The IAC also implements a standard interface via which authorised principals can modify its tables and settings.

In a similar way when an executing function of the cell produces an outgoing message, then this message activates the OAC. The OAC decides whether the message according to its destination can:

a) be allowed to be sent or
b) be denied transmission or
c) be forwarded to a nominated lock cell which acts as output security proxy for the cell.

If the message is permitted to leave then the OAC can specify any security information, pertinent to the specific target cell, to be added to the message (i.e. capabilities, access certificates that the target has provided to the source cell, lock cells to be contacted, etc.). The OAC also possesses a standard interface via which its contents can be customised.

Lock cells

The lock cells in general act as independent authorisation centres. They are classified into two categories: the Simple Lock Cells (SLC) and the Composite Lock Cells (CLC). All lock cells implement a standard interface that enables:

a) specific principals or graphs of principals to register themselves by following a predefined registration procedure and

b) specific cells to register pairs of ownership graph patterns and associated values to be used for proprietary access control. Each such pair has an owner field in order to distinguish between entries belonging to different cells/principals.

There are two different types of simple lock cells:

1. Binary lock cells. These lock cells contain a list of ownership graph patterns. They act as binary security switches. When they are requested to authorise a message they first check whether the target cell has registered any entries in their list. If this is the case the ownership graph of the message is matched against the sub-list of graphs that belong to the target cell. If a match is found then they send the notification that the message is authorised.

2. Layered lock cells. These lock cells contain a list of pairs of ownership graphs and values. The values belong to a range of security values the specific lock cell utilises. Different types of layered lock cells can be defined based on the range of security values they associate with the ownership graph patterns. Their functionality is similar to that of the binary lock cells with the difference that they also return the value associated with an ownership graph if a match is found. The layered lock cells act as labelling mechanisms for messages. The target cell of an invocation can utilise the label that a certain layered lock cell assigned to the incoming message as an input for its own proprietary access control mechanism (IAC).
Both types of simple lock cells also include a data structure, similar to that which the OAC maintains, in order to provide output access control. Furthermore, the SLCs maintain a list of CLCs that are based on them so that they can notify the composite cells of any changes to their contents.

The CLCs behave like SLCs when they are requested to authorise a message. Each CLC implements a lock expression. Their contents (input and output access control structures) are produced by combining the contents of their component simple lock cells such that each entry in their data structures satisfies the lock expression they implement.

Four simple rules are applied to determine the contents of CLCs:

a) one entry in an SLC $A$ matches an entry in SLC $B$ if their associated ownership graphs are identical;

b) if an identical entry occurs in two SLCs $A$ and $B$, then it will occur in both their intersection and union,

c) if an entry has a different associated security value in $A$ and $B$ then this entry will occur in the intersection with the smallest of the two security values whereas in the union it will be associated with the highest value of the two.

d) complement CLC can only depend on SLC directly (this can always be achieved by using de Morgan’s transformations on the lock expressions); a complement flag is used to mark up the fact that successful matches should cause access denial.

Cells can create simple and composite lock cells by sending an appropriate request to the lock factory. The functionality of the lock factory is dual:

a) It acts as a lock cell type repository. The type of an SLC is defined as an ordered list of security labels it can assign to the ownership graphs that will be registered in it. In reality the implementation of the lock factory will provide an interface to an object-oriented system like CORBA. In this way, the lock cell types can be defined in a standard definition language such as the CORBA IDL. Another advantage of such an implementation is that new types can be created by using existing types (inheritance).

b) It ensures that a CLC is created only if the types of the component SLC are identical. For example it would be meaningless to attempt to create the intersection or union of two SLCs that use different security label system since it is not possible to apply the aforementioned four rules to determine the contents of the new CLC.
5.7 An object-oriented view of the architecture

The object-oriented view of the locking scheme is presented in figure 5.6, which displays an abstract lock cell object, the lock factory object and three interfaces. The users can only access the lock factory and any existing lock cells. The simple and composite lock cell interfaces and the abstract lock cell object are internal system entities, which are opaque to the users. All new lock cells inherit the attributes of the abstract lock cell object and implement the respective interface. The abstract lock cell object interface contains a Java “properties” table, the abstract method “authorise” and the method “return_type”. The method “authorise” is implemented by the respective interface, every time a new lock cell is instantiated. The layered LC interface extends the binary LC interface, by adding one updating method and overriding an existing one. The lock factory keeps an array with the names of all the available types of locks. Its “create_LC” method is used to create either a simple or a composite lock cell. The methods of a lock cell are divided in three basic categories: administrative, updating and informative. The composite lock cell interface introduces 3 methods (“return_expression”, “update_LC_contents”, “calculate”). If one of the participating lock cells – in a composite lock cell – changes an entry, it sends an “update_LC_contents” request. The composite lock cell will update the stored LC_table of the particular SLC and then recalculate the table.
5.8 Conclusions

In this chapter the architecture of a distributed access control system for mobile agents was presented and analysed. The architecture is based on a natural abstract model, which satisfies the principles of simplicity, customisation and security knowledge sharing and reuse. The agents can create and dynamically manage their own access control policy, parts of which can become public and shared with other agents. As a whole, the proposed architecture provides a simple, customisable base that mobile agents can use to program their security requirements as an individual or as a group.
Chapter 6

Prototype client side

6.1 Introduction

The client part of a distributed computing system is considered to be an important, integral part of the system itself because it provides the interface to the system's services. A successful system design must always provide a comprehensive, uniform and user friendly interface via which the potential system users can efficiently access and use the system. The end users are after all a critical factor which can determine the overall success or failure of a system. Existing bibliography suggests that the developers of mobile agent systems have acknowledged the importance of the system interface and have started incorporating new or more sophisticated user interfaces into their systems.

This chapter focuses on the design and implementation of a prototype GUI for the MASIC architecture. The presented GUI improves the overall system usability and minimises potential unacceptable input from the users. The expected requirements, the design methodology and a short description of the main window of MASIC are presented. Furthermore the implementation of the GUI is also outlined.

6.2 Design Methodology

The methodology followed for the design was that of prototyping. The prototyping method is illustrated in figure 6.1:
The first stage of the prototyping method is to define the requirements that the design of the system has to satisfy. The second step is to design a limited version of the system which is then given to the end users to evaluate. The users provide relevant feedback to the system designer who is responsible for incorporating the suggested changes into the design. The system designer and the end users might participate in a number of recommendation loops until the design reaches its final form. The prototype design is then implemented, tested for bugs and assessed for any possible improvements.

Figure 6.1 Prototyping method
6.3 GUI Design Requirements

The main specifications the interface should meet were as follows:

- The user should be capable of connecting and disconnecting from the server of the prototype system implementation.
- The interface should offer an interactive console (like UNIX command prompt). This console should display messages (answers to queries) that cells send to the user as well as any information or prompts from the system itself.
- Cell management. The user should be able to create and delete cells from the system. In this prototype there are only two types of cells: the data cells and the program cells.
- Cell communication. The interface should enable the user to communicate directly with any active cell in the system by means of message exchange.
- Cell Information. The user should be informed of which cells are currently active in the system.
- Cell connections. When a user wishes to communicate with a cell then he always needs to specify that cell's address as the destination of the token he will send. This procedure is very inefficient if the user wishes to engage in a dialogue with the same cell. For this reason the interface should enable the user to connect to a cell and send commands to it without the need of specifying the destination. The interface should append automatically the appropriate cell address. The whole procedure should be transparent to the cell itself. Furthermore the user should be allowed to connect to more than one cell simultaneously and be given tools and information to assist him in managing these connections.

6.4 GUI Design Process

The initial prototype design was very simple. It consisted of:

1. one textArea occupying most of the area of the window. This textArea was acting as the interactive console displaying messages from the cells or the system itself.
2. a number of menus. The control menu included all the commands for GUI control and cell management (i.e. create/delete cell, clear display, etc.), whereby the
communicate menu included all the commands for cell communication such as sending tokens, connect/disconnect from cells, etc.

3. a number of buttons to enable the user to load and run program scripts. The file and edit menus contained typical file processing choices, such as file open, save, close, etc., to facilitate the process of program writing, editing and execution. Figure 6.2 shows an early version of the main window.

![Figure 6.2 An early version of the main window](image)

The described initial prototype design was then delivered to two final year students acting as end users of the system for evaluation. The feedback from the students included the following main points:

1. The prototype satisfied the main design requirements as they were set out in section 6.3 but the interface was not adequately user-friendly. More shortcut buttons were needed to enable the user to access fast the most common commands. Some aesthetic modifications were also proposed.

2. The contents of the menus needed reorganisation. It was suggested that the functions related to different types of cells be included in separate menus. Furthermore it was recommended that the file and edit menus be merged into one menu since the file processing commands do not constitute the main functionality of the interface.

3. The end users identified the possibility that the interactive console of the interface could be overflowed with information if a number of cells sent messages to the client.
concurrently. It was suggested that the user should have some message viewing control which will enable him to define which messages he wishes to view. The rest of the messages should be kept in a queue until the user decides to either view them or delete them.

After taking into account the aforementioned feedback, the file menu and the help menu were the only ones that were kept from the initial design. The edit menu choices were placed under the file menu. The rest changed to four new menus: the data cell, the program cell, the token and the connected cells. The first one offering all the functionality for data cells, the second one for program cells, the third one for sending and receiving tokens and finally one for handling the connected cells. After testing the window it was proven to be better to add a text field at the bottom of the window so that the user can send commands to cells he is connected to. Moreover, an on-line list was added at the right hand side of the window so that the incoming tokens could appear there. It is better having an on-line list so that incoming messages appear there instead of having to go to the token menu and select receive tokens and then check out if any tokens have arrived. The user can see the origin of the messages and can decide to either view them or delete them. The central console window displays automatically only the replies from cells the user is connected to. Furthermore two buttons were added so that the user can quickly connect/disconnect from cells. Figure 6.3 shows final layout of the interface:
6.5 Description of GUI

In this section the main window of the interface will be described more extensively. Figure 6.4 shows the main components on the main interface window.

![Figure 6.4 The main window of the interface](image)

The components of the main window

1. The active program.
2. The Menu Bar.
3. Main Screen.
5. Message List.
7. Connect/Disconnect Buttons.
8. Program Buttons.
9. Indication (online/offline).

Below figure 6.4 there is a key list to the components, and a short description. Number 0 is a choice box from where the user can select a program already loaded. Component 1 is the menu bar. There are six menus in the menu bar: the File, the Data
Cell, the Program Cell, the Token, the Connected Cells and finally the Help menu. Below is a list with all the menu choices under each menu.

<table>
<thead>
<tr>
<th>File</th>
<th>Data Cell</th>
<th>Program Cell</th>
<th>Token</th>
<th>Connected Cells</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>Info</td>
<td>Create</td>
<td>Create</td>
<td>Send</td>
<td>View Cells List</td>
<td>Help</td>
</tr>
<tr>
<td>Connect Server</td>
<td>Delete</td>
<td>Load</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disconnect Server</td>
<td>Connect</td>
<td>Run</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td>Disconnect</td>
<td>Terminate</td>
<td>Run-All</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Active Cells</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Remove</td>
<td></td>
<td></td>
<td></td>
<td>Help</td>
</tr>
<tr>
<td></td>
<td>Remove-All</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the first menu there are four menu items. The first one, “Info”, returns information about the operating system version, the default language, the platform type etc. The next item, “Connect Server”, is used for connection to the server. The “Disconnect Server” disconnects the client from the server. The “Exit” disconnects from the server and exits the program.

The next menu is the Data Cell, which consists of five menu choices. The first choice, “Create”, brings up an online form for creating a data cell. Figure 6.5 shows the online form. The user has to give the general attributes of the cell and then he can give the data/methods or the access control list, by clicking the relevant button. Also there is a cancel button, so that the user can return to the parent window without making any changes. In almost every window, wherever it is necessary, there is a help button, which provides a short help on the current window. Finally the “Create” button sends a request to the server for the cell creation and returns to the parent window.
The next choice is "Delete". It brings up a list with all the active cells so the user can select one and delete it. The idea of choosing from a list instead of giving the details of the cell in a form makes the task easier and minimises the possibilities for errors. Also, in the "Data Cell" menu there is a "Connect" choice. It brings up a small form providing a list of the available cells for connection. On the other hand, "Disconnect" brings up a list with the already connected cells. Figure 6.6 shows the Connected Cells List window.

"Program Cell" is the next menu. The "Create" item brings up exactly the same form as the data cell create item, with the only difference that in the Data/Methods form the user can also give the source code of the program. "Load" brings up a LOAD Dialog window, for loading a program. Then the name of the program is added to the program list, represented in figure 7.4 with number 0. "Run" runs the selected program from the
program list. If the user wants to terminate the program he has to select "Terminate". Similar to "Run", "Run-All" runs all the programs listed in the program list.

The next menu is the "Token", where the user can send a token to any cell, by just giving the destination. Figure 6.7 shows the "Send Token" window. The user has to provide only the destination, the data and some security information.

![Figure 6.7 Send Token form](image)

The number 0 component is the console where information from the server appears. When the program starts a welcome message is displayed and verification is shown that connection to the server has been established. The main screen in combination with number 0, the command line, acts like an interactive console. The user can send commands from the command line and the output will appear on the main screen. Figures 6.8 and 6.9 show the main window before sending a command and after.
Welcome to MAIC version v0.9a
Local Host is: kenny.ee.surrey.ac.uk/131.227.50.40
Connected to the server...at port: 2525
2525->Welcome to MAIC server v0.4 running on Solaris
Connection has been made to cell..., with GUID 34310

Figure 6.8 Sending a command to a connected cell

Welcome to MAIC version v0.9a
Local Host is: kenny.ee.surrey.ac.uk/131.227.50.40
Connected to the server...at port: 2525
2525->Welcome to MAIC server v0.4 running on Solaris
Connection has been made to cell..., with GUID 34310
zap1 => file1
zap2 => file2
zap3 => file3

You are connected to a Cell with GUID: 34310

Figure 6.9 Answer appears on the main screen
The idea of combining the `TextArea` and `textField` java classes to create an interactive console was an attempt to offer the user the ability to send commands to connected cells without having to fill in forms. The command line is only available if the user is connected to at least one cell and that cell is “active”. Number 0 is the message list. It shows all the incoming tokens. Through that list the user can see all messages of the incoming tokens. The list is updated automatically as a new token is received. Then the user can use the Message Buttons, number 0 to access the incoming messages. Number 0 shows the two buttons “Connect” and “Disconnect”. Both are shortcuts of the menu choices, under the “Data Cell” menu, described earlier in this section. The reason for the shortcuts is that both these functions are used frequently and thus they should be quickly accessible. If there are no connected cells, the “Disconnect” button is disabled.

Task bar is number 0. It informs the user which of the connected cells is active at the moment. If there is no active cell then the task bar prompts: “You are not connected to a cell...”. All the commands that the user sends from the command line apply to the cell shown in the task bar. Under number 0 there are three buttons: “Load”, “Run” and “Terminate”. All three are shortcuts of the choice menus described earlier in this section. On the top right corner of the windows, under the “Help” menu, it is number 0, the connection indicator. It is an animation that shows the connection to the server is active when it runs and that there is no connection to the server when is stopped. Finally, the “Clear” button, number 0, is used for clearing the contents of the main screen. Although the main screen scrolls down when the data exceed the screen, the user can clear the contents at any time using this button.

### 6.6 GUI Implementation

MASIC v0.9a was implemented using java JDK 1.1.3. The computer used for the implementation was a Sun SparcStation 10 running on Solaris 2.5.1. We decided to use Java because it is a powerful object oriented language which includes a package for GUI implementation. The platform that the project was implemented on did not support any other GUI design package. Of course a GUI could be designed and implemented using the standard X library (Xlib), the C Language programming interface to Version 11 of the X Window System but that would restrict the interface to run only under X Window...
systems. On the other hand, a Java implemented interface can run (theoretically) on any Java enabled platform.

6.7 Limitations of the implementation

In the current version the model has some limitations, some assumptions that the user should be aware of. Firstly, it is supposed that the server is running under garak.mcs.surrey.ac.uk and listens to port 2525 for connections. Port 2525 was found by a scanning program to be free. We tested it for a long time and found there were no problems caused, so it was decided to use it as the communication port between the client and the server.

Moreover, although the service is available from the client side, no program cells can be created. The only types of cells supported by the current version are:

- **File Managers**
  A cell responsible for managing a group of files. It has some default methods for the manipulation of the files, but the user can add more. At the current version only the default methods are supported, which are: showdata, showmethods, value, showalias, totalwords, mv, cp, rm, adddata, renalias, deldata.

- **Set Managers**
  The primary function of a set manager is to maintain and update a list of aliases. If a cell is registered in a set manager then effectively the cell has created a new access path to itself via this set manager. The set manager checks whether any token passing through has sufficient authorisation to proceed. If yes, then the token is allowed to proceed towards the target cell, if not the token is discarded. At the current version only the default methods are supported. But the user will be able to add more in a future version. The default methods are: smreg, smunreg, showsm, smupd.

- **Lock Cells**
  Lock cells keep a track of user names and the authorisation values. Lock cells have only one method, the showlc.
6.8 Discussion of source code

In this section a description of the classes and explanation of the algorithms used is given. Parts of the code are listed here for quick reference. The program consists of 18 code files (*.java). These files are listed below followed by a short description.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maic_client.java</td>
<td>The main file of the program. Creates the main window.</td>
</tr>
<tr>
<td>AboutDialog.java</td>
<td>Creates the window for the About information.</td>
</tr>
<tr>
<td>ConnectDialog.java</td>
<td>Creates the window for the authorisation check when the program starts.</td>
</tr>
<tr>
<td>CreateCellForm.java</td>
<td>Creates the online form for creating a Data Cell.</td>
</tr>
<tr>
<td>CreateProgramForm.java</td>
<td>Creates the online form for creating a Program Cell.</td>
</tr>
<tr>
<td>DelMes.java</td>
<td>Creates the window to delete a message.</td>
</tr>
<tr>
<td>PMethData.java</td>
<td>Creates the window for importing the data and methods of a Program Cell.</td>
</tr>
<tr>
<td>TextEditApp.java</td>
<td>A simple text editor.</td>
</tr>
<tr>
<td>UUEncode.java</td>
<td>Encodes any string with uuencode.</td>
</tr>
<tr>
<td>ViewActiveList.java</td>
<td>Creates the window with the Active Cells List.</td>
</tr>
<tr>
<td>ViewConn_List.java</td>
<td>Creates the window with the Connected Cells List.</td>
</tr>
<tr>
<td>checkConn_List_empty.java</td>
<td>Creates a thread that checks if the Connected Cells List is empty.</td>
</tr>
<tr>
<td>clientSocket.java</td>
<td>The thread that runs the connection with the server.</td>
</tr>
<tr>
<td>connectCell.java</td>
<td>Creates a window for connecting to a cell.</td>
</tr>
<tr>
<td>help.java</td>
<td>Creates a window with help for opening a help file.</td>
</tr>
<tr>
<td>mes_Left.java</td>
<td>Creates a window to inform that messages are still in the message list.</td>
</tr>
<tr>
<td>sendToken.java</td>
<td>Creates an online form to send a Token.</td>
</tr>
<tr>
<td>warning.java</td>
<td>Creates a window displaying a warning message.</td>
</tr>
</tbody>
</table>
6.8.1 Authorisation check

The main_client.java file creates the main window. Upon opening, it calls the public static void main(String args[]) method which creates the main window, connects to the server and finally calls the authorisation() method to check the authorisation. The process for the authorisation check calls the authorisation() method which creates an instance of the ConnectDialog class which is responsible for checking the username and password, and sends an authorisation request to the server. The password is encrypted using the uuencode encryption mechanism and is then sent together with the user name to the server. The token that carries the information has the following format:

    Authorisation_check#username#password!!

Then the system waits for an authorisation_check_suc token from the server to proceed with the main program. If the authorisation check fails it notifies the user that there was an error in the username and password given so that the user can try again. The password is passed to the uuencoder through a StringBufferInputstream and it returns encrypted into a file. Then the program reads the encrypted password from the file and sends it to the server. The uuencoding encryption of the password is clearly not secure and it was used only for demonstration purpose.

6.8.2 Handling incoming tokens

After the authorisation check has succeeded the connection is established and the GUI is ready for user input. The whole connection runs as a single thread, so the rest of the program can continue receiving user input without blocking. If the connection is not a separate thread then the user cannot do anything else until the socket stops listening for incoming messages.

The technique used here for handling incoming tokens is based on the event loop model. There is an infinite loop running, as long as the thread that holds the connection is alive. The incoming message is copied to a string and depending, on the format of the string, the system acts respectively. Figure 6.10 shows the part of the code confirming that the connection is established and the loop that the program uses to listen for incoming tokens is running.
The **String theTime** represents the incoming token. This string is checked from the `if` statements, in the figure below, to find out the type of the token. At the end of the `if` statements there is an `else` statement which handles all the unknown types of tokens.

```java
public class clientSocket extends Thread {
    public int port;
    Socket theSocket;
    DataInputStream theTimeStream;
    public clientSocket(int port)
    {
        this.port = port;
    }
    public void run()
    {
        try
        {
            theSocket = new Socket("garak.mos.surrey.ac.uk", port);
            AllDisable();
            theTimeStream = new DataInputStream(theSocket.getInputStream());
            while(this.isAlive())
            {
                String theTime = theTimeStream.readLine();
                if (theTime.startsWith("cell_connection_suc#") == true)
                {
                    
                } else if...
                else...
            }
        }
        catch(UnknownHostException e)
        {
            output("The system cannot connect to the Server...");
            mal_client.conCell.setText("You are not connected to a cell...");
            AllDisable();
        }
    }
}
```

Figure 6.10 Creation of connection and handling of the incoming messages

Now let us see how the type checking of the tokens works. Every token has a standard format. Each field is separated from the others with a "#" except the last field which has at the end "!!" and declares the termination of the token. Any carried data are between square brackets ([data]). In order to find the token type we tested the “title” of the token, which is the first field. The `startsWith(String)` method of the `String` class is used for that purpose. It checks if the `theTime` String starts with `title#` (where `title` is
the type of token). When it finds the type of the token discards the title# and keeps the rest so that the fields can be separated.

6.8.3 The main window

The main window is created from the maic_client class constructor. There is no layout manager used so that each component can be placed anywhere on the window. This is a more difficult way of creating a GUI, but it allows the programmer to arrange components, in any possible combination. Below is a list of the methods that class maic_client has:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public void addfile(String file)</td>
<td>Adds a program to the program list</td>
</tr>
<tr>
<td>Public void removefile(String file)</td>
<td>Removes a program from the program list</td>
</tr>
<tr>
<td>Public static void output(String out_mes)</td>
<td>Prints out_mes to the main screen.</td>
</tr>
<tr>
<td>Public void runfile(String file)</td>
<td>Runs a program in a separated process.</td>
</tr>
<tr>
<td>Public void connect()</td>
<td>It creates an instance of the connectCell class, for connecting to a cell.</td>
</tr>
<tr>
<td>Public void viewmessage()</td>
<td>It shows the contents of the selected message, from the message list, in a window.</td>
</tr>
<tr>
<td>Public void checkmes()</td>
<td>Checks if the message list is empty.</td>
</tr>
<tr>
<td>Public static void Terminate()</td>
<td>Starts the termination of the program.</td>
</tr>
<tr>
<td>Public static void RestoreCells()</td>
<td>Restores the saved cells, from a file.</td>
</tr>
<tr>
<td>Public static void authorisation()</td>
<td>Starts authorisation check mechanism.</td>
</tr>
</tbody>
</table>

The main window waits for user input. When an event occurs the event handler will take care of that event and perform the right task.
6.8.4 CreateCellForm and CreateProgramForm classes

This class creates an online form, whose purpose is to create a data cell. The user has to give the name, the type, the location and the Life-Time of the cell. There are two buttons that create new forms. One for the data and methods and the other for the Access Control List. In the data/methods form there are two textAreas, textData and textMethods, where the user can fill in the values. When the OK button is pressed the contents of each textArea are stored in another textArea, data for the data and meth for the methods, which play the role of a buffer.

The Access Control List works in almost the same way. This has three fields, one for Authority, one for Access Path and one for Security Expression. The difference is that the user has to fill in three fields and then press the Add button so that all the fields will be added to the list and the fields cleared for new entries to be given. If the user tries to add an entry without completing all three fields the program will notify him to complete all of them and then try again. Also, there is a remove button so that the user can select an entry from the list and remove it, in case he has made a mistake. Again after the OK button has been pressed the entries of the list are copied to a textArea called list, which plays the role of a buffer.

Finally when the Create button is pressed to the main Create Cell Form window, the program performs a check to find out if a cell already exists with the given name. Every time a cell is created, a file with the attributes of the cell is saved on the current directory. The name of the file has the following format: username_nickname.inf. The program checks if a file exists with the given name. If there is no other cell with the same name (nickname) the program checks that all fields are filled and if they are then sends a request to the server for a cell creation, otherwise it notifies the user to fill all the fields of the form. In every window there is a Help button, which creates an instance of the help class, which we will discuss later.

The CreateProgramForm class has the same general structure as the CreateCellForm class. The main difference between the two classes is that the Data/Methods window for the CreateProgramForm class contains the button Import Source Code which creates a new instance of the TextEditApp class (simple text editor) for the user to import the source code of the program that the cell represents.
6.9 Conclusions

In this chapter the design and implementation of a prototype graphical interface for the management of the agent architecture of this thesis was presented. The interface does offer some useful functionality but its current implementation is rather limited. It serves more as a guide for the future development of the complete agent architecture rather than as a complete client-side software. Nevertheless this prototype interface illustrates that it is feasible to enable the end user administration and management of our agent architecture via a simple, easy to use, graphical environment. At the time of writing of this thesis, this point is enforced by the fact that several agent development organisations have already started incorporating graphical interfaces into their agent systems. A typical example of this is the IBM aglets agent environment. It is the opinion of the author that the provision of interactive graphical interfaces is the only way of transforming the agent technology into a widely accepted means for doing efficient distributed computing.
Chapter 7

Conclusions

7.1 Summary

During the past decade the Internet has grown exponentially. It has been transformed from a national networking project to a global communications infrastructure which is open enough to support the deployment of distributed applications. The Internet nowadays is open to commercial exploitation and has already become the medium for global scale distributed applications (Internet computing).

The main assumption of Internet computing is that the cost of local processing is negligible compared to the cost of communication. For this reason the primary Internet computational architecture, the client-server, has become inadequate since its main mechanism, the RPC, requires ongoing communication. It was in the early 90’s when the first serious candidate to replace RPC came to the fore. The concept of making the code rather than the data mobile had a clear strategic advantage over RPC. With this technology ongoing interaction did not require ongoing communication.

Because of this important benefit hopes were raised in the distributed computing research community. Individuals and institutions from all over the world joined a research frenzy in the area of mobile agents and consequently the result has been somewhat chaotic. A large number of agent systems were produced within just five years which did not integrate with existing legacy software, nor interoperate with each other. Each system suggested its own view of the mobile agent technology. The first standardisation attempt appeared in 1997 but until now it has remained largely in the background of
developments within the field of agent research. In summary the domain of mobile agents is indeed very promising but it is still in its infancy. More work is required to transform this area into a stable, mature research domain in which real world problems could be solved.

7.2 Evaluation

In this section we will attempt to evaluate the agent framework presented in this thesis by matching its design against the requirements of section 1.2.

The proposed framework (MASIC) is architecture-based and enables the users to create component-based agents. In chapter 3 it was shown that each component of an agent can either be passive data, in which case it is expressed in XML, or active data (program or intelligent black box). In the latter case the programs can be defined in any interpreted programming language for which there is a safe interpreter. This characteristic enables the users to create agents by using their existing proprietary programs as building blocks. Moreover the users can incorporate within an agent the necessary navigation logic which is also expressed efficiently in XML. Therefore our framework is language neutral and does not define or depend on any proprietary declarative notation. Thus it can be well integrated within the current software scene.

The design of the proposed framework has taken into consideration the end user’s perspective as well (usability). In chapter 6, a graphical interface was presented via which users can interact and manage the implementation of the framework. The presented interface is simple and allows for mutual communication between the agents and the users. In general the interface supports the user creation, deletion and management of the framework’s agents.

The requirements of reusability and programming efficiency have been addressed at various points of the framework design. Firstly new agents can use the components of the existing agents to build up their own functionality (chapter 3). Thus the concept of software re-use is present within the mechanism of agent creation. Secondly both the organisation and the access control model support the creation of composite nodes by using the existing ones as building blocks (composite group managers discussed in chapter...
4 and composite lock cells discussed in chapter 5). Thirdly the cells are equipped with collaborative facilities (token store) and as a result the efficiency of collaboration is increased because it takes place where the associated action is located (chapter 3). Fourthly, the issues of communication and mobility management are integrated within a single agent system component (chapter 4). If these two issues were addressed by separate entities then extra communication would have been required since the communication and mobility management entities need to interact. Finally efficiency gains are possible because of the agent organisation model. Because the group managers act as high level message routers, unwanted messages can be discarded before they reach the destination agent. In this way this agent does not need to waste any processing time to check and reject such unwanted messages. This characteristic is important since valuable Internet bandwidth is wasted nowadays because of unwanted messages that eventually are discarded when they reach their destination.

Care has been taken to ensure the agents have well-structured representation and execution. Firstly, there are three different types of agents, any mixture of which can be used to represent user-defined programs. Agents that are highly mobile have lightweight size (token agents) and therefore require less communication bandwidth to perform their migrations. Secondly collaboration points can be interconnected with each other thus allowing the definition of distributed programs as dataflow style graphs. This program representation is desirable because parallelism can be exploited during the execution of the program (chapter 3).

The framework also supports the required services of access control, organisation, mobility management, collaboration and communication. Each of these services were mapped as either a separate layer in the framework’s conceptual model or as a layer within the agent facilitator. The services offered by the agent system are adaptive (interaction of queries with links in the group managers), customisable on a per-agent basis (individual agents can create their own organisational and access control structures as well as customise the message-passing services MARSA offers). A cell’s token store provides an efficient mechanism for defining and using collaboration points within the agents themselves. Each cell can implement its own access control policies by customising the contents of its onboard input and output access controller. Any cell can communicate with another by using the simple API MARSA offers. Via this single integrated interface, the cell can also request its migration to another physical location. The MARSA
component (chapter 4) will ensure that there is continuous access to an agent regardless of its mobility.

The applicability of the MASIC concepts, entities and mechanisms in the area of Electronic Commerce is outlined in Appendix A. In summary it is shown how simple and composite group managers can be used to structure a generic electronic market place. Secure, authorised transactions between producers and consumers are supported through the passing of all exchanged messages via the appropriate market place group managers which enforce appropriate security policies. The discovery process of available products and services is also facilitated by advertising group managers. This application, although not researched to a full extent, provides a simple electronic market place model which has some important advantages over similar models such as support for anonymity, group access control policies and development of more complex market place entities from existing simpler ones. Finally an initial investigation of the application of the MASIC framework in the area of Knowledge Management is presented in [8, 74].

However, there are five main disadvantages of the proposed framework:

1. There is no type checking mechanism. As mentioned in chapter 3, such a mechanism can be implemented by using the XML DOM but it is still slower than type checking mechanisms embedded within standard programming languages.
2. The mechanism provided to an executing program for accessing its associated data is slow compared to standard object-oriented programming languages.
3. Only a limited number of composite group managers are allowed due to scalability constraints.
4. There are no simulation results to show the efficiency of the communication system MARSA.
5. Finally, the access control model of the framework is identity-based. Again as the number of users increases it is important to define groups of users or introduce the concept of roles into the model for scalability purposes.
7.3 Future research

The presented agent framework of this thesis (MASIC) is still in the early stages of development. There are many avenues for future research. Below we mention a few of them:

- Incorporation of cryptography into the access control model. This would require that the notion of locks and keys are given cryptographic semantics. Furthermore the access control model could be extended to incorporate more security features such as auditing.

- Investigation on the automatic creation of links in the group managers. The framework in its current state assumes that the links are created explicitly by the corresponding agents. The organisation model could provide more automation in this process by using specialised discovery token agents that visit resource agents and connect them to the organisational structure automatically according to their contents.

- The MARSA component incorporates some groundwork in the area of agent communications and mobility management. Simulations and possibly the development of a simple prototype are required to evaluate the performance of this component. Further research is also needed in order to provide a mechanism for the discovery and resolution of distributed deadlock scenarios.

- The servers the framework will be deployed on need to have a common server-server protocol to support their interaction. In the same context, standardisation is also required for client-server interaction within the framework.

- The framework can be extended and provide more services to the agents such as persistence, agent life-cycle, transaction support, etc.

- Portals to legacy software. The framework could be equipped with specialised agents that act as portals between the framework and other useful legacy software such as databases, object-oriented systems (CORBA), etc.

In conclusion, the proposed framework does offer several advantages over existing language-based agent systems. It is hoped that it will offer a good template for the design of more advanced and complete frameworks which in turn will help bring the agent technology to the masses of ordinary Internet users. Mobile agents will only satisfy the
great expectations the research community has placed on them if the everyday Internet user comes in direct contact with them and experiences the benefits their use brings.
Bibliography


52. Gray, R. S., Agent Tel: A transportable agent system. Department of Computer Science, Dartmouth College, Hanover, November 1995.
58. IBM Aglets. Available at Internet URL: http://www.trl.ibm.com/aglets
60. James, R., JATLite. Available at Internet URL: http://java.stanford.edu


78. Li, W., Messnerschmitt, D. G., Itinerative Computing Using Java. Available at Internet URL: http://ptolemy.eecs.berkeley.edu/dgm/javatools/java-to-go


