WEB AGENT PROGRAMMING MODEL

By

VIDYA RENGANARAYANAN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001

To my Parents.

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Sumi Helal for his valuable guidance and support throughout the course of this thesis. I also thank Dr. Joachim Hammer and Dr. Beverly Sanders for agreeing to be on my committee.

I would also like to express my thanks to Amar Nalla for his suggestions, his great attitude in welcoming changes on the aZIMAs architecture and for his support while working with me. I would also like to thank Jinsuo Zhang, Network Administrator, and all the other students of Harris Lab who have assisted me during the various phases of my thesis.

On a more personal note I would like to thank my family and friends for their constant support and encouragement without which this work would never have been possible.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS iv

LIST OF TABLES vii

LIST OF FIGURES viii

ABSTRACT ix

CHAPTERS

1 INTRODUCTION 1

1.1 Mobile Agents 1
1.2 Mobile Agents and the Internet 1
1.2.1 Compelling Reasons for Mobile Agents 2
1.2.2 Why Are Mobile Agents Not Pervasive? 2
1.3 Goal of Thesis 3
1.4 Organization of Thesis 3

2 SURVEY OF MOBILE AGENT COMMUNICATION TECHNIQUES AND APPLICATIONS 5

2.1 Ajanta–A Mobile Agent System 5
2.2 Mobile Agent Systems in Existence Today 7
2.2.1 TeleScript 7
2.2.2 Odyssey 8
2.2.3 Tacoma 8
2.2.4 Agent Tcl 9
2.2.5 Aglets 9
2.2.6 Voyager 9
2.2.7 Concordia 10
2.3 Specialized Mobile Agent Languages and Applications 10
2.3.1 Knowledge Query Manipulation Language (KQML) 10
2.3.2 Aladin–Agent Specification Language 10
2.3.3 Safe-Tcl Language for Enabled Mail 11
2.3.4 Collaborative Interface Agents 11
2.3.5 Collagen 12
2.4 Summary 13

3 aZIMAs-AGENT SYSTEM ARCHITECTURE 15

3.1 Client Environment 16
3.2 Server Environment 17
3.3 APIs Provided for Web Agent Developer 18

4 THE WEB AGENT PROGRAMMING MODEL 19

4.1 Agent Specification Language 25
4.2 Agent Manager 26
4.3 Developing a Mobile Agent 27

5 APPLICATION AGENTS DEVELOPED USING WAPM 31

5.1 Query Application Agent 31
5.2 Filter Application Agent 32

6 CONCLUSIONS 34

REFERENCES 35

BIOGRAPHICAL SKETCH 37

LIST OF TABLES

Table Page

3.1 Multi-Part MIME (Containing the Agent) Embedded in a HTTP Request 16

4.1 DialogApplet.html 23

4.2 Template for an Application Agent Developed in WAPM 30

5.1 Usage of the Query Application Agent in the Script 32

LIST OF FIGURES

Figure Page

2.1 Ajanta Server Architecture 6

4.1 WAPM–Client Side 21

4.2 WAPM–Server Side.....22

4.3 WAPM–System Agent.....23

4.4 WAPM–Interactive Agent 24

4.5 Class Hierarchy of the Model 28

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science.

WEB AGENT PROGRAMMING MODEL
By

Vidya Renganarayanan

December 2001

Chairman: Dr. Abdelsalam (Sumi) Helal
Major Department: Computer and Information Science and Engineering

Mobile agents are software modules that can move from one place to another performing actions programmed in them. They make efficient use of bandwidth since they can move to the data rather than transfer bulky data to the point of processing. In today's world, where Internet and emails are a part of life, the mobile agent paradigm is extremely suitable since it provides an ideal medium to program distributed computing tasks.

Mobile agents have been around for more than a decade now, yet they have not made an impact in the Internet market. Most agent system architectures that provide agent development environments have heavy infrastructure requirements and a steep learning curve to build useful agents. This thesis brings together the familiar components of browsers and email to effectively enable mobile agents. A scripting language is also proposed that allows for easy coordination of agent activities.

This thesis focuses on agents that collaborate with people and uses Java's applets and serializability mechanism to couple the resources of the Internet and the power of the mobile agent to offer a simple and elegant programming model.

The mobile agents also require an architecture that provides for its mobility; this is provided by the aZIMAs architecture that uses HTTP-based infrastructure.

CHAPTER 1
INTRODUCTION

1.1 Mobile Agents

Mobile Agents are software modules that can migrate from machine to machine carrying out user-defined actions. Web Agents represent their user on the Internet. Agents perform a wide range of actions, from information access, being personal assistants – assisting users in daily computer-based tasks to semi-intelligent agents that adapt to the user behavior and environment so as to mimic the user and carry out tasks on behalf of the user.

Mobile Code is code that moves from host to host. It is typically written in an interpreted language to overcome the problem of heterogeneity. Mobile Code is a paradigm very different from Client Server computing. There is no active connection and the code physically moves over to the data and works on it. It is also different from process migration in that, mobile code used by agents is either user-controlled or autonomous – taking decisions to do their user's bidding.

Mobile code has been applied in many traditional computer science fields such as mobile computing, telecommunication applications and electronic commerce.

1.2 Mobile Agents and the Internet

With the current trends of rapidly evolving computer technology and the exponential growth of information and services on the Internet, we are fast approaching the scenario where everybody has a desktop at home, work or school and also be able to

access information from practically everywhere, televisions, pagers, cell phones and phones to name just a few. Mobile agent will be the essential tool to meet this need.

1.2.1 Compelling Reasons for Mobile Agents

Network technology has taken us very far and astonishing speeds and bandwidths can be obtained for a price. Backbone networks in businesses as well as the Internet are being upgraded to be able to carry tremendous amounts of information. But the end user still does not have such speed within his reach. Mobile Agents are relatively small programs that can traverse the Internet for the user and return with the required result. It can maximize the speed of the Internet and not burden the user by requiring an open connection [Kot99].

The popularity of mobile devices and web-based email services make it clear that users value the ability to access information and carry on with their duties regardless of their geographical location. Mobile code makes it possible to send the agents, disconnect, travel and later expect to find results on re-connecting.

Vendors on the web are already providing services that involve interacting with multiple other similar services and providing you with the best deal. Agents dispatched with a specific request can visit multiple sites providing the service and “choose” the best deal as per the user’s requirements.

1.2.2 Why Are Mobile Agents Not Pervasive?

Mobile Agents save network latency and bandwidth at the expense of higher loads on the server machines. Agents typically written in an interpreted language for portability needs to have an execution environment at the destination and must be activated. Hence in the absence of any disconnections, a typical agent might take longer than the traditional method of accomplishing the task.

The requirement of an execution environment at the destination implies that mobile agents cannot freely move within the Internet. They are constrained within the environment that support a compatible execution environment.

Many mobile agent systems have been proposed, each with their own merits and de-merits causing a standardization issue. Object Management Group (OMG)'s MASIF is an effort at standardizing mobile agent system architectures.

Agents have a great potential and if harnessed properly can be used to achieve user tasks with simplicity. But, the mechanism to accomplish that is yet to "arrive". Agent languages proposed still involve a significant amount of learning. An application that can effectively demonstrate the significant advantage of using agents, which is a simple tool that can be used without a great amount of learning and useful enough to immediately gain the interest of users, is still in it's infancy.

1.3 Goal of Thesis

This thesis aims at providing a simple but powerful programming model for agents. The model enables use of agents in activities that we perform everyday. It combines email and the Internet to provide simple solutions using agents. There has been an abundance of research on intelligent agents, autonomous agents and inter-agent interactions. This thesis does not focus on those topics. This thesis is about enabling agents like any method call and using them in applications that require user collaboration.

1.4 Organization of Thesis

This chapter was an introduction to the thesis, describing the motivation for using mobile agents and the goal of this thesis. Chapter 2 is a survey of the various agent architectures that have been developed, with regard to their agent development

environment and agent applications, specifically agent applications that require collaboration. Chapter 3 describes the agent system architecture (aZIMAs) that has been used for this thesis. Chapter 4 describes the Web Agent Programming Model Developed. Chapter 5 describes the application agents developed using the Programming Model. Chapter 6 presents the conclusions of this thesis.

CHAPTER 2
SURVEY OF MOBILE AGENT COMMUNICATION TECHNIQUES AND APPLICATIONS

Mobile Agents are programs performing tasks for their user, on the network. They are a representation of the user, carrying along with them the user profile and preferences, accomplishing tasks that require them to move from one location to another. The essence of distributed processing is this "mobility". Mobility is provided by the Mobile Agent Systems that provide the necessary infrastructure and is based on standard protocols to enable sending and receiving of agents.

The mobile agent system has three basic parts; the client end, agent server and the agent transfer protocol. Using Ajanta [Aja99] as a model, an overview of the mobile agent system is given. The agent system needs to provide a mechanism for transport of agents, activation, security, resource access and communication among agents.

2.1 Ajanta–A Mobile Agent System

Ajanta is a java-based mobile agent system. Agents are implemented in java and are sent and received using the agent transfer protocol [Tri98a].

A set of agent servers forms the environment for the mobile agents. An application can dispatch agents to agent servers and receive agents if they implement the agent transfer protocol. The agent itself consists of the agent state and agent code. Client creates an agent by sub classing the Agent class provided. An agent created at the client end is, in execution terms, inactive until it reaches the agent server and is activated. The agent server runs the agent code in a thread hence activating it.

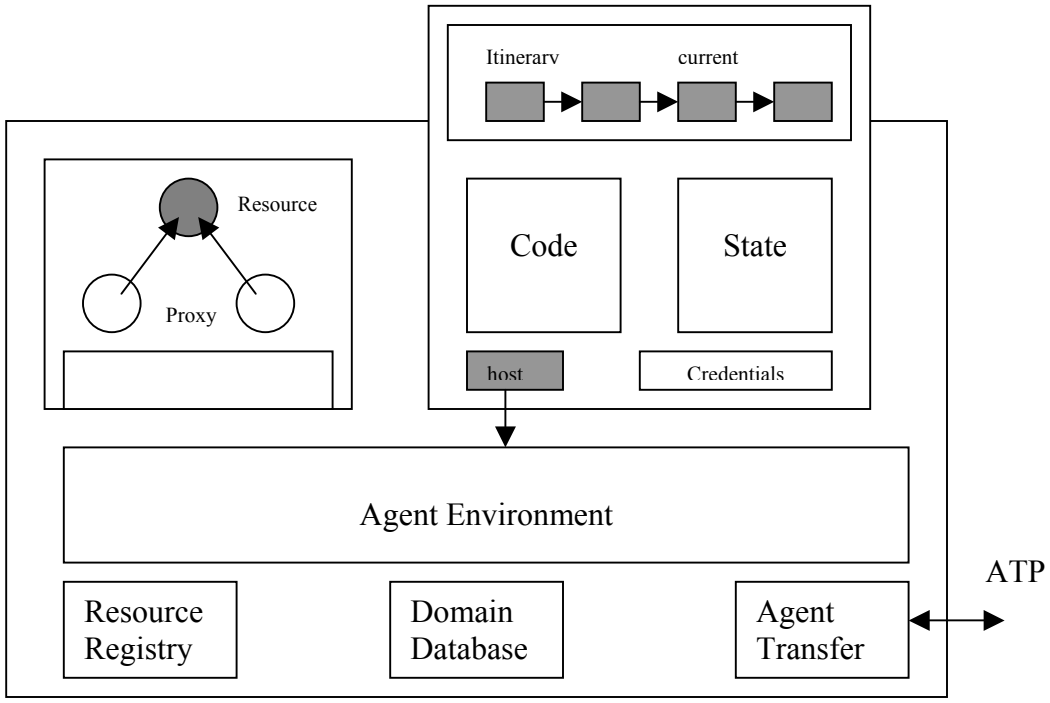

Figure 2.1 Ajanta Server Architecture

Object mobility is provided in Java by making use of Java's serializing facility. The client sends the serialized agent object and the server de-serializes it obtaining the state and activates the agent. The granularity of un-interrupted execution is however only method level. Acknowledging the restrictions of the Java Virtual Machine and the need to keep the code portable, the execution state of the agent is not sent. The agent stops execution at a point and can request that it be transferred to another location and activated by calling a specific method, say methodX().

Once the agent reaches the server, the agent "credentials" are verified and then permission is granted for the agent to be activated. At the server the agent can access resources using the resource access protocol and a resource registry is maintained by the server to keep track of the bindings between them. The server's domain database keeps

track of the agents currently executing in the server. The agent environment provides the interface between agents at the server and the agent server itself.

Agent communication is provided between agents on the same server by proxy-based resource access. Clients do not talk directly with the resource, a proxy is interposed between them, the proxy may be a proxy to an agent and hence communication is possible between agents on the same server. Alternatively the abstract nature of the resource access protocol can be used, considering the agent as a resource, another agent may communicate with it.

For agents on different servers, communication takes place via the RMIproxy interface provided. An agent wishing to allow other agents to communicate remotely with itself uses the createRMIProxy primitive to request it's server to install an RMI proxy for a specified interface. This however is a potential security loophole, so creation of such proxies is restricted to trusted code bases. The Ajanta architecture thus provides the mechanisms for agent creation, transfer, security, resource access and agent-to-agent communication.

2.2 Mobile Agent Systems in Existence Today

Mobile agents are an area of active research and several mobile agent systems [Tri98b] have been developed by academic as well as industrial research groups [Res97].

2.2.1 TeleScript

Developed by General Magic, TeleScript [Whi95] is a mobile agent system that provides servers at "places" that offer services. It also includes an object-oriented type-safe language for agent programming. Agents migrate to places specified by DNS-based host names and use the "go" primitive to do so.

Agent execution state is captured at thread level, so the agent resumes execution immediately after “go”. TeleScript also provide for agent communication between co-located agents and has an event-signaling facility.

2.2.2 Odyssey

Also developed by General Magic. This project was taken on after TeleScript proved unsuccessful, primarily because the learning curve was steep since programmers had to learn a new language for agents [Tri98b].

Odyssey is a java based agent system, which uses the same design framework as TeleScript. However as with most other Java based systems, it lacks thread-level capture of state.

2.2.3 Tacoma

A joint project [Tac95] by the University of Tromso (Norway) and Cornell University, it uses Tcl as the Agent Language. Technically they can also hold scripts written in other languages. Information is stored in “folders” and the agent is packed by aggregating these folders. Agent state is stored in one folder, the host is specified in another called HOST and application specific information is stored in other folders. The HOST must be an agent capable of accepting the incoming code and executing the associated, say Tcl, script. Thread-level state is not captures so the agents are restarted at the destination.

Agents are sent using the “meet” primitive and agent communication is also enabled by it. The “meet” primitive is used to co-locate agents and communicate by exchanging information in their folders. Agents can also share information by using “cabinets,” immobile repositories of data.

2.2.4 Agent Tcl

Developed by Dartmouth College, allows Tcl [Gra96] scripts to migrate between servers. These servers must support the necessary functions of send, receive, execute agents and provide for security.

A Tcl interpreter is used at the server to execute the Tcl scripts and to capture thread-level information. It uses Safe Tcl execution environment to provide restricted access to resources.

2.2.5 Aglets

Java-based system developed by IBM [Ibm98]. The programmer implements the agent class by inheriting default implementations from the Aglet class and overriding them with application specific code.

Mobility is achieved by using Java's object serialization. Message passing is the only means of agent-to-agent communication [Kar97].

2.2.6 Voyager

Java-based system developed by ObjectSpace [Obj97a]. Agents are implemented using a mechanism called virtual class. This utility takes a java class and builds a remotely accessible equivalent for it. The virtual reference returned is used to provide location independent access to the instance.

Agents are assigned a UUID and an optional symbolic name. Name service may be used to locate an agent. The moveTo primitive of the virtual class makes migration of agents possible. The primitive allows the user to specify a target method that is executed upon reaching the desired location.

Agent communication is achieved by method invocation on virtual references [Obj97b].

2.2.7 Concordia

Concordia [Mit97], developed by Mitsubishi Electric, supports mobile agents written in Java. As most java agent systems, it provides mobility by object serialization and class loading mechanisms and does not capture thread level state.

It has extensive support for agent communication, provides for asynchronous even signaling as well as a group collaboration mechanism.

2.3 Specialized Mobile Agent Languages and Applications

Each of the above mentioned Agent Systems has, associated with them, a mechanism for agent communication and transfer of information between agents. There are other specialized techniques proposed, which focus on the agent communication aspect assuming the existence of a suitable infrastructure.

2.3.1 Knowledge Query Manipulation Language (KQML)

KQML is a language that consists of primitives that the agents understand and "speak" [Lab97]. KQML does not have an implementation in the sense that it does not have to be compiled or interpreted. It specifies the syntax and semantics of a language which the application programmer, to understand such a language, will write code for. Thus are born the KQML-speaking agents.

So, KQML provides a specification or a protocol which when agents conform to, given them to ability to speak KQML. In an environment where multiple agents speak KQML, agent communication is achieved.

2.3.2 Aladin–Agent Specification Language

Aladin [Ros96] is a prototypical language for designing Interrap agents. Interrap is a layered architecture consisting of three control layers: a behavior-based layer, a local planning layer, and a cooperative planning layer. The language constructs provided in

Aladin are aimed towards supporting the implementation of agent capabilities in the different layers of Interrap.

Aladin has been based on the Oz programming language which provides concurrency and object orientation. The architecture is geared towards supporting autonomous agents that support requirements such as reactivity, efficiency, adaptability, the ability to produce goal-directed behavior and to interact purposefully with other agents.

2.3.3 Safe-Tcl Language for Enabled Mail

Safe-Tcl an extension of Tcl, is proposed to enhance the utility of electronic mail and to permit delivery of active messages that interact with the user and take action depending on their response [Bor97].

Safe-Tcl is Tcl made MIME smart and restricted from some commands that is considered unsafe in the email environment. Commands that involved accessing files or executing system commands were restricted.

Tcl is extended with ability to interact with the user and mime types and interface styles are provided. It is also made multimedia capable.

Hence Tcl is made “safer” and extended for interaction with user to provide Safe-Tcl that changes email to a more potent form that can take user inputs and perform various actions based on it.

2.3.4 Collaborative Interface Agents

Interface agents are semi-intelligent agents that perform computer-based tasks for users. These agents “learn” by watching the tasks performed by the user and observing patterns and behaviors. The problem with such agents is the steep learning curve. For a

newly created agent to learn about the user and perform useful tasks takes a long time [Las97].

Hence, the idea of agents collaborating among themselves becomes an attractive proposal. Now agents that do not have the required knowledge can seek to obtain this knowledge from another agent. This framework has three basic components: registering, locating peers and collaboration.

Agents wishing to assist other agents must register themselves with a "Bulletin Board," whose existence and location, is known to all agents. While registering, agents also provide information regarding their domain and their ontology. Each registering agent is given a unique identifier by the Bulletin Board.

Agents wishing to locate suitable peers may contact this Bulleting Board. Queries may take many forms depending on the information required. Namely, queries may be provided to locate a domain in a specific domain or maybe for an agent than can help with a particular ontology.

Agent collaboration is achieved by a series of request and reply messages. A request contains the requestor's identity, the request with the ontology indicated and a "request id". The request id helps in distinguishing between multiple requests. The types of requests and format of replies are pre-defined by the framework.

2.3.5 Collagen

Collagen [Ric97] is a toolkit that embodies a set of conventions for collaborative discourse, built with the inherent idea that the collaboration is with people. It does not specify how the agent works regarding its decision-making abilities; instead it concentrates on the mechanism to interact with the user.

From the user's point of view a Collagen agent presents itself in three ways. First, the user sees direct communication with the agent. This takes place in the form and questions, answers, proposals and is presented by popping open a window or by writing to a pre-defined message area.

Second, the user can communicate to the agent through the user communication menu. The communication menu dynamically changes to reflect the discourse between agent and user.

Third, the user can request a printout of parts of the agents internal discourse state in the form of a segmented interaction history. They are like log files, but hierarchical including not only the primitive actions and also the intentions and high-level goals of the user and agent.

From the agent's point of view, each entry in the user communication menu is generated from an artificial language representation. So, even though the menu is in a natural language, no extra effort has to be spent in interpreting it. The agent also maintains a history list and a recipe tree. The agent tasks are stacked and the history list maintains a record of the top-level segments that have been popped off the stack. The recipe tree helps the agent in choosing the best suggestion when too many constraints have been added.

Collagen also provides software interfaces to the recipe library and discourse library.

2.4 Summary

Mobile Agent Systems have been around for a very long time. We have looked at some of them here. Each system has its own development environment and many

applications have been built over them. These also carry the burden of learning the architecture.

Agent applications exist in many domains. There have also been developed language specifications for agent communication, KQML, Aladin to name a few. Yet, mobile agents are still largely a research topic. They have heavy infrastructure requirements; dedicated servers need to be running at every machine that wants to receive or send mobile agents. Applications such as calendar manager or web search tools developed over Ajanta [Tri99] had servers running at every user machine.

Mobile agents offer us many advantages; they are the ideal accompaniment to the Internet that has vast number of resources, users or information. Mobile agents have the potential to be a very powerful tool. The heavy infrastructure and steep learning curve to develop agents are stumbling blocks for the widespread use of mobile agents.

CHAPTER 3
aZIMAs-AGENT SYSTEM ARCHITECTURE

The Internet is a well-established network of computers. It works on the concept of web servers serving web pages upon receiving HTTP requests. HTTP is the protocol used by web servers. aZIMAs [Nal01] achieves almost zero infrastructure by building the agent system over the established network of web servers that form the Internet.

aZIMAs is built over the Apache web server. Apache was chosen since it was open source and would be easily available for research. A module was added to Apache, which would be invoked once a mobile agent comes in to the web server. The agent is sent to the web server in the form of a MIME [Fre96] message embedded in the HTTP request. The HTTP request header is filled with specific values to specify that it is an agent application. The web server is configured to forward requests with the application/agent attribute to the newly added module.

This agent system architecture does not need a dedicated agent server running at every host that wants to receive and send agents. The web servers that are already configured on various machines throughout the world and which are providing a needed service may be modified to send and receive agents [Lin95].

This module will not affect the functionalities of a web server, it only enhances it to make is agent-friendly. The module can receive agents, set up an agent execution environment and send agents to other web servers.

3.1 Client Environment

aZIMAs receives agents by parsing the MIME message in the HTTP request. The message is a multi-part MIME message, which includes the agent state, agent attributes and the various classes required for agent execution.

Table 3.1 Multi-Part MIME (Containing the Agent) Embedded in a HTTP Request

```
POST /agents HTTP/1.1
From: analla@cise.ufl.edu
Content-Type: multipart/mixed; boundary=abxyzf

--abxyzf
Content-Type: application/agent-attributes

Agent Name: aZIMAs agent
Source: analla@cise.ufl.edu
Last Host: web.cise.ufl.edu
[Other Agent attributes]
--abxyzf
Content-Disposition: attachment; filename="Agent.class"
Content-Type: application/agent-code

[Class File follows]
--abxyzf
Content-Disposition: attachment; filename="agent_obj"
Content-Type: application/agent-state

[Agent State Follows]
--abxyzf--
```

At the client end, the agent is sent using the *go* API provided by the architecture. The API takes the agent state in serialized form, agent attributes and related class files; constructs a multi-part MIME message from it. This message is then sent via a HTTP request to the first web server. The name of the web server is specified either as a configuration parameter or as part of the agent logic. The HTTP header will indicate the application type of the request, namely application/agent.

The agent must extend from `AzimasAgent`, a class provided by the architecture, to be able to use the *go* method and jump to a web server.

3.2 Server Environment

The Apache server receives the HTTP request and the application type in the header, application/agent, ensures that the aZIMAs module is invoked for this request. At the web server, there exists a directory space for mobile agents.

The aZIMAs module creates a sub-directory for this newly received agent. The MIME message is extracted from the HTTP request and parsed to extract agent attributes, agent state and the class information. The classes are stored under their respective class file names in the sub-directory created. The sub-directory has the same name as the agent name, specified among the agent attributes.

The module then starts a Java server that takes the agent state, de-serializes it and runs it as a thread. The agent must have extended from the `AzimasAgent` class and implemented the *run* method to be started by the Java server. This Java server will be active only for as long as there are agents running on the web server. The server will automatically shutdown when there are no more agents.

The aZIMAs module starts a Java server only if one is not already running. A single Java server can handle multiple agents.

The aZIMAs module can receive agents from clients, other web servers as well as partially executed agents returning after user interaction environments. The two are differentiated by the fact that an agent arriving from the client or another web server will have all its class files arriving along with it. The agent arriving after getting user input, is returning to its code base, hence will have only agent state. The partially executed agent

will be de-serialized and run in its environment which was already established when it arrived at this web server for the first time.

3.3 APIs Provided for Web Agent Developer

- *lauch* API

  Agents use this API to move from client to the first web server. The agent will indicate the classes it requires and the API will transport the agent as well as the classes to the destination web server.

- *go* API

  The *go* API gives mobility to the agents. It can be used by agents to move between web servers and to return from a user's web browser to the code base server. There are three implementations and the version used depends on the environment of the agent.

- *getCodeSpace* API

  This API returns the path to the directory that contains the classes related to this agent. The agent may use this space if it needs to write to files.

- *done* API

  The agent to indicate that it has completed its work should invoke this API. The aZIMAs module will remove the class files once this method is invoked or when the agent calls the *go* API to move to another web server.

The mobile agents developed for this architecture must extend `AzimasAgent`; a class provided by the aZIMAs architecture. The class provides the implementation for the APIs mentioned above.

CHAPTER 4
THE WEB AGENT PROGRAMMING MODEL

The aZIMAs infrastructure provides the means to send agents to the web server, run the agents at the web server and move the agents between web servers. It provides an agent system architecture. Using this system, users can develop agents to use the HTTP as their means of transport.

The Web Agent Programming Model (WAPM) enables development and deployment of Web Agents over the aZIMAs architecture. It provides a web agent development environment. It provides guidelines for agent developers on how to put the agent architecture to work.

It also enhances the architecture by providing agent developers with a means to express in a more intuitive way, what they want to do with their agents. Many agents have been developed and if we could harness the functionality of these, to do what we desire rather than re-invent the agent every time; that will provide very effective code reuse.

The WAPM addresses two issues associated with any agent architecture. One, it provides a development environment specifically geared towards building agents that need to interact with multiple users. Two, given any architecture, a means to treat agents at an abstract level and co-ordinate actions of the agents as per the logic of the specific algorithm for that application.

The agents built using WAPM can actually reside in the user's web browser interacting with the user, providing the information, getting feedback and then move on,

to the web server to continue executing its application logic. The agent has the ability to freeze its state, using Java's Serializability technique, and stay dormant until the intended user picks it up via the applet sent by WAPM; as well as to return to the code base and re-start execution of the application logic.

WAPM provides an agent specification language and an infrastructure for web agent development.

The agent specification language is a high level language that treats agent invocations like function calls. It is a scripting language that provides a restricted but powerful set of language features to direct agent activities and relate the multiple agents. The agent instructions are first run through a pre-processor. The pre-processor checks for syntax errors and provides strong type checking.

Web Agents are agents that use the Internet as their roaming ground and jump from browser to browser carrying out activities for the user. Primary focus is on agents that interact with users to obtain information and process that information to retrieve meaningful results for the owner. Applets have been chosen as the means of communicating with users on their browsers.

Applets activate the Internet. Most popular browsers are already Java-Enabled. Users are familiar with the process of downloading applets off the web and running them. Web Servers already have a way of handling applets and security related issues could be met. Hence applets provide an established mechanism to interact with users.

Agent Manager is the interpreter for the Agent Specification Language. A script written in the agent specification language is input to a pre-processor. The pre-processor ensures correct syntax and performs type checking.

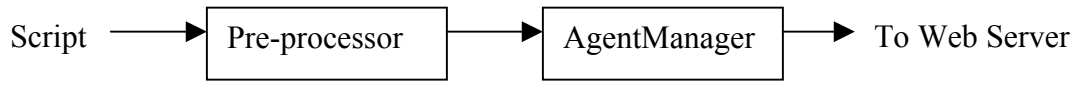

Figure 4.1 WAPM–Client Side

Various agents written for the WAPM are pre-registered with the Agent Manager. The pre-processor gets the information and performs type checking for the agents invoked from the script. The pre-processed script is then given to the Agent Manager.

Agent Manager is the module that interprets the script written in the agent specification language. It executes the body of the script, which might require jumping to a web server or to a user. The agent manager maintains state between agent calls so that it can resume execution once the agent completes its job.

The Agent Manager reads from configuration at the client side and uses the architecture's *go* method to move itself to the first web server. The go method as described earlier, composes a multi-part MIME message consisting of the agent-attributes, agent state and the various class files required by the agent. This MIME message is then sent as a HTTP POST request. At the web server, the application type is read from the HTTP header; an application/agent type of agent is handed over to the aZIMAs module of the Apache web server. This module stores the class files and transfers the agent state to the Java Run-time support environment, which is a Java server running on the web server. The Java Server then starts the agent on a separate thread.

Agents migrate from one machine to another by using the aZIMAs infrastructure's *go* method or using applet communication provided by a combination of *sendEmail* and the WAPM.

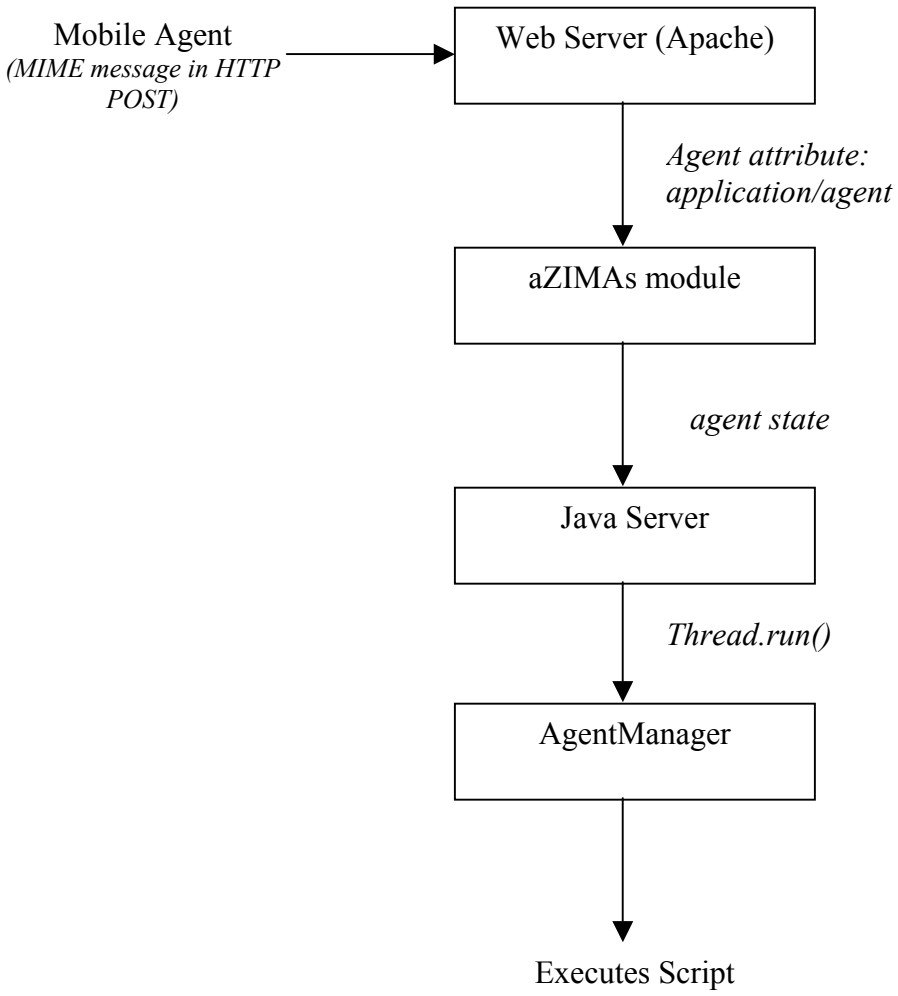

Figure 4.2 WAPM–Server Side

WAPM provides applet-agent technology for agent developers to develop agents that need to interact with users. Agents send the link to the applet using the *sendEmail* API in the architecture. Java's serialization mechanism is used to serialize the agent and store it in a file. The applet will contain a parameter that indicates the name of the file containing the serialized agent.

When the user clicks on the link, the applet is downloaded from the web server. The applet is a class provided by WAPM, it is called `DialogApplet`.

Table 4.1 DialogApplet.html

```
<HTML>
<TITLE> Dialog Applet </TITLE>
<BODY>
<APPLET CLASS=DialogApplet WIDTH=600 HEIGHT=600>
<PARAM NAME=”agent” VALUE=”agent.ser”>
</APPLET>
</BODY>
</HTML>
```

As soon as the web browser downloads the applet, the *init* method is invoked. `DialogApplet` will then read the parameter, get the serialized agent from the web server, de-serialize it and call *doInit* on the agent. The agent's *doInit* method has the look-and-feel of the applet coded by the agent developer.

The applet has now been rendered on the web browser. The user actions are processed and once the agent has completed its work, it goes back to the web-server by using the architecture API's *go* method.

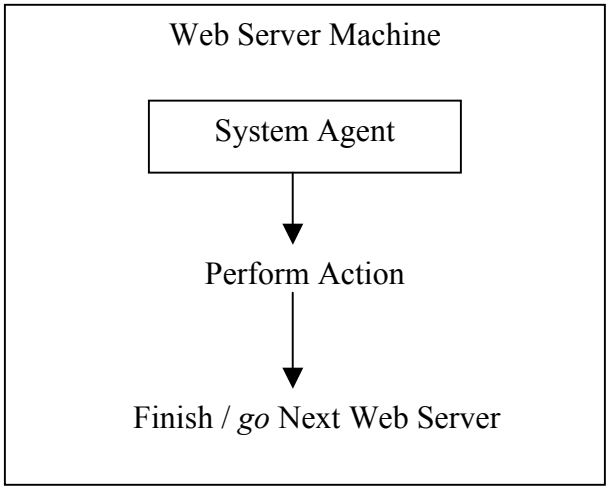

Figure 4.3 WAPM–System Agent

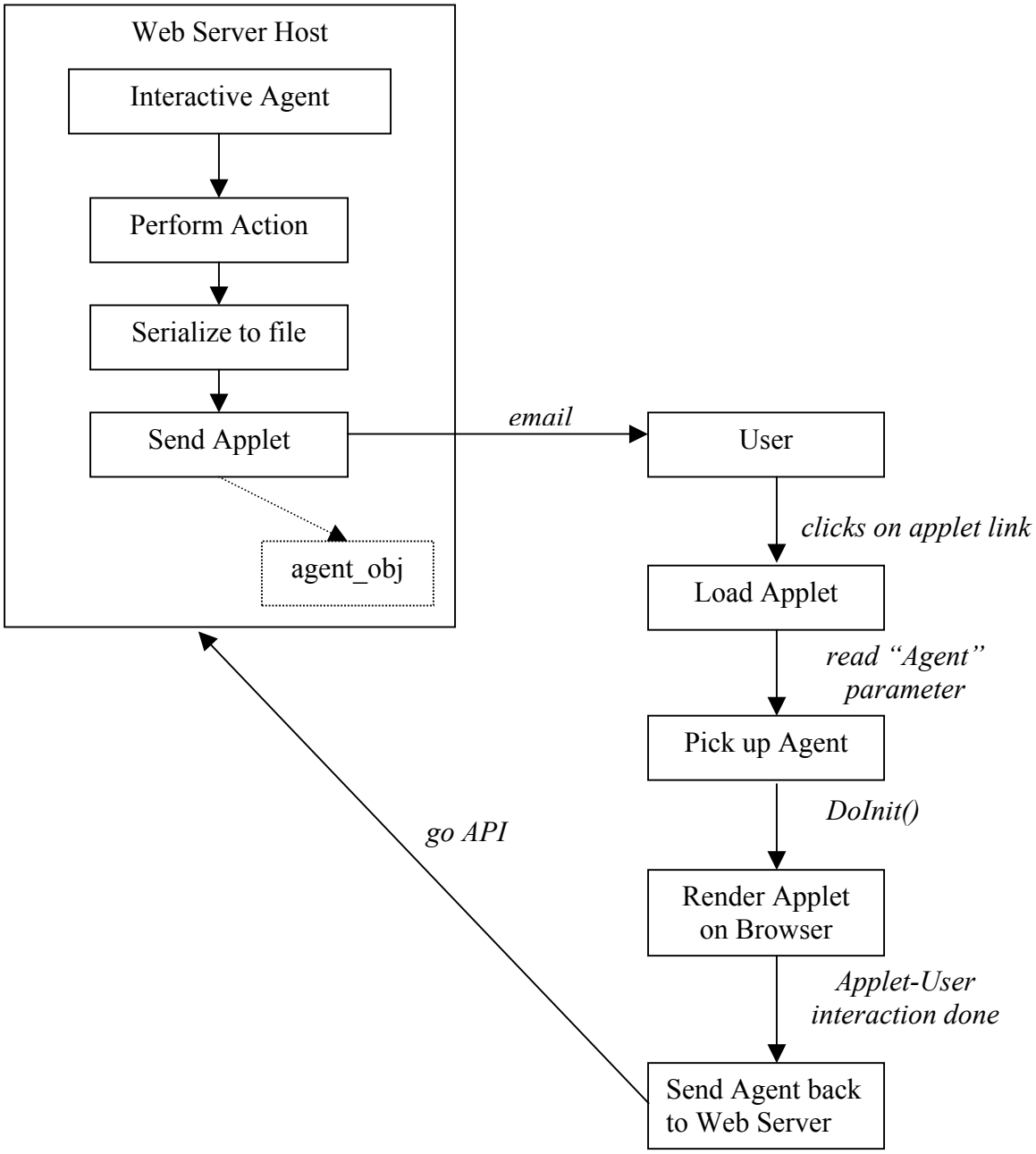

Figure 4.4 WAPM–Interactive Agent

aZIMAs receives the agent and starts it off on a separate thread. The currently executing agent still has control and can now decide if it needs to contact other users or if has obtained the results. Contacting other users follow the same process as outlined

above. Once the agent has collected all the results, it returns control to the agent manager by activating the agent manager's thread of execution.

4.1 Agent Specification Language

The agent specification language describes the language features that can be understood by the agent manager. It supports a limited set of data-types and language constructs to effectively relate activities of multiple agents.

The script can have two sections DECLARE and BODY sections. The DECLARE section specifies the name value mappings to be used in the script. The data types supported are String, array of Strings and integer.

Language constructs supported are if-then-else and assign statements. The if-then-else construct can have an agent invocation as part of its condition and the result of the agent action can be compared to a value to produce a boolean valued expression that evaluates to true or false. The then-stmt and else-stmt parts can have agent actions or assign statements. The assign statements can assign the return value of an agent action or the results of an agent action to a name and use the name elsewhere.

Every agent to be used in this script must register with the agent manager. At the time of registration, it will specify the RESULTS that it produces. The pre-processor will type-check agent invocation using this information.

The script written in the agent specification language is input to the agent manager, as a command line argument, once it has been cleared by the pre-processor. All syntax and type correctness issues are solved at the client end and the agent manager gets a syntactically correct, type safe set of instructions to be executed.

4.2 Agent Manager

Agent Manager extends the `AzimasAgent` class provided by the architecture. Hence agent manager is also an agent conforming to the aZIMAs architecture. It has a *run* method to be able to be started in a thread. It can also use the various API's provided as part of the `AzimasAgent` class.

The agent manager is also an interpreter for the script written in the agent specification language. It saves state when an agent is to be invoked and stays dormant until the agent has completed its job. Every agent is aware of its agent manager and once the agent has done its action, it will re-start the agent manager. The agent manager will then restore state and resume execution of the script.

The agent manager is also a serializable class. It implements Java's Serializable interface and has its own *readObject* and *writeObject* methods to save state and restore state during execution of the script.

Upon startup the agent manager will automatically discover the agents in the current directory that are written for this architecture. It will then invoke static methods *initialize* to be provided in every conforming agent to get agent information such as agent name and agent results.

APIs Provided By Agent Manager:

- *initialize* – This method should be implemented by the agent developer and is called by the `AgentManager` to get information about the various application agents available and the actions they provide.

- *registerApplicationAgent* – The agent developer in his initialize method, to register the agent with the agent manager, should call this method. It takes as

arguments the application name, namely the function that this agent can provide users.

- *registerResults* – The agent developer in his initialize method, to register the results of agent action with the agent manager, should call this method. It takes as arguments the name of the result and the result type, namely String String array or integer.

- *setResult* - The agent developer in his *doAction* method must use the *setResult* method, upon completion of agent action, to give the result values to the agent manager.

- *setReturnValue* – The agent developer in his *doAction* method must use the *setReturnValue*, upon completion of agent action, to give the return value to the agent manager. The return value must be an integer and the its meaning must be clearly documented by the agent developer.

4.3 Developing a Mobile Agent

An agent in the WAPM is an agent that attains its mobility from the aZIMAs architecture and is an agent that can be used in the agent specification language to be linked in a pipeline of agent activities.

A set of classes is provided by the WAPM infrastructure for mobile agent developers. These include `ApplicationAgent, AgentManager, SystemAgent, InteractiveAgent, AgentScriptPreProcessor` and the `DialogApplet` class.

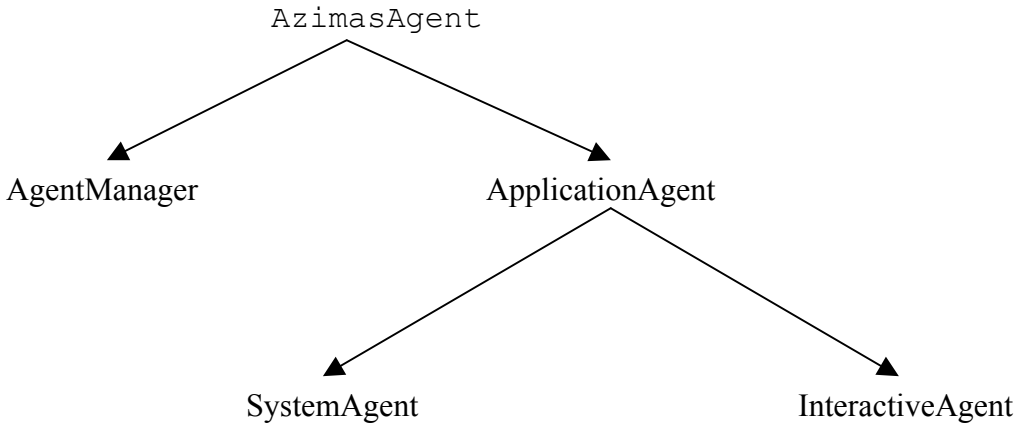

Figure 4.5 Class Hierarchy of the Model

Steps to be followed to develop a mobile agent:

Step 1: Extend the `ApplicationAgent` class. The `ApplicationAgent` class extends from the `AzimasAgent` class provided by the architecture. This class also holds a reference to the `AgentManager`. Every agent has an agent manager associated with it; the reference is set when the agent is instantiated.

Step 2: Implement the abstract methods in the `ApplicationAgent` class, namely

- *initialize()*

  This method is a static abstract method specified in the `ApplicationAgent` class. It is used by the `AgentManager` to discover and get information about the Application Agent developed. This method must invoke two methods on the associated agent manager:

- *registerApplicationAgent()*

This method registers the name of the application agent. Once registration is done, the user can use this application agent in the agent specification language.

- *registerResults()*

  This method informs the agent manager of the results produced by this application agent. Each result is given a name, the value for which will be set once the agent executes. The result name, as specified, can also be used in the agent specification language.

- *doAction()*

  This method is an abstract method in the `ApplicationAgent` class. It is invoked when the agent manager comes across the agent name in the agent script. This implements the application logic of the agent.

A single application agent may perform more than one action. The *doAction* method takes String name as a parameter. Depending on the name passed the appropriate logic may be executed. To provide for this functionality, *registerApplicationAgent*, which associates a name with the agent, may be called more than once in the initialize method.

Step 3: The infrastructure provides additional facilities for agents that intend to interact with users. Applets are the means of interacting with users. If the agent being developed is to interact with the user, then it should over-ride the *doInit* method of the `ApplicationAgent` class. The infrastructure applet, `DialogApplet`, invokes the agent's *doInit* method to render the applet on the web browser.

Table 4.2 Template for an Application Agent Developed in WAPM

```
class AnApplicationAgent extends ApplicationAgent
{
        public void initialize()
        {
                // The following two calls repeated as many times as the
                // number of functions this agent provides.
                getAgentManager().registerApplicationAgent("Function");
                getAgentManager().registerResults ({
                        "Result1", Result2",…,"Resultn"});
        }

        public void doAction (String name, Object[] params)
        {
                // application logic for the function passed in parameter 'name'
        }

        // Optional method: doInit
        // To be provided if the agent needs to interact with user on the
        // Web browser.

        public void doInit()
        {
              // Application logic to render applet on web browser and process
             // user input.
           }
   }
```

WAPM provides some agent primitives that may be useful for users in writing more effective scripts using the agent specification language. Operations that are very commonly used, such as: sending information via email to a list of people, confirming agent actions that obtained user input, canceling agent actions that required interaction with the user. These primitives may be accessed in the agent specification language by the names: SEND, CONFIRM and CANCEL. These are SystemAgents and extend from the `SystemAgent` class. They do not require the applet model, since they do not interact with users. System agents are typically used to send information or perform action at the web server.

CHAPTER 5
APPLICATION AGENTS DEVELOPED USING WAPM

Application Agents were built using the Web Agent Programming Model to demonstrate the features and to be able to present, visually, this new scheme of mobile agents and their interaction with users.

5.1 Query Application Agent

Frequently we come across scenarios where it is required to collect input from numerous users; namely surveys, feedback forms, scheduling meetings to name a few. To send out emails to multiple users and then track their replies, co-relate them and process the information to release meaningful data is a very involved process. To expedite this process, the Query agent can be programmed with the necessary logic to process user information.

The agent can jump from web browser to web browser, getting user input for users in a domain. It will jump from web server to web server to reach users in different domains. Identification of users is done using email addresses. The agent code is written using the model described in Chapter 4. The steps to develop a mobile agent using WAPM were followed and the application agent written conforms to the template.

The query application needs to interact with the user, hence the applet interaction model was used and a *doInit* method was implemented in the query application agent. The WAPM provides the facility to freeze the agent's state by serializing it and then re-starting it at some pre-defined point. This fits in with the requirement of the query

application agent to be able to stay quiet until the user picks it up and interacts with it and then, return to the web server to find out the next user and continue collecting input.

The query application agent needs inputs of the form: from address, to addresses, subject and body (actual query). It can be invoked in the script using the application agent name registered with the agent manager; ASK. This information is specified to the agent manager in the *initialize* method as per the model specification. The syntax and data types are enforced when the AgentScriptPreProcessor runs through it.

Table 5.1 Usage of the Query Application Agent in the Script

```
DECLARE
FROM = vidyare@ufl.edu
LIST = { x@ufl.edu , y@ufl.edu , z@ufl.edu }
SUB = Survey for the Lab
QUERY = " Would you like card access to the lab? "
END DECLARE
BODY
if ASK (  FROM , LIST , SUB , QUERY ) > 1
then
        CONFIRM (  FROM , LIST , SUB , QUERY )
end_then
END BODY
```

This agent can be used to conduct surveys, implement a distributed voting system and many other application. This simple API which provides a mechanism to receive yes or no responses from multiple users, can be used as a building block to develop more involved applications.

5.2 Filter Application Agent

The Internet is a treasure house of resources, resources in the form of information and in the form of the multitude of users who log-on everyday to browse or check email.

We could tap these resources to obtain useful information; use a specialist in a field to filter out a huge list of information to something more appropriate for your needs.

An application agent that could take a set of input and interact with the user to filter it and reduce it to a more meaningful set was developed. This agent, since it needs to provide a user interface, also uses the applet-enabled version; hence it extends from the `InteractiveAgent` class.

The agent is registered with the name FILTER; it takes a list of information, sends it to users and returns the filtered results. This agent uses the DialogApplet to render the applet on the web browser. The AgentManager, activates the agent by invoking the *doAction* method. This method does some processing to initialize the state of the agent, and then invokes the *sendApplet* method, provided in the `InteractiveAgent`, to send email containing the applet link to the user indicated in the script.

The *sendApplet* method as described earlier, will serialize the agent state into a file, indicate the name of the file as a PARAM in the applet html file and send the email. The serialized agent is now in an inactive state, waiting to be loaded by the applet, once the user clicks on the link.

The agent once loaded by the applet, sits on the web browser of the user, renders the interface via its *doInit* method and gets user input.

The Filter application agent is to be used as a building block. It makes possible application such as scheduling meetings; where a user may be given a number of choices and he chooses the ones convenient for him, applications to specialize search results by sending it to a live person and many others that involve a selection process.

CHAPTER 6
CONCLUSIONS

The Web Agent Programming Model (WAPM) was developed and deployed over the aZIMAs architecture. The programming model provides a scripting language that is used to co-ordinate multiple agent activities. A pre-processor was provided, which runs over the script identifying syntax error and ensuring conformance to the format required by the AgentManager.

The AgentManager takes the script as input and co-ordinates the agents as per the instructions. The AgentManager acts as a parser in understanding the script and as a manager in activating the agents.

Classes are provided that form the agent development environment. There are two kinds of agents that can be developed, first – System Agents that do not need to interact with users. These agents use the aZIMAs architecture to move between web servers and perform their actions. Second – Interactive Agents; this thesis focuses on these agents. Interactive agents collaborate with users by moving to the user's web browser. This is made possible by a dynamic combination of Applets and Java's Serializable Interface.

Various application agents were developed to show the viability of the model and to demonstrate the simplicity of development. This model makes it possible to develop agents that interact with users to provide useful information and implement day-to-day functions that would otherwise be much more tedious.

REFERENCES

[Aja99] University of Minnesota, Ajanta – Mobile Agents Research Project, http://www.cs.umn.edu/Ajanta (04/2001) (April 1998).

[Bor97] Borenstein, Nathaniel S., "Email with a Mind of Its Own: The Safe-Tcl Language for Enabled Mail," Huhns, Michael N., and Singh, Muninder P., editors, *Readings in Agents,* Morgan Kaufmann Publications. (1997).

[Fre96] Freed, N., and Borenstein, N., "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types," Request for Comments: 2046, (November 1996).

[Gra96] Gray, Robert S., "Agent Tcl: A Flexible and Secure Mobile Agent System," In Proceedings of the 4th Annual Tcl/Tk Workshop (TCL'96) (July 1996).

[Ibm98] IBM, Aglets – Software Development Kit, http://www.trl.ibm.com/aglets, (04/2001) (1998).

[Kar97] Karjoth, G., Lange, D., and Oshima, M., "A Security Model for Aglets," *IEEE Internet Computing*, Vol. 1, Jul-Aug 1997, pp. 68-77 (1997).

[Kot99] Kotz, D., and Gray, Robert S., "Mobile Agents and the Future of the Internet," In *ACM Operating Systems Review,* Vol. 33, Number 3, August 1999, pp. 7-13 (Aug 1999).

[Lab97] Labrou, Y., and Finin, T., "Semantics and Conversations for an Agent Communication Language," Huhns, Michael N., and Singh, Muninder P., editors, *Readings in Agents*, Morgan Kaufmann Publications. (1997).

[Las97] Lashkari, Y., Metral, M., and Maes, P., "Collaborative Interface Agents," Huhns, Michael N., and Singh, Muninder P., editors, *Readings in Agents*, Morgan Kaufmann Publications. (1997).

[Lin95] Lingnau, A., Drobnik, O., and Domel, P., "An HTTP-based Infrastructure for Mobile Agents," Proceedings of the 4th International WWW Conference, Boston, MA Dec 11-15, 1995 (Dec 1995).

[Mit97] Mitsubishi Electric, "Concordia: An Infrastructure for Collaborating Mobile Agents," Proceedings of the 1st International Workshop on Mobile Agents (MA'97) Berlin, Germany (April 1997).

[Nal01] Nalla, A., "aZIMAs – almost Zero Infrastructure Mobile Agent System," Masters Thesis, Computer Science Department, University of Florida, http://www.harris.cise.ufl.edu/projects/azimas.htm (05/2001) (2001).

[Obj97a] ObjectSpace Inc., ObjectSpace, http://www.objectspace.com (04/2001) (1997).

[Obj97b] ObjectSpace, ObjectSpace Voyager Core Package Technical Overview, Technical Report, ObjectSpace, Inc. (Jul 1997).

[Res97] ResearchIndex (CiteSeer), The NECI Scientific Literature Digital Library, http://www.researchindex.com (05/2001) (1997).

[Ric97] Rich, C., and Sidner, Candance L., "COLLAGEN: When Agents Collaborate with People," Huhns, Michael N., and Singh, Muninder P., editors, *Readings in Agents*, Morgan Kaufmann Publications. (1997).

[Ros96] Rosinus, M., Muller, Jorg P., and Pischel, M., "An Agent Specification Language," http://www.researchindex.com (04/2001) (1996).

[Tac95] University of Norway, Tacoma – Operating System Support for Agents, http://www.tacoma.cs.uit.no (04/2001) (1995).

[Tri98a] Tripathi, Anand R., and Karnik, Neeran M., "Agent Server Architecture for the Ajanta Mobile-Agent System," In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98) Austin, TX (1998).

[Tri98b] Tripathi, Anand R., and Karnik, Neeran M., "Design Issues in Mobile Agent Programming Systems," *IEEE Concurrency*, Vol. 6, Number 3, July-Sep 1998, pp. 52-61. (1998).

[Tri99] Tripathi, Anand R., Karnik, Neeran M., Vora, M., Ahmed, T., and Singh, R., "Mobile Agent Programming in Ajanta," In Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99) Austin, TX (1999).

[Whi95] White, James E., "Mobile Agents," Technical Report, General Magic Inc. (1995).

BIOGRAPHICAL SKETCH

Vidya Renganarayanan was born in Manganam, Kerala, India. She received her Bachelor of Engineering degree in Computer Science and Engineering from Anna University, Chennai, India, in September 1997. She worked at Hewlett-Packard, Bangalore, India, for two years as Software Engineer. She will receive her Master of Science degree in Computer Science from the University of Florida, Gainesville, FL, in December 2001. She has found employment with Brocade Communications Systems Inc., San Jose, CA.