THE USE OF MOBILE AGENTS IN NETWORK MANAGEMENT APPLICATIONS

Marcus Naylor

Submitted in partial fulfilment of the
requirements of Napier University
for the degree of

MSc Information Technology (Systems Integration)

School of Computing

January 2000

AUTHORSHIP DECLARATION

I, Marcus Naylor, confirm that this dissertation and the work presented in it are my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;
2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;
3. I have acknowledged all main sources of help;
4. If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;
5. I have read and understand the penalties associated with plagiarism.

Signed:

Date:

Matriculation no: 98007807

ABSTRACT

There is a trend toward increasingly heterogeneous networks in today's communicating environments. Managing these diverse networks requires the collection of large amounts of data from diverse and dispersed areas of the network. Simultaneously, from a user perspective there is a greater expectation in terms of reliability and service from the network. New management paradigms are being proposed as an alternative to the centralised, client/server architecture, and new technologies and programming languages make them feasible.

There has been an enormous amount of hype regarding the potential productivity gains from so called *intelligent* software agents, a technology that has implications for new network management applications. These however require complex artificial intelligence (AI) functionality. Where agents can be realistically of benefit is in those areas concerned with mobility. Agent mobility addresses some limitations faced by classic client/server architecture, namely, in minimising bandwidth consumption, in supporting adaptive network load balancing and in solving problems caused by intermittent or unreliable network connections. This report begins by discussing the usage of mobile agents and the advantages they may offer over client/server architectures. We discuss the main requirements expected from a mobile agent system and its implementation. Java has enjoyed wide spread acceptance as the programming language of choice for distributed applications. We demonstrate how it has the fundamental components that embrace the requirements from a mobile agent based system. Three mobile agent development toolkits are evaluated, Voyager from Object Space, JATLite and the Aglets Software Development Kit (ASDK) from IBM. The latter is chosen as our development tool.

A detailed requirements analysis identifies the task of information retrieval as key to many distributed management tasks. We then report the development of two example prototypes which encompass this generic task. Using first a client/server based approach and then a mobile agent based approach we plot the development lifecycle of both prototypes. Thus we formulate our evaluation of both system architectures and offer conclusions as to whether mobile based systems *do* offer significant advantages over client/server based systems.

TABLE OF CONTENTS

Authorship declaration 2
ABSTRACT 3
TABLE OF CONTENTS 4
LIST OF FIGURES 6
ACKNOWLEDGEMENTS 7
Section 1: INTRODUCTION 8
1.1 Project background 8
1.2 Project Objectives 9
Section 2: RESEARCH 10
2.1 Distributed Computing 10
2.2 Agent Technologies 10
2.2.1 What is an Agent? 10
2.2.2 Why use agents? 12
2.2.3 A Theoretical Agent Architecture 15
2.2.4 Agent Mobility 16
2.2.5 Agent Communication and Collaboration 17
2.3 Java 19
2.3.1 Sockets 20
2.3.2 Threads 21
2.3.3 Object Serialization 22
2.3.4 Security 22
2.4 An Evaluation of Agent Development tools 23
2.4.1 Introduction to the Aglet Object Model 24
2.4.1 Aglet System Architecture 25
Section 3: REQUIREMENTS ANALYSIS AND DESIGN 27
1.1 The Problem Domain – Distributed Systems 27
3.2 The Problem – Network Management Tasks Identified 28
3.3 Requirements Specification and Feasibility Study 29
3.4 Client/Server Design Requirements 30
3.4.1 Introduction 30
3.4.2 Protocols 31

3.4.3 Further Requirements 33
3.5 Mobile Agent Design Requirements 34
3.5.1. Introduction 34
3.5.2 Mobility Orientated Approach 34
Section 4: IMPLEMENTATION 36
4.1 The Info Client/Server Application 36
Implementation Details Summary 37
4.2 The gatherInfo Mobile Agent 38
4.2.1 Introduction to Programming Aglets 38
4.2.2 gatherInfo Agent Implementation 42
Implementation Details Summary 43
Section 5: EVALUATION 44
5.1 Mobile Agent versus Client/Server Architecture 44
5.2 Evaluation of Objectives 45
5.3 Recommendations for Further Work 46
5.4 Conclusions 47
APPENDIX 1: Project Plan 48
APPENDIX 2: Project Journal 49
APPENDIX 3: SMTP Protocol 56
4.1. SMTP Commands 56
4.2. SMTP REPLIES 57
APPENDIX 4: Extracts of code – e mail agent 59
APPENDIX 5: Extracts of code : Info Client/Server 62
APPENDIX 6: Extracts of Code : Gather Info Agent 68
APPENDIX 7: Aglets API Class Hierarchy 69
REFERENCES 71

LIST OF FIGURES

Figure 1: Client/Server architecture 12
Figure 2: Mobile agent architecture 13
Figure 3: Agent characteristics.. 15
Figure 4: Socket creation in Java 20
Figure 5: Process splitting into threads 21
Figure 6: Aglet object model... 24
Figure 7: Aglet system architecture 25
Figure 8: Info client/server.. 30
Figure 9: Info Client/Server transaction.. 32
Figure 10: Agent mobility 34
Figure 11: Tahiti the aglet GUI 40
Figure 12: Aglet information dialog 41

ACKNOWLEDGEMENTS

I would like to thank Sarah Cole for untangling my prose, and for her support throughout Also thanks to Bill Buchanan my Project Supervisor.

SECTION 1: INTRODUCTION

Software agents have been the focus of much attention. Agents display a number of both common and distinguishing characteristics that set them apart from software programs. One such characteristic, mobility, is likely to play an important role in future distributed systems. A mobile agent is not bound to the system where it begins execution but is free to travel among the hosts in the network. It has been claimed that this mobility represents an evolutionary step for distributed computing systems. This projects attempts to asses the significance of this claim by developing simple prototype applications, through design to implementation and testing, using both a 'traditional' client/server based approach and by utilising a mobile agent architecture.

This section serves to introduce the project, state its objectives and present a brief background that explains the impetus behind the work.

Section 2 introduces the research and emerging concepts behind agent, and more specifically mobile agent technologies. The requirements of an agent architecture are discussed and the suitability of Java as the enabling development language is highlighted. We then move on to evaluate three mobile agent development toolkits.

Having identified our problem statement; *Do mobile agent systems offer significant advantages over client/server systems?* Section 3 moves through the formulation of the requirements analysis to the design phase of the project. Our requirements are put into context by exploring potential tasks to assist the network manager in information and systems management in a distributed computing environment. With these needs in mind we are able to formulate an application suitable for the purposes of this project – the information retrieval agent.

Section 4 goes on to document the implementation stages of each prototype, pointing out specific issues relevant in each prototype implementation.

Finally, in Section 5 we conclude by offering a critical analysis of the work undertaken and specifically examine whether the advantages of adopting a mobile agent architecture over a client/server architecture are indeed persuasive. We offer some recommendations for further work and conclude.

1.1 Project background

The use of software agents as digital representatives acting on a users behalf is an attractive proposition. It is not hard to imagine a network inhabited by intelligent and collaborating agents attending to our more laborious electronic tasks. There has been an enormous amount of hype regarding the potential productivity gains from so called *intelligent* software agents. These however require complex artificial intelligence (AI) functionality and we are some way from realising this ideal. There still remains, however, a strong impetus to investigate the possible implications that may be realised by exploiting the characteristic of agent mobility. Indeed, it is this idea that gave rise to this project .

Together with mobility and autonomy there exits an enormous amount of potential application areas in which agents may be of assistance. Our agent need no longer be restricted to local execution environments, nor forced to communicate through structured but robust mechanisms, rather it has the ability to halt execution, travel to another host and resume its execution there. The whole network is at its disposal. This seems to represent a great leap forward in terms of distributed systems and information management. Migrating computation toward resources, having agents notify us of changes in the network environment and dispatching our agent to carry out some task offline all become easily obtainable goals.

The Systems Integration degree which preceded this work introduced many of the underlying technologies: distributed systems, data transport and communication protocols, systems administration and object-orientated design methods to name a few. Not only did this project integrate these areas of interest into a common research effort but it allowed new areas, most notably Java and mobile agent architectures to be explored.

1.2 Project Objectives

Having introduced the project and discussed the broad areas from which the research will draw, we may identify more specific project objectives:

1. To provide a review of mobile agent architectures and emerging agent technologies

2. To assess the suitability of the Java programming language as the development language for distributed applications and in particular for the development of mobile agent applications

3. To evaluate agent development tools available commercially and select one to adopt for the prototype application development

4. To discuss and identify potential network management tasks

5. To develop a prototype application using a traditional client/server approach and in so doing learn Java to a level of competence that will allow the development of the same application using a mobile agent orientated approach

6. To evaluate these two application design methods

7. Finally to document these findings and to identify areas for further research.

SECTION 2: RESEARCH

This section presents the findings of the research effort that were the precursor to the work that follows. The fundamentals of an agent, and in particular a mobile agent architecture are explored and the suitability of Java as the enabling programming tool to facilitate such an architecture is discussed. Finally an evaluation of three agent development toolkits, Voyager, JATLite and the Aglets Software Development Kit (ASDK) is made.

2.1 Distributed Computing

The rise of the networked workstation has been the most dramatic change in the last two decades of information technology. A shift from a shared centralised mainframe to the desktop PC put more processing power in the hands of the end-user. As networks of computing resources have become prevalent, the concept of distributing related processing among multiple resources has become increasingly important. We have seen an explosion in information availability on increasingly heterogeneous networks. Managing these diverse networks often requires the collection of large quantities of data from possibly dispersed parts of the network. This challenge provides a driving force to research the use of agents and in particular mobile agents to automate some network management tasks.

2.2 Agent Technologies

2.2.1 What is an Agent?

> *"Software agents automate tasks that otherwise we would have to do ourselves. What distinguishes software agents from other utility software programs is both the ability to perform in distributed computing environments and the ability to supply some domain knowledge to automating tasks for users." (Watson 1997)*

There has been an enormous amount of hype concerning the potential use of agents. The term *agent* has found its way into a number of technologies. It has been applied to aspects of artificial intelligence, as a means for improving collaborative online social environments and in a distributed system as an enhancement to client/server architecture. Agents can, it is claimed, sort your mail, act as electronic gophers, or automated errand boys, monitor the stock market and purchase shares, locate and purchase a cheap flight, notify a network administrator of a network fault (and even execute fault rectifying procedures) or monitor your website.

Despite this, it is a challenge to provide a succinct, comprehensive definition of an *agent.*

Many attempts at a comprehensive definition have been made by various sources; some of the more simple and brief definitions are stated as follows:

> *"An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors"* (Russel & Norvig 1995, p.33).
>
> *"Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realise a set of goals or tasks for which they are designed"* (Maes 1995, p.108).
>
> *"Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires."* (IBM, 1998)

These definitions, whilst informative, appear to result from specific examples of agents which the definer had in mind. In an attempt to clarify matters, Franklin & Graesser (1996) provide a thorough and well thought out classification, in which they state that agents possess several or all of the following characteristics:

- *reactive* — responds in a timely fashion to changes in the environment
- *autonomous* — exercises control over its own actions
- *goal-oriented* — does not simply act in response to the environment
- *temporally continuous* — is a continually running process
- *communicative* — communicates with other agents, perhaps including people
- *learning* — changes its behaviour based on previous experience
- *mobile* — able to transport itself from one machine to another
- *flexible* — actions are not scripted

Every agent satisfies the first four properties (reactive, autonomous, goal-oriented and temporally continuous). Other properties are defined on adding a hierarchical classification. Thus we may talk of mobile-learning agents, a subclass of mobile agents.

Clearly there is an enormous scope and agent research will draw on and integrate many diverse disciplines of computer science, including distributed object architectures, adaptive learning systems, artificial intelligence, expert systems and others.

2.2.2 Why use agents?

Having identified the key properties that any agent should posses let us now shift the focus from what an agent could do to what an agent *can* do and what problems they may solve. We are already using agents in modern software. Microsoft has developed *Office Assistants* to accompany their Office 97 and recently Office 2000 suite of programs. These satisfy the some of the definitions of an agent as previously described by attempting to monitor user actions and to offer help based on these actions. The spell checker can also be classed as an agent since it 'watches' the user as they type and makes corrections on the fly. Other common types of agent to which we have become accustomed are those associated with e-mail readers who permit various filters to sort or block incoming mail based on criteria that the user specifies.

These examples of agents draw heavily on AI for their functionality. However one key property which these agents do not possess is *mobility*, and as such they are unable to operate or take advantage of a distributed network environment. A mobile agent can travel to multiple nodes on the network, executing whichever tasks it deems appropriate in the dynamic context it discovers at each stop. A mobile agent has the whole network at its disposal. Of course web browsers, distributed object architectures (CORBA) and client server applications all utilise a distributed model. What makes a mobile agent architecture unique is the way in which network communication takes place.

A client/server application consists of the client and the server, usually on separate machines communicating over the network..

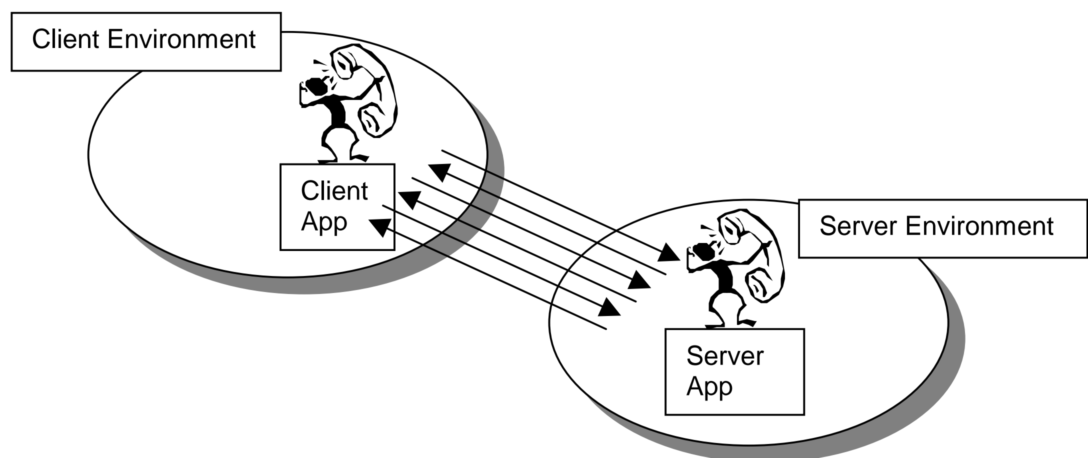


Figure 1: Client/Server architecture

When the client needs data or access to resources that the server provides, the client sends a request to the server (which must be on-line). The server in turn sends a response to the client. This handshake occurs over and over again, each request and response requiring a complete round trip across the network., see Figure 1. The

fundamental difference in mobile agent architecture is that instead of the client talking to the server over the network, the client actually migrates to the server's machine. Once on the server's machine, the client makes its requests of the server directly, see Figure 2.

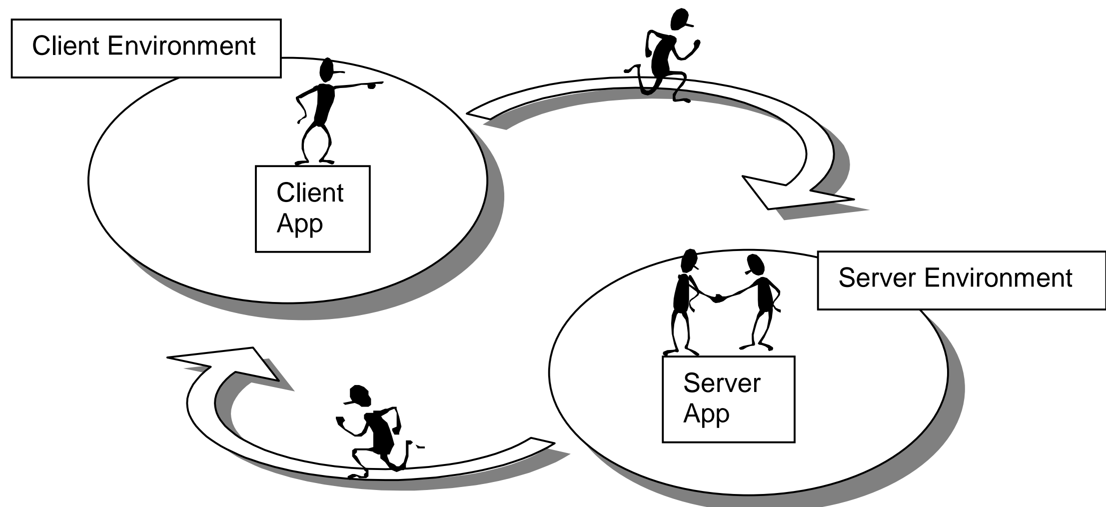


Figure 2: Mobile agent architecture

This represents a significant advantage over the client/server model and Sundsted (1998) states three reasons to adopt a mobile agent architecture:

1 Mobile agents solve the client/server network bandwidth problem. By moving a query or transaction from the client to the server, the repetitive request/response handshake is eliminated.

2 Agents allow decisions about the location of code (client vs. server) to be pushed toward the end of the development effort when more is known about how the application will perform, so reducing design risk.

3 Agent architectures also solve the problems created by intermittent or unreliable network connections. Agents that work off-line and communicate their results when the application is back on-line may be built quite easily

With regard to network management applications these three reasons go some way to solving several problems:

Network bandwidth is a valuable resource. Any client/server transaction will place demands on this often scarce resource. In any computationally intense activity it would clearly be an advantage to create an agent to handle the query, sending the agent from the client to the server. The agent could then carry out the task and then communicate the results back to the client upon completion (or indeed at any time deemed appropriate depending on network traffic), without the need for a complete transaction session. So instead of intermediate results and information being passed back and forth, only the agent need be sent.

In the design of traditional client/server applications, the programmer must have a clear idea about the roles of the client component and server component at design time. The intent and scope of the application must be clearly set out. Since the client and server communicate in a well-structured manner using a protocol, this offers little scope for modifications or enhancements. By contrast, in an agent architecture there is a greater degree of flexibility. An agent can be given an itinerary, which may be static or dynamic. By definition the agent has the ability to 'decide' upon its route based on factors in the environment it encounters. The only prerequisite is that at any node it visits, the agent has some environment in which it can be hosted. Agents can be easily created and their itineraries adapted as required. The means of communication, usually via message passing, should be clearly defined and the programmer need only concentrate on the tasks to which a particular agent is assigned.

An agent architecture may also solve the problems created by an intermittent or unreliable network. Typically a network connection must be alive and healthy over the entire time a transaction is taking place. If the network connection is lost the client must restart the transaction or query. An agent is able to carry out the transaction of its own accord. This could be extremely advantageous in an environment where frequent connections are made to the network with a laptop via a dial up connection. An agent could be dispatched to the server to perform some off-line calculation. Upon reconnection the agent would report back. This could perhaps be useful for some remote monitoring tasks.

There are other advantages to be gained including real time notification where an agent situated at some remote site may notify a local host of any important event immediately. Parallel execution (or load balancing) where a large computation can be divided amongst processing resources is another possible advantage. All these offer compelling reasons to adopt an agent architecture for network management tasks.

2.2.3 A Theoretical Agent Architecture

In an agent architecture we look for support for several agent characteristics (Figure 3). Sundsted (1998) identifies these as

1. the *tactile* characteristics of mobility and persistence
2. the *social* characteristics of communication and collaboration
3. the *cognitive* characteristics of adaptation, learning and goal orientation.

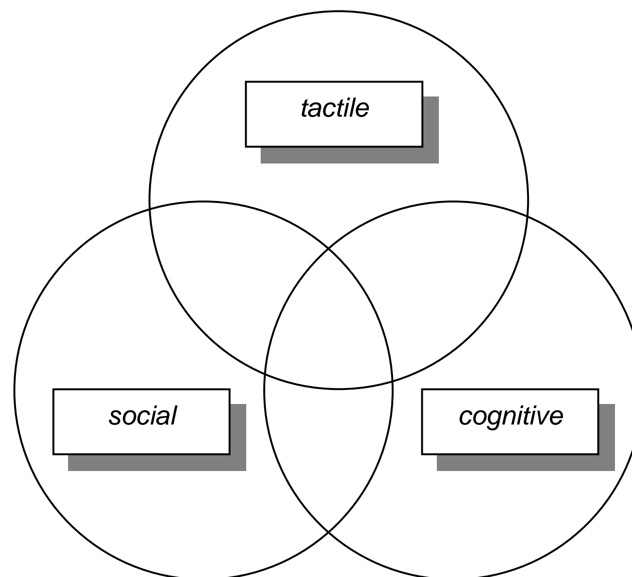


Figure 3: Agent characteristics

The cognitive characteristics, those of adaptation, learning and goal orientation, have enormous potential and are the focus of much of the hype that surrounds agents. It is an area that the Artificial Intelligence (AI) community has been addressing for a number of years – the problem of getting a computer to think for us. This is likely to remain an important issue to be solved for years to come. We will, however, concentrate on the first two characteristics of mobility and communication. These offer more realistic characteristics that we can exploit in order to address some of the limitations of traditional client/server applications in increasingly dispersed and heterogeneous network environments.

2.2.4 Agent Mobility

The idea of moving code and computation to the location of the data and the resources is not a new one. Java and applets revolutionised the World Wide Web (Niemeyer & Peck 1996). A web browser that supports Java can incorporate Java applets as executable content contained within documents, transforming web pages from static, flat hyper text documents to interactive applications.

Java enables us to write portable components that may be distributed over a network. (its suitability for such a purpose is discussed in detail in the following section). In particular, applets allow us to distribute executable content over the Internet along with data. Such mobile code is dynamically loaded and executed by standalone programs. For the applet this is the Web browser. Unlike the applet however, an agent takes with it its state of execution and an agent's characteristic of autonomy means that the agent can have responsibility for deciding where it can go and what it will do. Agents can receive requests from external sources, follow a predetermined itinerary or make decisions such as to travel across the network to a particular host, all independently of any external influences.

Security is a concern in any network. As connectivity increases so the avenues for potential attack increase. This concern is especially relevant when we consider mobile code as a potential source of attack. If an agent architecture ultimately allows agents free reign over the network, where the boundaries of hosts become blurred and access to local resources is available, security becomes a major concern. Such a situation would be a convenient way to spread viruses and a security mechanism must have in place facilities to handle both trusted code, where we can be sure of its source, and untrusted code, where we aren't certain of the source of a particular piece of code. Security mechanisms are of paramount importance to establish trust in network mobile code. Every browser implements security policies to keep applets from compromising system security. The following restrictions apply:

- An applet cannot read or write files on the executing host
- An applet cannot make network connections except to the host it came from
- An applet cannot read certain system properties
- An applet cannot start any program on the executing host
- An applet cannot load libraries or define native methods

Such draconian measures are very necessary if inhibiting. When considering a mobile agent architecture we have to address some of these concerns whilst hopefully allowing a greater deal of flexibility to enable a mobile agent to carry out its work. As we will see later, Java provides a highly customisable security model which allows us to go some way in achieving this aim. So, just as the applet requires a Web browser in which to execute, so an agent needs a safe environment in which to be hosted.

At a minimum, any agent operating within its host must have

- a unique identity
- a means of identifying which other agents are also operating within the host
- a means of determining what messages other agents accept and send

And the agent host must

- allow multiple agents to co-exist and execute simultaneously
- allow agents to communicate with one another and with the host
- provide a transport mechanism to transfer agents to another host and to accept agents from other hosts
- offer a way of 'freezing' an agent's state of execution prior to transfer and conversely 'thawing' an agent so as to allow its execution to continue after transfer
- inhibit agents from directly interfering with one another

As well as these requirements made upon the agent host which permit agent mobility, the agent must also be given a workplace. The agent's workplace is a window on the environment it lives, offering access only to those resources and data that are needed for the agent to fulfil its role. An agent's workplace is termed its *context*. The context is a stationary object residing on the host computer that provides a means for maintaining and running agents in a uniform execution environment, securing the host against possible malicious agent attacks. The context must prevent an untrusted agent from having access to sensitive data or resources, whilst ensuring that a trusted agent has useful access to those resources that it may need. Whilst it is fairly straightforward to build adequate security models to protect against malicious agents, it is not so easy to protect agents themselves against malicious hosts. If there exists no adequate security mechanism to prevent such attacks, a host could implant its own tasks into the agent or modify the agent's state. This could lead to the theft of the agent's resources, since if a host uploads an agent's class files and state of execution, it could potentially have access to such sensitive information. The agent context must then include some system whereby rules can be set stating what privileges an agent may be granted within its context. These will most likely be based on knowledge of the agent's origin.

2.2.5 Agent Communication and Collaboration

Agents interact with one another and applications through message passing. Message handling must be able to support both synchronous messages and asynchronous messages.

- A now-type message is synchronous and blocks the execution until the receiver has completed the handling of the message.

- A future-type message is asynchronous and does not block the execution. The sending agent can then either wait for a reply from the recipient or continue processing its task and get a reply later.

Another useful addition to agent communication is multicasting, whereby a message can be sent to all agents within a context simultaneously - only those agents who have subscribed to the message will receive it.

Thus a mobile agent architecture has the potential to support a powerful programming style. The challenge is to resist the temptation to revert to a traditional client/server approach and to look at opportunities in which agents may be given the capability of collaborating and relocating on a variety of hosts to perform the requisite work in a true distributed sense.

2.3 Java

Java, developed by Sun Microsystems, despite being relatively new to the computing industry, has rapidly gained a great deal of acceptance. Java was designed from the ground up as a complete execution environment, rather than just another programming language; therefore it is able to provide a consistent and abstract interface regardless of the underlying platform. This platform independence is accomplished through the use of a Java Virtual Machine (JVM) that emulates a computing platform itself. The JVM is provided for each actual combination of hardware and operating system upon which Java is to run. Because Java programs all appear, at the application level, to be running on the same computing platform, communication between Java applications is made significantly easier.

Java has become the development language of choice for building distributed application components. It offers (Smith 1999) :

- Portability and mobility of compiled code (class files)
- On-demand loading of functionality
- Built in support for low-level network programming
- Fine grained and configurable security control

In addition Java has additional features that make it an excellent language for other applications:

- It is completely object-orientated
- Java programs can execute multiple tasks simultaneously (multithreading)
- Memory recycling with automatic garbage collection

Object-orientated (OO) design is the art of decomposing an application into a number of objects – self contained application components that work together (Niemeyer & Peck 1996) to model some scenario. The programmer can implement these objects from scratch or they may exist already and can be re-used. Since an object is defined in terms of its interface and not by its implementation we are more concerned with *what* an object does rather than *how* it does it. All objects have state, which represents the information the object holds, and behaviour, representing what actions the object can perform. Objects interact through passing messages to one another. This is achieved through access to an object’s public methods. These methods surround and hide the object's implementation details from other objects and provide a well defined interface through which objects may communicate. Class definitions are a means of grouping objects that share common key abstractions: shared behaviour or shared properties. An object is therefore an instance of a class. The property of inheritance allows classes to inherit common properties, whilst defining specialised behaviours that distinguish it. Both of these are key properties in OO design and promote reusable code. This was a brief introduction only to some of the more important aspects of OO design. (Booch 1993) provides one of many comprehensive sources for further information.

In addition to these Java offers specific advantages for its use as the development language for mobile agent applications. These are presented in detail in the sections that follow.

2.3.1 Sockets

Java provides excellent support for low-level network socket programming. The `java.net` package provides classes for communications and working with networked resources. The socket interface provides access to standard network protocols used for communications between nodes on a network. TCP/IP communications provides for the usage of a socket where applications transmit though a data stream that may or may not be on the same host. Java supports a simplified object-oriented interface to sockets making their use considerably easier.

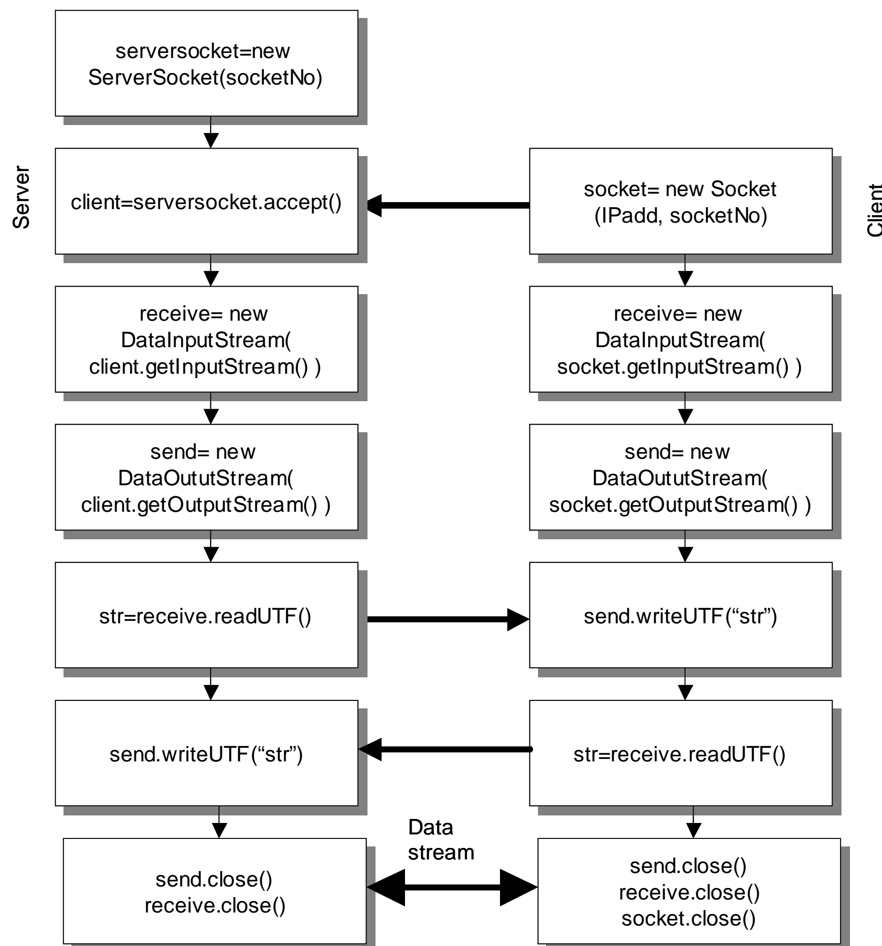


Figure 4: Socket creation in Java

Reading from and writing to a socket across a network is as easy as reading and writing any standard I/O stream. Figure 4 shows, as an example, a socket creation

using Java. In our example `ServerSocket` and `Socket` are instantiated. These objects are Java versions of a TCP socket, and are excellent examples of the advantages of OO design. We know nothing of the implementation (writing socket code is very complicated) but we do know about the interfaces, how we can get our socket objects to work for us. When a `ServerSocket` is created it listens for connection requests. When a connection request is accepted the `accept()` method returns a socket for future data transfer between the client and server. Data streams are then initialised which provide interfaces for streams that read strings and Java primitive types in a machine independent manner. Used in this manner `DataInputStream` and `DataOutputStream` provide a 'wrapper' to the socket data streams. The server then reads the output from the client and conversely supplies the client with some input using a standard method (in this case a string which has been encoded using a modified UTF-8 format). Finally the socket connections are closed.

2.3.2 Threads

Multitasking involves running several processes at a time. Multitasking programs split into a number of parts (threads) and each of these is run on the multitasking system (multithreading). A program that is running more than one thread at a time is known as a multithreaded program (Figure 5). These threads allow for smoother operation. A server application that could only handle a request from one client would be of limited use. Threads provide a means to allow an application to perform multiple tasks simultaneously. Java makes creating, controlling, and co-ordinating threads relatively simple.

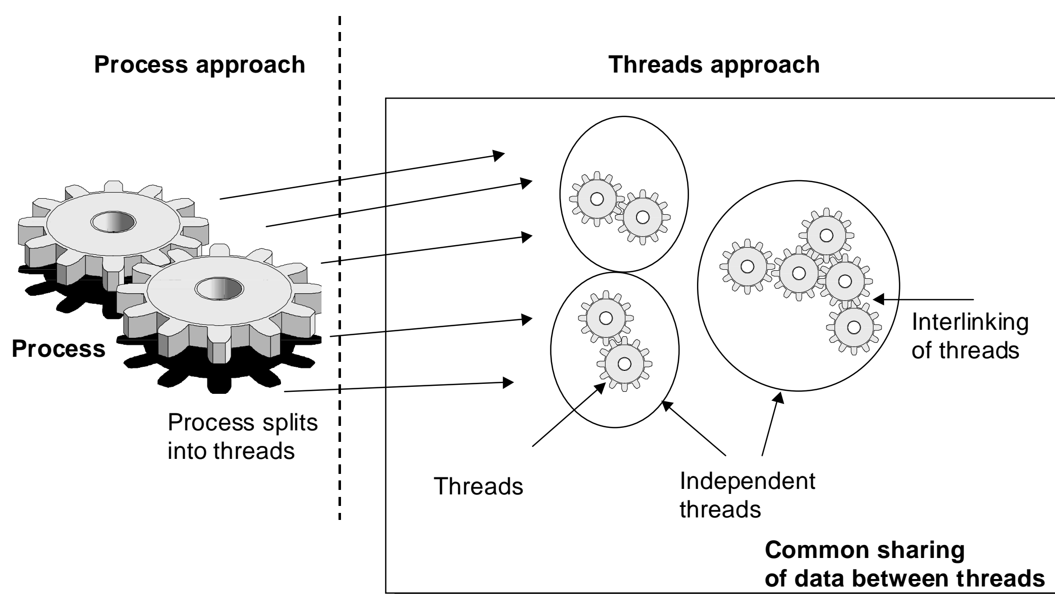


Figure 5: Process splitting into threads

2.3.3 Object Serialization

The Java API provides object serialization which is a very good mechanism for mobilising agents. Through this process an agent object may be serialized, that is converted to a stream of bytes, then written to any opened standard output stream (a file, memory, or a socket). Prior to its imminent serialization the agent must be informed so allowing it to write to the heap all information necessary for its reconstruction. The agent complete with its state may then be reconstructed from the stream of bytes at its new location in the reverse process. In adopting this serialization technique, we also encompass persistence, both mobility and persistence being properties in the first of the characteristics required in a mobile agent architecture. Java-based mobile agent systems have a lot in common. Apart from the programming language they all rely on standard versions of the Java virtual machine and Java's object serialization mechanism.

2.3.4 Security

The original security model provided by the Java platform, known as the *sandbox* model, existed in order to provide a very restricted environment in which to run untrusted code obtained from the open network. In the sandbox model, local code is trusted to have full access to vital system resources, such as the file system. Downloaded code (an applet or agent) is not trusted and can only access limited resources provided inside the sandbox. A security manager is responsible for this and subsequent platforms for determining which resource accesses are allowed. JDK 1.1 introduced the concept of a *signed applet*. A digitally signed applet is treated like local code, with full access to resources, if the public key used to verify the signature is trusted. Unsigned applets are still run in the sandbox. JDK 1.2 introduces a number of improvements over JDK 1.1, and attempts to integrate all aspects of security into a manageable whole. All code, regardless of whether it is local or remote, can now be subject to a security *policy*. The security policy defines the set of *permissions* available for code from various signers or locations and can be configured by a user or a system administrator. Each permission specifies a permitted access to a particular resource, such as read and write access to a specified file or directory or connect access to a given host and port. The runtime system organises code into individual *domains* each of which encloses a set of classes whose instances are granted the same set of permissions. This fine-grained level of security will provide a highly customisable security mechanism necessary in a mobile agent architecture.

2.4 An Evaluation of Agent Development tools

Whilst Java is an easy choice as the development language it is not so easy to pick a clear choice for our agent development tool. It would of course be possible to develop from scratch a suitable environment. Java, as we have seen in Section 2.3, supports all the necessary requirements. This however would be an undertaking out with the scope of this project. Instead it is necessary to examine those tools already available and evaluate their suitability in developing mobile agent applications.

Java has generated a flood of experimental mobile agent systems. Numerous systems are currently under development, and most of them are available for evaluation on the Web. Research during this project revealed many to choose from, making selection a difficult task. Three were selected to investigate further.

Arguably, the development environment with the most backing is ***Voyager*** from Object space. It offers a system capable of taking an existing Java class and 'treating' it to create a new class with an identical interface, known as a proxy. This proxy is an image of another class and seems just like the original to those using it. A proxy however uses network communications to create and control an instance of the real class it represents. It offers an extension to RMI. Its strength lies mainly in the integration and support of other distributed technologies, most notably CORBA.

JATLite facilitates the development of agents that exchange messages and also provides Agent Router functionality. The Agent Router allows any registered agent to send messages to any other registered agent by making a single socket connection to the Agent Router – messages are forwarded without the sending agent having to know the receiving agent's address and make a separate socket connection. All messages are buffered avoiding losses due to intermittent network problems. It is in essence a robust message passing system, in which agents communicate through passing messages. It does not however benefit from allowing agent mobility.

The ***Aglets Software Development Kit*** (ASDK) from IBM's Tokyo Research Laboratory allows the development of its mobile agent - the *aglet*. An aglet is a Java object that can move from one host on the network to another. When the aglet moves it takes the program code as well as all the objects it is carrying. In addition the aglet framework provides useful generic aglet pairs and aglet patterns, from which the development of network management tasks could evolve. These are explored in detail later.

Distributed applications can be classified into two groups:

1. those where applications are partitioned among participating nodes
2. those in which computation is migrated toward resources.

The first two development environments discussed, Voyager and JATLite are examples of the first group. The ASDK provides the mobility we explored previously in Section 2 and is an example of the second group. It will be examined further.

2.4.1 Introduction to the Aglet Object Model

The ASDK provides a framework for developing mobile agent solutions which includes a visual aglet management and monitoring GUI known as Tahiti, and an API. Termed J-APPI (Java Aglet Application Programming Interface) it provides a standard for interfacing aglets and their environments.

Its design goals are to provide (Oshima and Karjoth, 1997):

- ***Simplicity and extensibility*** – it should be easy for the Java programmer to write aglets
- ***Platform independence*** – aglets should be able to run on any agent host that supports J-APPI.
- ***Security*** – untrusted aglets should not be considered a security threat for the agent host

The Aglet Object Model describes all the abstractions and behaviours necessary to implement Java mobile agents. Figure 6 shows the major interfaces and classes defined in J-AAPI and the relationships between these.

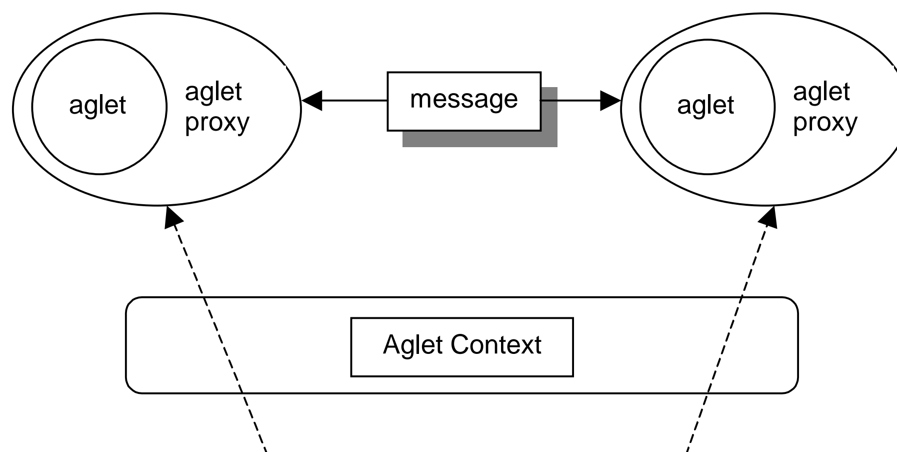


Figure 6: Aglet object model

Aglet	defines the fundamental methods for a mobile agent to control mobility and lifecycle.
Aglet Proxy	interface to act as the agent representative or shield object that protects an agent from other malicious agents. Also provides location transparency and handles message passing.
Aglet Context	the stationary, uniform execution environment for the mobile agent, implementing security and management.
Message	objects exchanged by mobile agents for the purpose of task collaboration and information sharing.

2.4.1 Aglet System Architecture

The Aglets architecture consists of two APIs and two implementation layers. Figure 7 illustrates these.

- Aglet API
- Aglets Runtime Layer - The implementation of Aglet API
- Agent Transport and Communication Interface
- Transport Layer

The Aglets runtime layer is the implementation of Aglet API, which provides the fundamental functionality such as creation, management or transfer of aglets. The transport layer is responsible for transporting an agent to the destination in the form of a byte stream that contains class definitions as well as the state of the agent. The current Aglets implementation uses the Agent Transfer Protocol (ATP), which is an application-level protocol for transmission of mobile agents. ATP is modelled on the HTTP protocol, and can be used to transfer the content of an agent in an agent-system-independent manner. To enable communication between agents, ATP also supports message passing.

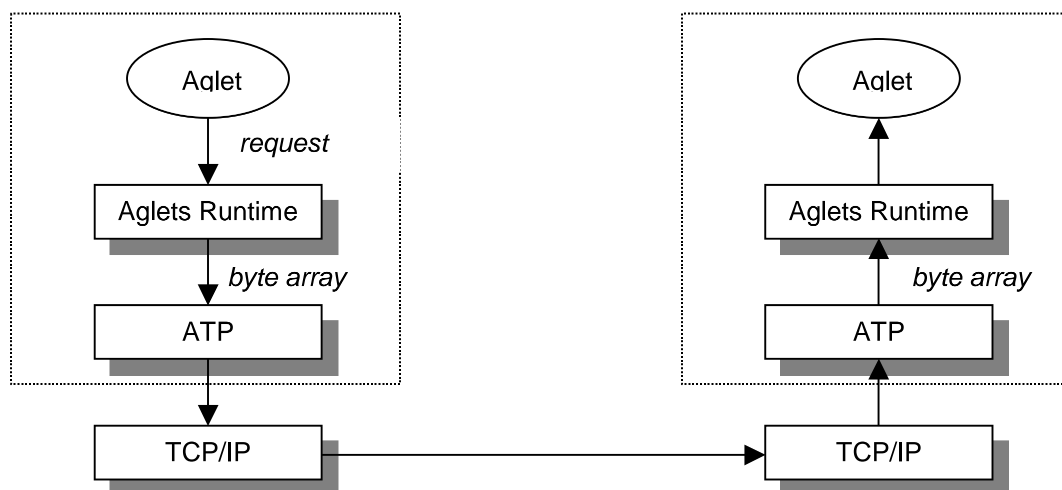


Figure 7: Aglet system architecture

When an aglet issues a request to dispatch itself to a destination, the request travels down to the Aglets runtime layer, which converts the aglet into the form of a byte array consisting of its state data and its code. If the request is successful, the aglet is terminated, and the byte array is passed to the ATP. The ATP then constructs a bit stream that contains general information such as the agent system name and agent identifier, as well as the byte array from the Aglets runtime layer.

The aglet object model and system architecture bear a reasonable resemblance to those features important in a mobile agent architecture described previously in

Section 2.2. In addition the ASDK is available as a free download and is enjoying acceptance in the mobile agent community as an appropriate environment in which to test the waters for mobile agent development. Although it is a very competitive arena and this will likely change, the ASDK is an attractive choice in that J-AAPI offers an extensible framework based on Java and the developer need not be concerned with becoming locked into a proprietary system. For these reasons the ASDK was selected as the mobile agent development toolkit for the work that follows in this project.

SECTION 3: REQUIREMENTS ANALYSIS AND DESIGN

In this section we identify and formulate our requirements analysis. Starting with the problem domain we review some of the limitations in a client/server architecture. We discuss a number of problems within that problem domain, highlighting its limitations. Then a prototype application is suggested which may be developed in order to address our problem: *do mobile agent systems offer significant advantages over client/server systems?*

Finally, in Sections 3.4 and 3.5, we set out the design approach for each of our prototypes.

1.1 The Problem Domain – Distributed Systems

Distributed network management is gaining importance due to the explosive growth of the size of computer networks. The network manager is faced with an increasingly complex job of managing a typical network. As information and its access becomes more dispersed and diverse the challenge is to discover new technologies to assist in this information management. The use of agents, in particular mobile agents is one such emerging technology that may assist in this information management process.

Typically networks and information on those networks are managed in a centralised manner, with a client/server based architecture. In the client/server paradigm, a server advertises a set of *services* that provide access to some *resources* (i.e. databases). The code that implements these services is hosted locally by the server. It is the server itself that executes the service, and thus has the *processor* capability. If the client is interested in accessing some resource hosted by the server, it will simply use one or more of the services provided by the server. The client needs some limited 'intelligence' to decide which of the services it should use. The server then holds, in one central location, all of the facilities required: the services, the resources and the processor capability.

In the mobile agent paradigm the roles of client and server are less distinct. Since the agent can migrate to the server directly, taking with it the code that implements the service; the resources and services need no longer reside at the same host. This leads to a potentially more flexible and distributed system where computation migrates toward resources.

It remains to be seen whether this potential can be realised in a practical sense by mobile agent based systems such as the ASDK, and whether these will offer significant advantages in application development over a client/server based architecture.

3.2 The Problem – Network Management Tasks Identified

A few scenarios are sketched below which demonstrate advantages to be gained from migrating computation toward resources. These serve to highlight potential benefits to be gained from a mobile agents based architecture.

- ***Data Backup Management***

As networks grow in size and complexity the job of managing a consistent backup procedure becomes increasingly difficult. Agents could be deployed to roam the network and check or confirm the backup status for every disc locally.

- ***Software Deployment***

Deploying software on a static network is relatively simple. Where users of a network are mobile and perhaps use laptops to connect to the network, only occasionally, tracking software upgrades is more difficult. Agents could be used which inform a user, upon connection to the network, of any recent software upgrades. This will make the task of tracking and informing all network users simpler.

- ***Software Acceptance Trials***

In an academic network an agent could be used to monitor how often a certain application gets accessed. The data could then be relayed to another agent responsible for collecting and collating this data. These results could then be communicated to the network manager on a daily, weekly or monthly basis and could form the basis for software acceptance trials.

- ***Accessing Databases***

Typically data may be spread over a number of databases. An agent may be given an itinerary in which it is instructed to visit a number of database servers to collect information. The agent could be employed for computationally intense retrieval tasks since the data is accessed locally and there are no problems arising from network latency.

- ***Network Usage***

Where network usage is heavy at certain key times. Agents could be deployed to visit nodes locally and gather information relevant to network use. The agents could return to a common point of origin at a more appropriate time so relieving the network of extra traffic.

- ***Intranet Web Site Monitor***

Increasingly organisations are using local web sites on their Intranets as a means to sharing and distributing information. A monitoring agent could be deployed which could sit and watch specific documents for updates. When such an update occurs the agent could inform interested parties.

3.3 Requirements Specification and Feasibility Study

The majority of management tasks discussed in the previous Section employ in some way or another the retrieval of information, whether this is through file access, making a database enquiry or by interrogating the local host. It seems obvious then to implement a prototype application that addresses this requirement. The actual mode of information retrieval need not be explicitly defined at this stage. Instead we focus on two design methods, namely a traditional client/server approach and a mobile agent approach, in order that we may compare their design and implementation methods.

We have identified then our rather general but very extensible application prototype the information retrieval agent.

Before the actual design and implementation is started, the feasibility of the project is assessed. This will mostly address technical aspects – hardware and software requirements. Much of this information can be gleaned from the research phase of the project provided this was thorough. The results of the feasibility study are summarised thus:

1. The potential advantages that mobile agents may provide over traditional client/server will be assessed through the development of a simple application – the information retrieval agent.

2. Java is identified as the ideal language for both client/server application development and mobile agent application development.

3. The Aglet Software Development Kit is identified as a suitable development environment to mobilise agents and encompasses the requirements we expect from a mobile agent architecture, as discussed in Section 2.2.3.

3.4 Client/Server Design Requirements

3.4.1 Introduction

The client/server approach to our information retrieval agent requires a number of information clients all of whom reside on their local hosts. The server application is required to process requests from one (or more) clients. The information gathered by the clients must be relayed to the server who will store the information. Figure 8 illustrates the client/server relationships. In addition the stored information must be made available to any client who so requests. This prototype application under development will be known as the info client/server.

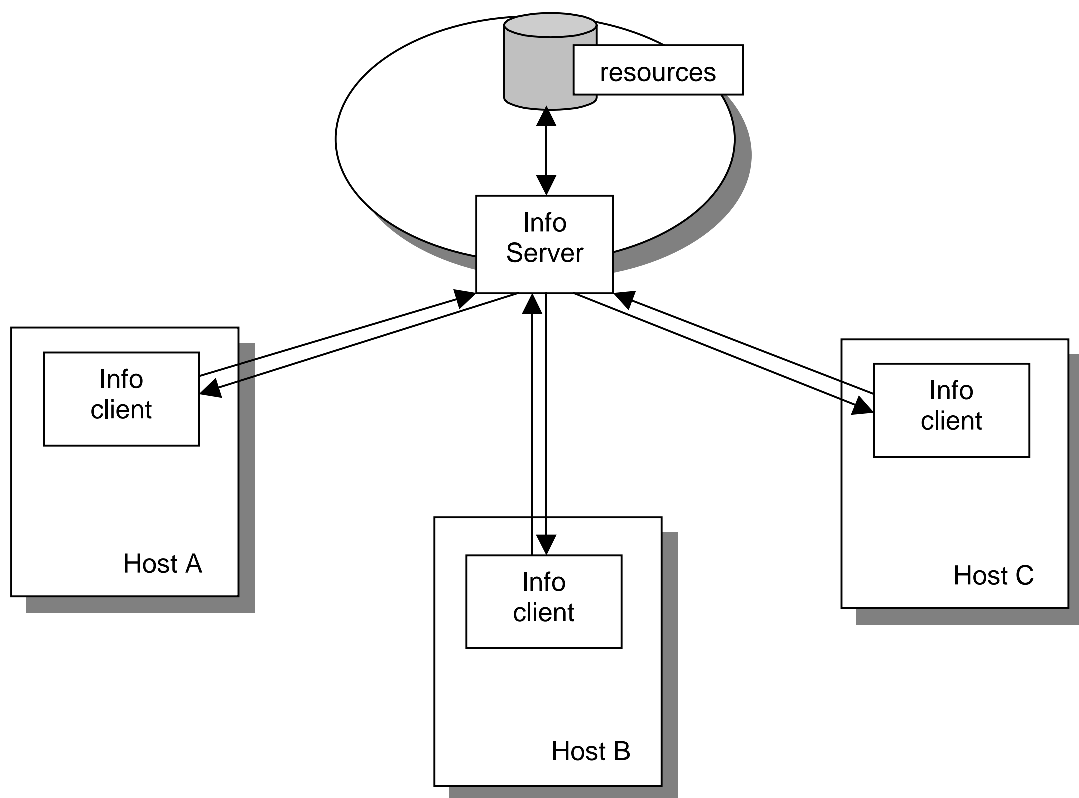


Figure 8: Info client/server

3.4.2 Protocols

A protocol is a set of procedures and customs that aid communication. When used in the context of computer networking, a protocol has a similar meaning, but is more specific. A network protocol is the set of very detailed rules, sequences, message formats, and procedures that computer systems use and understand when exchanging data with each other. Low-level protocols are concerned with how computer systems communicate with each other at the bit and byte level. The Internet Protocol (IP) operates on the internet layer, and is responsible for data routing and delivery, providing much the same functionality as the network layer in the OSI (Open Systems Interconnection) Reference Model (see Tanenbaum 1996). Above that is the transport layer, very similar to the OSI transport layer, and has two protocols defined. TCP (Transmission Control Protocol) provides a virtual connection between two systems (there may be many actual physical connections that make up the virtual connection), along with certain guarantees on the data, or 'packets', passed between the systems. Two guarantees are retransmission of packets that are dropped, and ensuring that the packets are received in the same order that they are sent. A third guarantee is that each packet received by the application has exactly the same content as when it was sent. If a bit has changed or been dropped for some reason, TCP will detect it and cause the packet to be re-transmitted. UDP (User Datagram Protocol) is the other protocol; in contrast to TCP/IP, data is transmitted in datagrams and there is no virtual connection. UDP therefore cannot provide the same guarantees so datagrams may be lost or arrive out of sequence. This reduces the overhead considerably and where performance is favoured over reliability UDP may be chosen over TCP/IP. Java supports both.

In client/server applications, the server provides some service and the client uses that service. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it. TCP provides this reliable, point-to-point communication. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

Higher-level protocols operate on the application layer and control data flow between applications. One such example is the Standard Mail Transfer Protocol (SMTP).

SMTP is the Internet's standard host-to-host mail transport protocol and traditionally operates over TCP, port 25. SMTP uses a style of asymmetric request-response protocol popular in the early 1980s. The protocol is designed to be equally useful to either a computer or a human. From the server's viewpoint, a clear set of commands is provided and well-documented (Appendix 3). For the human, all the commands are clearly terminated by new lines and a HELP command lists all of them. From the sender's viewpoint, the command replies always take the form of text lines, each starting with a three-digit code identifying the result of the operation, a continuation character to indicate another lines following, and then arbitrary text information designed to be informative to a human. If mail delivery fails, sendmail (the most important SMTP implementation) will queue mail messages and retry delivery later.

This well-known protocol was used as a basis on which to model a protocol suitable for the needs of the info client/server application. In studying the mechanics of the protocol, and taking a prompt from Watson (p.136), an e-mail agent application was built. This could be easily incorporated into the prototype applications allowing an agent to inform the network manager by e-mail of the results of its task. This represents a very interesting interface between agent and human interaction.

Our info client/server does not require a protocol as complex as that required for the purposes of sending mail. The SMTP does however illustrate the mechanics of a robust, well-defined higher level protocol. The requirements of our protocol can be demonstrated in Figure 9. The server must wait for a request to connect from the client. Assuming this initial connection is successful the server responds with a connection confirmation response, the client receives this confirmation and knows it is appropriate to proceed with the transaction. The client can then issue a command to the server to prepare to receive data it is about to send, or conversely, to request data to be sent from the server. The server responds in three ways; by accepting the data, by sending the data or by issuing an unknown request error. Finally the transaction is terminated.

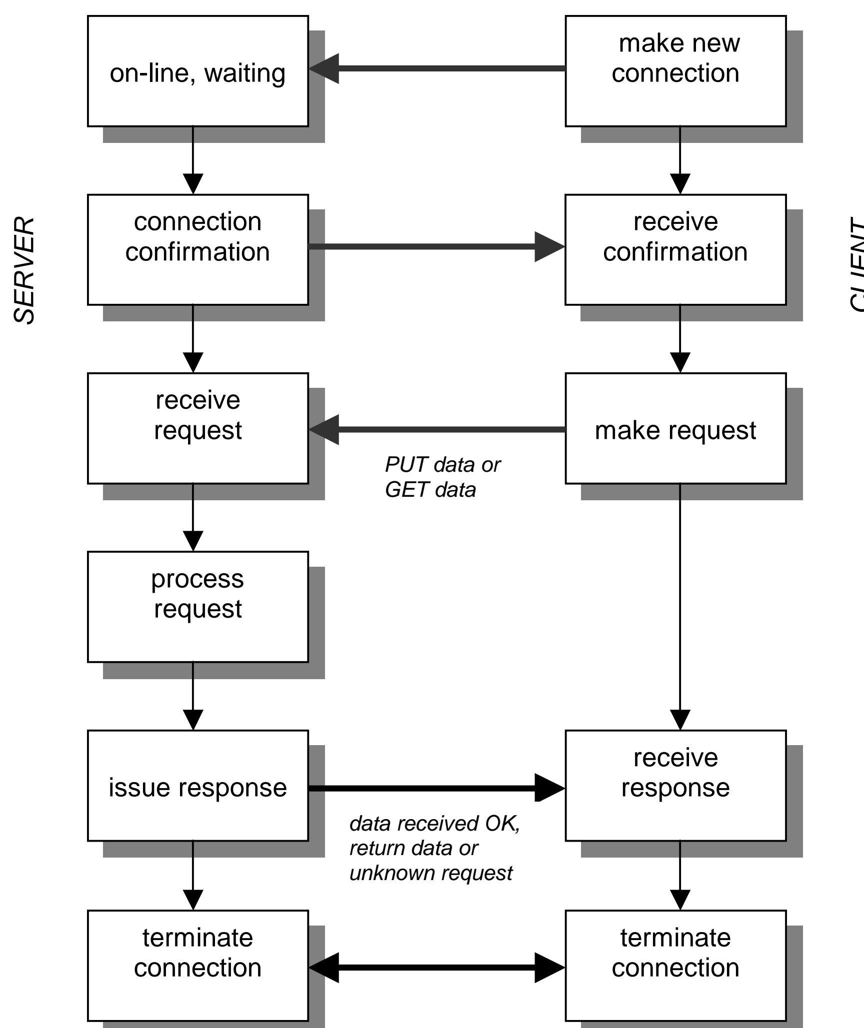


Figure 9: Info Client/Server transaction

3.4.3 Further Requirements

The transaction outlined demonstrates only one client connecting with the server. Our application must of course be able to handle multiple clients connecting possibly simultaneously.

As we have detailed in our requirements analysis we wish to gather information which is located at a particular host and assimilate that information. For the purposes of our prototype we define a local information object (LocalInfo) which will obtain various system information data relevant to the host and may include local time (and date), username and operating system. This information is only used as an example. The information object could of course form a base class from which other information objects could inherit. Such abstractions are a key property of the object orientated design principle that Java supports. Abstractions may be made that are fixed, yet represent an unbounded group of possible behaviours available through derivative or inherited classes. This has an obvious advantage in prototype design since a basic functionality framework may be defined and tested, and different areas of functionality added, where required, as the project progresses .

One of the most critical tasks that applications have to perform is to save and restore data. An OO approach to software design encourages the use of objects that encapsulate both state and behaviour. The state, represented by object state variables, and the behaviour, represented by the object methods, will typically be made up of any number of primitive data types. If we are to store or transport our object we need a means to guarantee preservation both of its state and behaviour. An object is said to be *persistent* if it can be deconstructed (to primitive data types or a bit stream) and then reconstructed, whilst maintaining its original state and behaviour. In our application the local information object encapsulates information, in theory from any number of sources that we can already define or may wish to define in the future. It is essential then that our base class, from which all other information subclasses may inherit, is persistent in nature so facilitating both transport across the network and object storage.

The final area we should address is that of a data storage mechanism. The server must have some means of storing information that can be relayed to the client upon receipt of the appropriate request. We could choose to store this data in a non-volatile manner, by writing it to a file on the disk, but this is beyond the requirements of this project. We are primarily concerned in providing a comparison between the information retrieval possibilities open to us, and a means of storing the objects during the execution lifetime of the application will suffice.

3.5 Mobile Agent Design Requirements

3.5.1. Introduction

As stated previously the agent development tool chosen for the design of a mobile agent solution was the Aglets Software Development Kit (ASDK) from IBM. In Section 2.4 we outlined some of the broader concepts underpinning its architecture and design. These will enable us to formulate our design for the information retrieval agent that will be known as the gatherInfo agent.

3.5.2 Mobility Orientated Approach

Since we are developing our prototype within an architecture that facilitates mobility, our design emphasis has shifted from a protocol orientated approach, which we saw in the previous section to a mobility orientated approach. The gatherInfo agent is able to access resources directly on the host as illustrated in Figure 10.

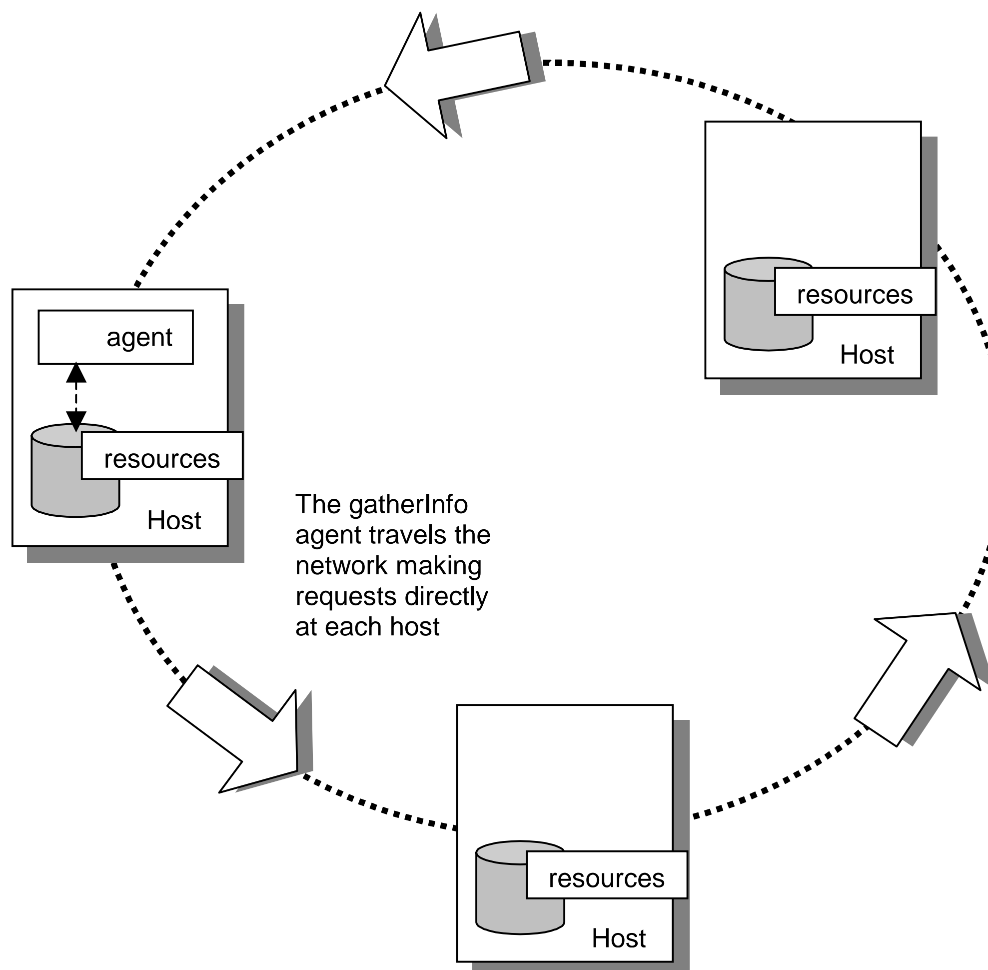


Figure 10: Agent mobility

We concern ourselves then with the practicalities of mobilising our agent. A very useful and simple approach is to devise an itinerary.

Supplied with an itinerary an agent has the knowledge about where to go and what to do before it embarks on its journey. It is a very simple, but highly effective means of mobilising our agent. The itinerary will include a list of destinations and the appropriate tasks to carry out at these destinations. In addition to these basic requirements there is some autonomous behaviour that distinguishes our agent - a means of dictating an alternative set of actions should some foreseen or unforeseen event occur.

The requirements from our agent are then to travel the network following some pre-determined itinerary. Upon arrival at each successive destination the agent will perform the information retrieval task. Upon completion of the task the agent will inform us, by returning to a specified location and printing the results. Alternatively the agent could e-mail the results.

SECTION 4: IMPLEMENTATION

This section describes the phase of the project in which the information retrieval agent was implemented and tested. Two prototypes were required, the first adopting a traditional client/server approach and the second adopting the use of mobile agents.

The realisation of the client/server solution served two purposes:

- Firstly to explore those areas of the Java programming language relevant in a mobile agent architecture as discussed in Section 2.3, and implement these in a practical sense through the building of the first prototype using a client/server approach.

- Secondly to enable a comparison and evaluation of a client/server based approach to our information retrieval problem verses a mobile agent approach.

4.1 The Info Client/Server Application

Since a major part of the exercise was to gain familiarity with the Java API, an evolutionary and iterative approach to developing the client/server prototype was required. Vilet (1997) details a number of models that suit a variety of software development strategies. One such strategy often referred to as the Boehm's 'Spiral Model' details how the problem should be broken down into a series of milestones, the most difficult of which are tackled first. This method reduces design risk by identifying those parts of the project that are perceived to be most difficult. These milestones become appraisal points upon which further design decisions may be made. In this way the spiral may be tracked as many times as necessary and the project progresses in an iterative and self-governing manner. The sub problems with the highest associated risk are solved first.

The milestones were thus identified as follows:

1. Develop a client/server communication transaction
2. Incorporate a simple protocol
3. Provide functionality to handle multiple clients
4. Implement object persistence to extend functionality

These milestones proved to be realistic targets and the development of the final prototype emerged successfully from the development of the smaller milestone target applications. These are described fully in the Project Diary that is included in Appendix 2. It proved to be an extremely valuable learning exercise highlighting some shortcomings that arise in the development of client/server applications. These are explored further as an evaluation critique in the final Section.

The final prototype incorporates all the functionality described in Section 3.3. A brief overview of the implementation details is appropriate and will be discussed next. Extracts from the code that make up the info client/server prototype, the Java classes infoServer, infoClient and LocalInfo, are listed in Appendix 5.

The prototype application runs in a DOS window and accepts user input from the keyboard. Small programs runinfoServer and runinfoClient were used to call the main classes. These were actually invoked from a batch file and shortcut from the desktop. infoClient has two constructors: if runinfoClient is supplied with a command line argument, this is passed to the relevant infoClient constructor and is the IP address of the host to which a socket connection is made. Where no address is supplied, it is assumed the server and client are running on the same address (useful for testing on a standalone desktop PC). All socket connections are made on an unreserved port, number 5000. With infoServer running any amount of clients can then be run. The Server responds to a client connection with a greeting indicating a successful connection. The client then waits for user input. The protocol mirrors a simple File Transfer Protocol (FTP) type transaction. The user may type in "PUT", in which case the LocalInfo object is sent to the server, or "GET", in which case all LocalInfo objects currently stored on the server are sent to the client. The socket connection is then terminated. The server stores all LocalInfo objects in an array, actually a Java vector object that is essentially a dynamic array.

Implementation Details Summary

The LocalInfo object used actually accesses some system properties that are normally not permitted. This demonstrates how a Java application can bypass security restrictions normally imposed by the SecurityManager on aglets (and applets).

The LocalInfo object written over the socket serves to demonstrate a very powerful Java concept, that of object persistence. This is the mechanism that allows a Java agent to be transported whilst maintaining its state of execution. Java's object serialization is the mechanism that grants mobility in all Java based agent systems.

The client/server application is of course static. Decisions about the location of the server must be made prior to implementation. Similarly, each participating host must have a client application resident locally.

The communication can only take place while the network is live, and the socket connection is maintained. Any interruptions will cause the session to terminate, while the client is transmitting data this would result in loss of that data. The protocol could be adapted to cope with such a situation, but it would however, require the complete re-establishment of the session.

The client has no means of detecting if the server has failed, until it makes a request, after which a SocketException is thrown.

Threads can use an excessive amount of memory and this implementation places no upper limit on the number of connections that are active at one time. This could cause the server to become very slow.

4.2 The gatherInfo Mobile Agent

4.2.1 Introduction to Programming Aglets

To recap, the ASDK provides a mobile agent framework written in pure Java that facilitates the development of distributed applications using a mobile agent architecture.

> *"Aglets are Java objects that can move from one host on the network to another. That is, an aglet that executes on one host can suddenly halt execution, dispatch to a remote host, and start executing again. When the aglet moves, it takes along its program code as well as the states of all the objects it is carrying. A built-in security mechanism makes it safe to host untrusted aglets."* (IBM Corp, 1998)

The aglet derives its name from a combination of agent and applet, and the similarities to the ubiquitous applet don't end there. Where an applet is event driven and provides methods that the programmer may override in order to control its life cycle, the ASDK provides a mobility orientated and mobility triggered framework.

To create an aglet the programmer creates a subclass of the class `aglet`. The `onCreation()` method may then be used to initialise the aglet in a similar way the `init()` method is the starting point for any applet. The other major methods the programmer may override to customise the behaviour of the aglet are:

`onDispatch()` called before an aglet is dispatched
`onCloning()` called before an aglet is cloned
`onDisposing()` called before an aglet is disposed (killed)

Each of these callback methods corresponds to a triggering action `dispatch()`, `clone()` and `dispose()` which is invoked by the aglet host. Each time an aglet begins execution at a host, the host invokes an initialisation method that will depend on the preceding event as follows:

`onCreation()` called the first time an aglet is born
`onClone()` called on a clone after a cloning operation
`onArrival()` called after a dispatch (or retract) action, where the aglet arrives at a new host

The `run()` method is reserved as the entry point for the aglets main thread and is invoked each time an aglet arrives at a new host, directly after the `onCreation()` or `onArrival()` methods have been called to initialise the aglet.

As a means of illustrating these methods let us consider a simple example. The code below creates a very simple aglet, easyAglet that illustrates how an aglet is created and runs within the aglet context.

```
package myAglets ;
import com.ibm.aglet.*;

public class easyAglet extends Aglet {

  private String text="I'm not born yet" ;

  public easyAglet() {
    System.out.println("Entering constuctor, easyAglet") ;
    System.out.println("Leaving constructor, easyAglet") ;
  }

  public void onCreation (Object init) {
    System.out.println("Entering onCreation method...") ;
    System.out.println("I've been passed the message :" + text)
;
    text="I've been created" ;
    System.out.println("Leaving onCreation...") ;
  }

  public void run() {
    System.out.println("Entering run method...");
    System.out.println("The message has changed :" + text) ;
    System.out.println("Leaving the run method...") ;
  }
}
```

The agent host supplied in the ASDK is called Tahiti, shown in Figure 11. It offers a graphical user interface in which to run aglets and provides a customisable security interface configurable at each aglet host.

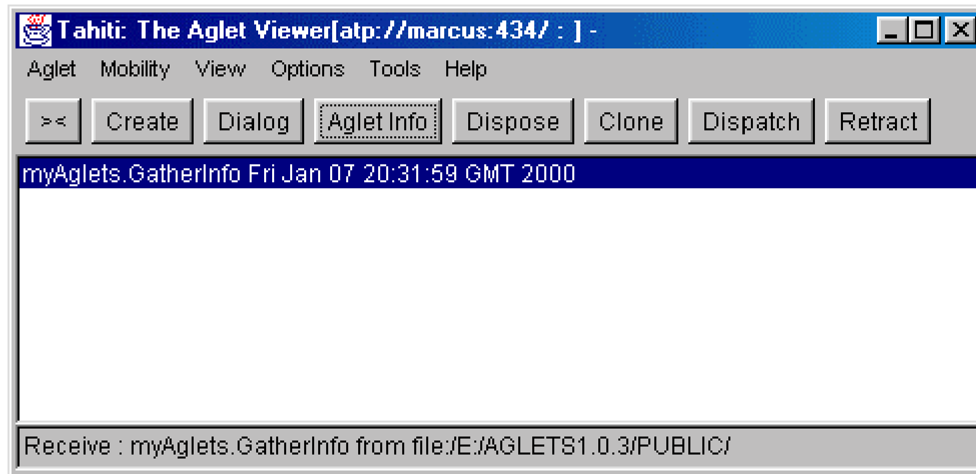


Figure 11: Tahiti the aglet GUI

At the time of writing the ASDK v. 1.0.3 supports Sun's Java Development Kit (JDK) 1.1 and later versions, but not Java 2 (which incorporates more integrated security options). Nonetheless, Tahiti provides a customisable security environment and a safe environment in which to host aglets. By default Tahiti listens for aglets on the reserved port 434. However, it can be set up to listen on any unreserved port. Two or more Tahiti servers may run alongside one another, on the same computer but on different ports, emulating a network. Tahiti provides command buttons for creating and controlling a particular aglet. Once created, an aglet may be dispatched to a remote location, and subsequently retracted. Similarly, an aglet may be cloned or disposed. Of course all of these operations can be accessed directly through programming the appropriate methods. Tahiti does however provide a useful environment in which to test aglet code.

Within each host the aglet is given a unique identity based on its class name and code base. This is known as the Global Unique Identifier (GUID). Figure 12 shows the Aglet Information dialog which details the information attributed to each aglet operating within a particular context. The owner of the aglet may also be identified by his/her name and e-mail address.

Aglets are further categorised as *trusted* or *untrusted*. They are considered to be trusted if their code base (see Figure 12) is local, that is if the class definition is loaded from a trusted host, which in the current version of the ASDK is limited to the local hard drive. All other aglets are treated as untrusted.

Security options are defined for each category of aglet (trusted or untrusted) and include:

- *file access control*
  defines the parts of a file system that are accessible in either read or write mode

- *network access control*
  reserves ports on which network connections may be made

- *properties*
  defines which system properties are made available

These properties may be set through a Security Preferences menu in Tahiti.

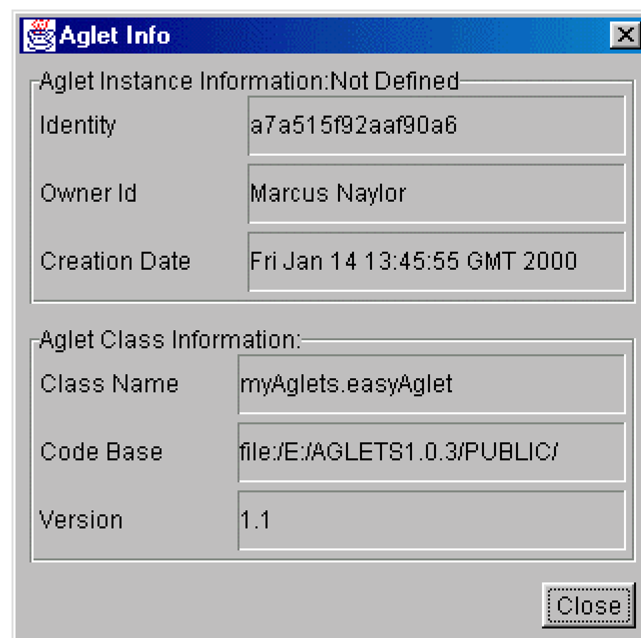


Figure 12: Aglet information dialog

Aglets communicate by message passing in a manner very similar to that already described in Section 2.2.5. An aglet wishing to communicate with another aglet first creates a message object, declaring its intent to subscribe to a particular message. The receiving aglet then uses the method `handleMessage()` to process incoming messages. A Boolean value is returned indicating whether that message is subscribed to. Messages may be synchronous or asynchronous.

We introduced a simple aglet in Section 3.5.1 for the purpose of demonstrating the construction methods to govern an aglet lifecycle.. For our prototype application we need to develop these framework methods in a slightly more sophisticated manner, these will be discussed next.

4.2.2 gatherInfo Agent Implementation

Having discussed the mechanics of the ASDK in terms of writing aglet code, we can now concentrate on our mobile agent prototype.

Our prototype requires an agent whose job it is to visit certain nodes in a network and perform an appropriate task, following a pre-determined itinerary. The ASDK supports this commonly used application through purpose built classes available in the package `com.ibm.agletx.util`.

SeqPanItinerary defines an aglet trip plan, which has a sequence of destination and message pairs. For example, the plan defined like

```
itinerary.addPlan("atp://first", "job1");
itinerary.addPlan("atp://second", "job2");
```

first dispatches an aglet to the "atp://first" and sends new Message("job1") message to the aglet. After that message handling is completed, the itinerary dispatches the aglet to the next address, "atp://second", and sends the corresponding message new Message("job2", init) to the aglet again. The order of plan is defined in the order addPlan method is called, or you can insert a new plan item at the specified index by calling insertPlanAt.

Supplied with an itinerary the aglet is dispatched, visiting each node on its journey and performing the `getLocalInfo()` task. As the aglet arrives an each network node on its itinerary, it invokes its `run()` method and displays an arrival message. The requisite task is then invoked. This occurs by pairing the itinerary destination with a message to pass to the method `handlemessage()` on arrival at that particular destination. The message may then be acted upon. Upon completion of the itinerary, the gathered information is displayed (or e-mailed to an interested party). Our mobile agent implementation makes use of the LocalInfo object (with some modifications due to security restrictions). Extracts from the code are presented in Appendix 6.

This is only one way to implement our gatherInfo retrieval agent. We could dispatch a slave which travels to the require destination and triggered by the `onArrival()` method performs the appropriate information retrieval task. However our itinerary example is more suitable since we modelled a perceived pre-determined route for our agent. The master-slave example would be more appropriate to a dynamic routing scenario. It does however illustrate the degree of flexibility open to us if we adopt the generic aglet patterns and pairs that are available for re-use with the ASDK.

Implementation Details Summary

Because the Aglet Context imposes its own SecurityManager, access to the system properties accessed in the client/server prototype is not permitted. The ASDK documentation states that this restriction can be overridden by changing some Security Preferences in Tahiti but this was not the case. (The software is still in alpha release and bugs/inconsistencies are expected). Therefore when attempting to call the LocalInfo class from within the aglet context a security exception was thrown. Attempts to bypass the security model were unsuccessful, at least illustrating the integrity of the security model. The LocalInfo object was then modified to return system variables accessible within the aglet context.

The agent property of autonomy can easily be tested. If a host to be visited is not available during the lifecycle of the aglet, an exception is thrown. This can be caught and acted upon, whether it is to wait, to follow an alternative itinerary or otherwise. In fact the class `SeqPlanItinerary` includes the method `handleTripException` for this purpose; by default the aglet will continue to the next destination, but this may be overridden.

Another key property, that of disconnected operation, was also easily demonstrated. The aglet was made to wait at each host it visited for a period of two minutes. This allowed any node on the network to be disconnected temporarily, simply by shutting down the appropriate Tahiti server, and then reconnected by restarting the Tahiti server. This had no effect upon the circulating aglet.

A final implementation of the gatherInfo aglet was tested. The agent was modified to reflect a more specific information retrieval task. The gatherInfo aglet followed the same itinerary, but this time checked the date last modified status of a given file (this could be a useful means to trigger backup procedures, or to check backup status, Section 3.2). File permissions can be granted from the Security Preferences within Tahiti. An aglet can be given permissions to access either individual files or a directory of files. The permissions set can be read access only, write only or read and write access. This is a good example of the level of fine-grained security control Java may exercise. This more specific information retrieval task demonstrates a typical scenario in which mobile agents are particularly suited. Since the results were e-mailed back to the user we can dispatch our agent and forget about it. The task could take hours or even days. The agent will continue its task in a goal-orientated and autonomous manner and communicate the results back to us upon completion.

SECTION 5: EVALUATION

We draw the project to a close by offering a critical analysis of the work undertaken and specifically to examine whether the advantages of adopting a mobile agent architecture over a client/server architecture, as discussed in Section 2.2, are indeed persuasive.

The project is then discussed in terms of its objectives and its merits on a personal note. We then recommend areas for future work and the project concludes by suggesting whilst there are still hurdles to overcome, the biggest of which is security, a mobile agent architecture can provide an extremely powerful, flexible and adaptive environment for the purpose of assisting network management. This proves advantageous over existing client/server architecture.

5.1 Mobile Agent versus Client/Server Architecture

These comments will be based on opinions arising from experimenting with the ASDK but could equally apply to the majority of Java based agent development toolkits.

The gatherInfo agent brought to light some security issues, and we saw that these could be managed to a certain extent within the Tahiti environment. Each aglet context has the potential to offer its own configurable security environment, made possible with the fine-grained security control available in Java. If the ASDK is updated to be compatible with Java 2, these security properties are likely to improve. We can then be fairly confident about how our agent system may treat a malicious agent. But we have yet to mention the implications presented by a malicious host. If there exists no mechanism to protect agents themselves from a malicious host, an attack on the agent by stealing its resources or overriding its tasks could be possible. If we confine ourselves to a network of trusted hosts this may not be a problem but if agents are to make an impact, and discover their potential as a truly distributed management solution, this is an area of utmost importance.

Adopting a mobile agent architecture doesn’t offer a solution to an insurmountable problem. Rather it provides an alternative solution. Anything that can be achieved with mobile agents can already be achieved using a centralised client/server approach. The prototypes discussed in this project demonstrate this. There is not an overwhelming argument in favour of adopting a mobile agent architecture, so what then is the impetus in researching the use of mobile agents?

Mobile agents *do* offer a more uniform approach to handling code and data in a distributed system. The client/server architecture is much more static and hard to adapt. If we consider the simple prototype developed in this project, adding extra functionality to the server would require that functionality to be mirrored by the client. The client server roles are clearly defined and spelt out in the design phase. The development process for each prototype illustrates this well. The Info client/server application was developed by first examining the client/server

transaction or protocol (Section 3.4). Whereas the development of the gatherInfo mobile agent was much less rigid, location and mobility were considered first, the task second. The mobile agent system is inherently much more flexible, and the mobile agent may encapsulate protocols. One machine can be a server and another machine can dynamically take over that role by simply having the application move.

Mobile agents also support disconnected network applications by operating in an autonomous and asynchronous manner. Dispatched agents do not require the host system to remain active while they execute (see Section 4.2.2). This has reaching implications in dynamic network scenarios where laptops (or Personal Digital Assistants or any other network compliant device) may only connect to a network for a short period. Agents can then be dispatched to perform tasks off-line and wait for reconnection before returning with the relevant task completed. This could equally apply to less reliable connection situations. Client/server applications have been developed for LAN (Local Area Network) based systems, where developers could make strong assumptions about the integrity of the communication and availability of the server. WAN (Wide Area Networks) are typically based on less reliable communications, dial-up-access, prioritised routing etc. Since mobile agent based queries and transactions are less reliant on network connections, they are ideally suited for WAN based network services.

It is more difficult to quantify the gains expected from the claim that mobile agents reduce bandwidth consumption. That is, by moving a query or transaction from the client to the server the repetitive request/response handshake is eliminated. A suitably weighty transaction would be hard to implement in the short period of time available for the project. There are however other advantages, as we have discussed, for moving the computation toward the resources and it would not be unreasonable to expect some advantages to be reaped in terms of bandwidth conservation.

The ASDK promotes rapid development by including generic aglet patterns. These patterns can easily incorporated into small development applications and other generic agent behaviours are likely to emerge. These are very easily accommodated by the OO framework that Java supports.

5.2 Evaluation of Objectives

The work on this project began in a very optimistic manner. Agent technology, or more precisely the aglet seems to offer a great deal on paper. The idea of mobile gofers scurrying around the network collaborating and communicating as personal digital assistants is a very seductive one - a big draw to this project at the outset. Where the concept seems at first simple, the practical implementations prove more challenging. The aims of this project as set out in Section 1.2 have been met. There were however difficulties experienced with the ASDK that prevented more ambitious network management tasks from being undertaken.

Unsurprisingly the greatest stumbling block was a lack of time. Work began developing aglets only after a reasonable level of competence had been gained programming in Java. Initially I attempted to get to grips with the Java AWT to provide a GUI for the aglets I was developing. I quickly realised however that this

would slow down the development process considerably. I would have liked to have been able to develop more agents and build up a collaborating network of co-operating agents. This however was impossible in the allocated time. The purpose of this work was instead to highlight the potential to be gained from the adoption of mobile agents and to attempt to clarify whether a mobile agent architecture has realistic practical benefits over a client/server architecture.

The poor documentation available with the ASDK was frustrating. The documentation supplied with the API was at best vague and at worst non-existent. This had a limiting factor on application development. The ASDK is in alpha release (in fact a beta version has recently been made available), and these hiccups are to be expected. Nonetheless this project has shown how small mobile agent applications, with a very promising potential, may rapidly be built.

A large proportion of the project time was spent discovering the subtleties of Java. Indeed it has probably been the most worthwhile aspect to this project. Whereas the mobile agent community is still in a state of flux, with the majority of systems still very much at an experimental stage, I am left in no doubt as to the importance of Java as the foundation for building distributed applications. The ASDK, an ambitious and commendable undertaking by IBM, was after all built entirely around current Java technology.

5.3 Recommendations for Further Work

A reasonable argument against the adoption of an agent architecture is that a suitable environment must first be in place, a context in which the agents can be hosted (Section 2.2.4) Adding one more application layer to the network may be undesirable. An alternative, then, would be to incorporate some form of agent context into existing software that already claims a strong network presence. The browser seems an ideal candidate for this, although it would probably be unwise to attempt this on an Internet wide scale because of security implications. An agent context within a browser used on the Intranet seems a very attractive option and could form the basis of further work.

It is likely that for the near future agents will remain relatively simple, their tasks being fairly common to those explored in this project, namely in the retrieval of information, for monitoring tasks and for real time notification. A possible avenue for further work could be in the development of a GUI environment in which to empower our agents. This make agent use more accessible and speed development Simple generic tasks such as ‘retrieve’, ‘monitor’ or ‘notify’ could be bestowed upon the agent through menu selections and the like. Alternatively simple rules based goals could be associated with them. The agents could be given itineraries or routes to take through the network through a drag and drop type interface.

To promote interoperability, some aspects of mobile agent technology have been standardised by the Object Management Group (OMG). It remains to be seen what impact increased standardisation activities will have on the mobile agent community. More research is required to investigate and draw together the various experimental agent frameworks into a more cohesive unit.

5.4 Conclusions

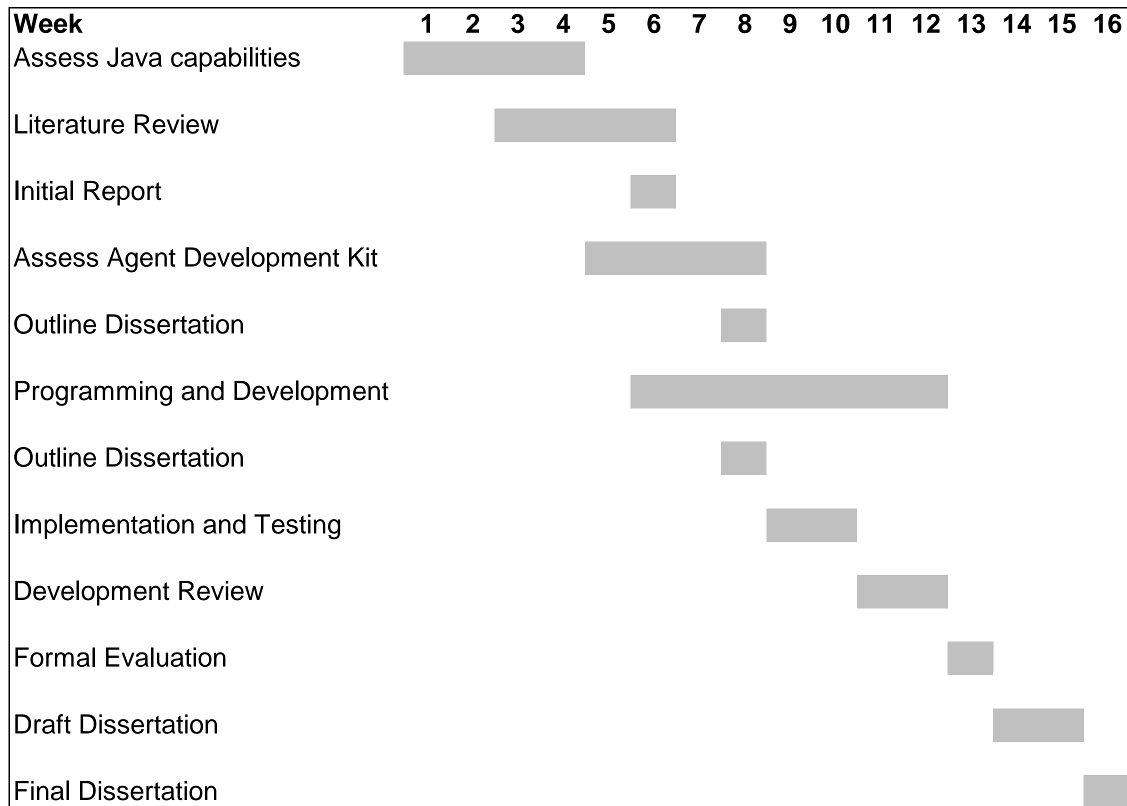
We may be some way off from allowing agents to act as our personalised digital gofers. There are serious security concerns to be addressed when adopting an agent based solution in a truly open and fully distributed sense. Agents can however offer assistance especially to the simple tasks to which they are most suited – the collection of data over a dispersed base, in operating in a disconnected network and in real time monitoring.

Network management tasks to which these may apply include gathering agents which, following a predetermined itinerary, roam the network checking, for example, the backup status of every disk hooked to the network. The agent could then report back to its point of origin. In an academic network an agent could be employed to watch how often a certain application gets accessed. This data could then be collated and could form the basis of software acceptance trials. Similarly agents could be deployed to visit nodes on the network at certain times during the day and monitor logins and so assist in collecting data regarding network usage. Other network design decisions may be helped by sending agents to remote locations to monitor the network performance.

The list of possible applications seem endless, and since the architecture is flexible and adaptive, new agents can be written to inhabit the framework with little effort. These agents will typically have only a very limited intelligence and will therefore be small and unobtrusive, unlike their weightier AI endowed brethren. The future could well provide a drag and drop environment where agents could be created and deployed by a graphical means, with certain generic tasks bestowed upon them selectable from a menu. This would surely offer network managers a powerful tool to assist them in all manner of network management tasks. The argument for the use of mobile agents over traditional client/server methods as a solution to one specific problem is not entirely persuasive. However, what is important is that the use of a mobile agent architecture offers a multitude of potential applications, all of which may be implemented in a single flexible and adaptive distributed processing environment. As increasing acceptance is gained, standards may emerge and serious security concerns overcome that will make the use of mobile agents a viable proposition.

APPENDIX 1: PROJECT PLAN

Below is a Gannt chart which outlines the project work schedule



APPENDIX 2: PROJECT JOURNAL

The following are some extracts from the project journal which was kept throughout the period of this work.

October 4, 1999	**Week 1**	Official project start date. Met my supervisor Bill Buchanan and we discussed the project proposal and set up review meetings schedule. To be held weekly on Friday. Next one scheduled Friday Week2. First steps in project to have a look into Java and sockets?
October 5, 1999		Researching Java, reading around the subject and familiarising myself with its concepts. Although the syntax is similar to C++ (with which I am already familiar) there seems little else that is comparable. Good support for OO, with multiple inheritance and polymorphism.
October 6, 1999		Read more in depth, familiarising myself with sockets and the java.net package. A quote from Niemeyer & Peck "*The network is the soul of java*" sounds promising!
October 7, 1999		Downloaded the JDK. Version 1.1.7b. (a hefty 31.3Mb) And installed it on my PC. Browsed the on line tutorial, and got familiar with the location of API documentation.
October 8, 1999		Made first progress with Java, with a few hello world type applications. It is obvious though that I will need some IDE to assist, working directly with the JDK, through DOS, and notepad is very slow and cumbersome. I need a more visual environment with help available and debugging options.
October 11, 1999	**Week 2**	Installed on my PC a copy of Microsoft Visual J++. Didn't get on to well. I believe that Microsoft Java is not pure Sun Java, and a lot of emphasis on integrating with MFC, and producing GUI interfaces for applets. Not appropriate for my needs.
October 12, 1999		Craiglockhart have Borlands Jbuilder 3 available so I spent today working through its tutorials.
October 13, 1999		As yesterday, working with Jbuilder 3.
October 14, 1999		Realised that although I was making progress with Jbuilder, it weighs in at over 100Mb, too big for me to use at home. Used the internet and www.tucows.com to locate some shareware. A nice program called Kawa 3.3.2, works like a wrapper around the JDK, giving context sensitive help (linked to Suns documentation) and a debugging window. All for only 4 Mb. Evaluation period is 30 days so I'll try it.
October 15, 1999		Made more progress with small java applications. A `whoami` application that returns a hostname, when supplied an IP address and conversely an IP address when supplied a hostname. These are methods of the `InetAddress` class, and will be used for socket communication. Meeting with Bill: reviewed my progress and discussed next steps – a client/server communication, use a protocol and build an application on top.
October 18, 1999	**Week 3**	Started literature review. Continued with Java development applications. I will split my time in the next 2 weeks between the literature review and Java development. The focus of the literature review will be to attempt to gauge the current state of affairs with agent technology in both research and commercial environments. I suspect that much of the information available will be on the internet. I have

	some sources from the literature review carried out during the project proposal, but these will have to be developed further.
October 19, 1999	Progress with Java applications much faster now as I develop further understanding and familiarity with Java, and have an appropriate IDE in which to work at home.

October 20, 1999	First client/server communication was attempted. The trickiest parts were setting up the I/O streams and exception handling, implementing sockets, once the I/O streams were in place, things were fairly straightforward. Handling I/O streams can be simplified using wrappers as follows :
	`Out=new PrintWriter`
	`(client.getOutputStream(),true)`
	
	So `PrintWriter` wraps the socket (client) and so you can access its method `println` to write data across the socket. The first communication session was as follows
	
	Server: Wait for connection
	Client: Makes connection (creates socket)
	Server: Sends connection greeting
	Client: Receives greeting
	Client: Get user input, write to server
	Server: Read message from client, echo back.
	Client: Read from server.
	
	Obviously must implement other improvements
	1. User input to end communication
	2. to take the address of the server as an argument, so can create the socket on a specified server host. (Since right now, working at home the client and server reside on the same machine)
	3. A protocol ?
	4. Handling multiple clients.

October 21, 1999	Spent the day at Craiglockart on the network. Its difficult because the labs are busy, but did get the chance to test my simple client/server application on two machines. Further improvement to client/server app. Developed a simple protocol a knock-knock application: *Server*: Wait for connection *Client*: Makes connection (creates socket) *Server*: Sends connection greeting *Client*: Receives greeting. Gets user input. *Server*: If user input is “Knock Knock”, respond with a “whos there” and set a joke flag, otherwise just echo the input. *Client:* Finish the joke. *Server:* If the joke flag is set respond with a “…. Who?” *Client*: Supply the punchline. Unfortunately it’s a little more tricky to include a sense of humour! Also the communication session can be terminated with an “adios”, typed from the user. This example demonstrates a simple protocol activated by passing string messages. Thought it may be a good idea to have a look at some well known Internet protocols to get an idea about how they are designed. The mail transfer protocol SMTP, may be suitable. We covered it briefly in lectures.
October 22, 1999	Meeting with Bill : I’m fairly happy with progress made, perhaps need to spend a little more time with the literature review.

October 25, 1999	**Week 4**	Gathering resources for literature review. Reading and research.
October 26, 1999		As yesterday, gathering resources for literature review. Having gained a reasonable level of competence with Java and what's more understanding more of the potential, I can now focus on assessing the suitability of Agent development toolkits. Identified the key property of mobility as a selection criteria, my toolkit should also be non proprietary, so I can develop applications using Java. Those under investigation : 1. Voyager 2. JATLite 3. IBM Aglets
October 27, 1999		Having read around the subject of mobile agents, potential applications and their architectures etc. I feel able to formulate the requirements spec. The outline will be as follows: Problem : *Do agent systems offer significant advantages over traditional client/server systems?* Requirements : 1. Identify problem domain (ie networks/distributed systems) 2. Identify problem tasks (ie those which may assist network management) 3. Distil these tasks into a suitable application which could be used for this project, given that the aims are to evaluate design and implementation approaches to the initially stated problem.
October 28, 1999		Further research regarding the requirements analysis. Documented those tasks pertinent to network management, and identified a common requirement, apparent in most – information retrieval. The prototype application for this project will then be an information retrieval agent, which will be developed as a client/server application and as a mobile agent application. The design and implementation approaches used can then form the basis of an evaluation to address the problem : Do agent systems offer significant advantages over client/server systems?
October 29, 1999		Meeting with Bill. Discussed the information retrieval agent with respect to a client/server application.
November 1, 1999	**Week 5**	Over the weekend did some work with the SMTP, located the definitive source for its use, document RFC 821, a very thorough example of protocol design. …"designed to make as much sense to humans as computers" (to paraphrase) this would make an interesting interface between our agents and users, the agents could be equipped with the facility to e-mail results or notification. Did some work to this effect, and largely taking a hint from (Watson, 1997) modified a sample application to send e-mails written in Java using the SMTP. Concentrate effort on literature review in coming weeks prior to the hand in of an initial report, Week 6. But also to continue with some application development, goals as of October 20. Made some progress in this area having incorporated threads into the Server application, which I worked on over the weekend, to now handle multiple clients.
November 2, 1999		Added finishing touches to Server application, MultiServer now handles multiple clients using Java threads to handle multiple connections. Its been tested, and found to work well,

		both on my machine and at Craiglockhart over multiple machines. (Using an argument passed at the command line I can specify which host the client is to make a connection to, so this makes testing easier. With no argument supplied the socket is created on the local host.) I also made the first attempts at developing the simple protocol, and incorporated the handling of PUT and GET statements, although the local information retrieved is simply the date (converted to a string) at the local host. I realise that I have some work regarding object serialization and persistence.
November 3, 1999		Concentrated on literature review. Reading around the agent toolkits I have selected for review. It covers quite a lot of technologies, CORBA, security issues, RMI etc.
November 4, 1999		More research have written a small report on my findings concerning the agent development toolkits to be included in the initial report.
November 5, 1999		Meeting with Bill: Discussed the agent toolkit I propose to develop some applications with – its mobile agent, the aglet, shares some characteristics with the applet, but of course can be made mobile, sent, with a task to some location, whereupon this task is executed. Just as the applet is driven by events, the aglet is 'mobility triggered'. So an `onarrival()` method can be overridden to perform some task.
November 8, 1999	**Week 6**	Having made the choice of which agent toolkit to use, I really need to understand its architecture to enable me to write mobile agent applications. This, at first glance seems quite daunting. The resources I've uncovered are quite limited and vague about the actual working of the aglet API and there aren't many actual examples of code. Spent most of today reading more about aglets.
November 9, 1999		Writing up initial report (the culmination of the research effort).
November 10, 1999		Continued write up.
November 11, 1999		Continued write up.
November 12, 1999		The results of the literature review were submitted to Supervisor in the form of an Initial Report (format as specified in the Dissertation Module guidelines) the results of which can be summarised: 1. Introduce the potential advantages that mobile agents may provide over traditional client/server applications 2. Identify Java as an ideal development language 3. Choose a suitable development environment in which I may develop some mobile agents for Network Management tasks. Meeting with Bill: discussed report and talked about some security issues. Java2 has an enhanced security model, I am using JDK 1.1.7b, and I'll have to investigate whether it is appropriate that the improvements to the security model warrant a switch to Java2.
November 15, 1999	**Week 7**	The switch to Java2 is inappropriate for a number of reasons 1. The ASDK doesn't as yet support Java2 2. Security and the Security Manager in Java is a vast, and complex topic. The ASDK incorporates a level of 'fine' grained control adequate for my purposes. 3. My IDE would need to be re configured and this could lose a couple of days, tweaking.

November 16, 1999		Resumed work with the ASDK. Downloaded and installed v1.0.3. Not very easy to setup.
November 17, 1999		Continued having trouble with ASDK, there doesn't seem to be much documentation and there are a multitude of environment variables to set up in batch files. The batch files themselves are quite complicated but I am making some sense of them.
November 18, 1999		Prompted by the security issues which were raised on Monday I made some investigations into security concerns, limitations, and practical implementations of the security model in the ASDK. A paper by Danny B Lange and Gunter Karjoth surmises the major threats and indicates that there is still much work to be done in this area.
November 19, 1999		Meeting with Bill. He mentioned that, if I could supply a draft paper that he might submit it for review for possible inclusion an IEEE seminar (ECBS) to be held at Napier in April.
November 22, 1999	**Week 8**	My initial report will form the basis of the paper to submit to Bill. (This could also count as the draft chapter which is due to be submitted with the Outline Dissertation, Week 10).
November 23, 1999		Working on paper.
November 24, 1999		Completed draft of Paper to submit to Bill.
November 25, 1999		Working on paper.
November 26, 1999		Meeting with Bill. Supplied him with a redraft of the paper.
November 29, 1999	**Week 9**	Finalising an annotated contents list to be supplied with the Outline Dissertation.
November 30, 1999		Completed and submitted Outline Dissertation. Continued to add functionality to the client/server application.
December 1, 1999		The client/server application is progressing well. Now called InfoClient and InfoServer, I have incorporated object persistence into the application, this actually was reasonably straightforward, once you understand the concepts. I created an object – LocalInfo, which implements Serializable (ie it can be serialized). This object, which contains information about the local hosts environment , is called from the client following a PUT command. I used various system variables, available from the `System.getProperty` method to retrieve information about username, Operating system, OS version and Java version. This is generally sensitive information and if a `securityManager` were in place access to these system properties would not be permitted, so I hope to highlight this as a contrast in the mobile agent application which will only be able to access system variables through its context (I think). So, a `Localnfo` instance is created and serialized, then passed over the socket, to the server. The server then 'reconstructs' the object in the complementary process to serialization. Although my object is quite simple this mechanism is very important. My class could be very complex with any number of differing (types) of variables. This mechanism is what gives agents in Java based systems their mobility. Very nifty.
December 2, 1999		Further work on the Info Client/Server, and the functionality now includes a dynamic array (Java `vector` object) to store the incoming LocalInfo objects. I don't think that for the purposes of this work that it is necessary to store the objects to a file. A client can make GET request, to which the server responds by sending over the socket all information gathered so far. I would like to continue tweaking and improving this application, but I feel it has demonstrated its purpose, only ever intended as a prototype, of course through OO the

Date	Week	Entry
		LocalInfo object could be an abstraction to any manner of data retrieval mechanisms – file access, database access etc. I'd like to explore these but my initial efforts with the ASDK seem to indicate that it won't be plain sailing with the aglets.
December 3, 1999		More baffling work attempting to install and get the ASDK working. Reading through the specification to familiarise myself with it.
December 6, 1999	**Week 10**	The ASDK is now installed on my machine at home and working. The lack of documentation is frustrating, but it is a free download, and only in alpha release! There are some example aglets but these don't seem to work. The aglets mailing list is a reasonably informative source of information and I'm encouraged to see I'm not the only one experiencing difficulties in getting the ball rolling.
December 7, 1999		The example aglets supplied with the ASDK don't seem to work.
December 8, 1999		I managed to get a simple example aglet working, the aglet server Tahiti can be run on the same host by specifying a different port address from the default (434). This is done by calling a switch option from the batch file that launches Tahiti. Two, or more servers can be run alongside each other. I simplified this process by writing batch files to launch Tahiti on different ports and making them execute via a shortcut on my desktop. The simple 'Hello World' type aglet was then examined, and the Tahiti controls to create, dispatch, clone and retract it examined. More promising.
December 9, 1999		The aglet development environment uses a GUI to interact with the user. My initial impressions are that if I develop applications along these lines I will become bogged down in the intricacies of the Java AWT and will not be able to devote enough time developing my mobile agent application. Developed a design strategy for my mobile agent application. Will write a gatherInfo agent which travels to different (pre specified) network nodes to carry out its task, in this case gathering LocalInfo. I hope to incorporate the LocalInfo object already written. An already obvious distinction between the two design methods, is whereas the client/server application was protocol orientated this design method emphasises the mobility requirements. How will my agent travel around the network?
December 10, 1999		Further work continued with the ASDK. Meeting with Bill: Discussed my progress. Voiced concerns about being held up by learning the Java AWT.
December 13, 1999	**Week 11**	Work continuing with the ASDK. The ASDK provides an itinerary class which provides a mechanism which my agent can utilise. An aglet can be created with an itinerary to visit any number of network hosts and perform a task, triggered by passing a message upon its arrival at the new node. This seems fine in principle but getting the code figured out is troublesome. At the same time I'm also having to attempt to learn the Java AWT, to give my aglets a suitable way to interact with the user.
December 14, 1999		Work continuing with the ASDK.
December 15, 1999		Decided to dispense with attempting to write interfaces for the aglets as it is too time consuming trying to incorporate the Java GUI interfaces. Managed to write my first aglet which simply demonstrates the execution order of each of the classes called during its

		lifecycle, `easyAglet`. This 'easy' aglet is static and first calls a constructor, then the `onCreation()` method which is similar in purpose to the applets `init()` method, and finally call a `run()` method. Also had some success with an aglet which dispatches itself from its originating host to another and writes a message upon arrival at the new host to the standard ouput, (DOS window). This looks more promising - I can concentrate on the agent mechanism without having to worry about debugging (and fully understanding) the AWT code.
December 16, 1999		More experiments with aglets. Extended the simple aglet, which dispatches itself, writes a message on the new host, (to the standard input) returns and writes a message to inform us it has returned.
December 17, 1999		Meeting with Bill. Reviewed progress.
December 20, 1999	**Week 12**	Working on the gatherInfo aglet – the aglet follows an itinerary `SeqPlanItenerary()` and prints a message to the standard output announcing its arrival. (On arrival the aglet at a new destination the aglet invokes its `run()` method.
December 21, 1999		Incorporated the `LocalInfo` object into the aglet code. However access to these system properties is prohibited within the aglet context, and Security Exceptions are thrown (the aglet still maintains its itinerary but just returns a null, instead of the information object). The aglet API states that you can grant these system privileges to an aglet through the Security Preferences menu in Tahiti, but this does not work. There is a bug, or probably this feature has yet to be implemented. As a workaround I modified the information retrieval object to request data that the aglet is privy to, through the aglet context. (Date, aglet author, aglet users email). The aglet submits a report stating where it has been, and then lists the information retrieved at each location.
December 22, 1999		Experimented demonstrating disconnected operation. The easiest (if not the most sophisticated) way of doing this was to force the aglet to wait for a period of time at each destination, 2 minutes was long enough. Then I could shut down and restart any Tahiti server, which in effect mimics a node being disconnected from the network. I will have to incorporate some exception handling code that lets the aglet user know if the aglet was unable to reach a point on its itinerary.
December 23, 1999		Made modifications to the `gatherInfo` aglet to perform a more specific task, it now follows the same itinerary but the information retrieval task is to check the date last modified status of a file. These results are then e-mailed back to me. This demonstrates quite well how useful aglets could be - interacting with users through e-mail, working autonomously, I just create it, send it and forget about it (the task set could take many hours, even days).
January 3, 2000	**Week 13**	Over the next two weeks I will be concentrating on consolidating all the work done to date, and of course writing up the final dissertation. The submission date for review by the project supervisor is 14 January.
January 10, 2000	**Week 14**	Continuing to write up dissertation.
January 17, 2000	**Week 15**	Submitted draft dissertation to supervisor, Bill Buchanan for review. The following weeks up until the hand in date will involve finishing touches (some diagrams to work on) and re drafts of the written work.

APPENDIX 3: SMTP PROTOCOL

The document RFC 821 from the Information Sciences Institute, University of Southern California specifies the complete workings of the Standard Mail Transfer Protocol. Some extracts and specifications are presented.

4.1. SMTP Commands

4.1.1. Command Semantics

The SMTP commands define the mail transfer or the mail system function requested by the user. SMTP commands are character strings terminated by <CRLF>. The command codes themselves are alphabetic characters terminated by <SP> if parameters follow and <CRLF> otherwise. The syntax of mailboxes must conform to receiver site conventions. The SMTP commands are discussed below. The SMTP replies are discussed in the Section 4.2.

HELLO (HELO)
This command is used to identify the sender-SMTP to the receiver-SMTP. The argument field contains the host name of the sender-SMTP.

MAIL (MAIL)
This command is used to initiate a mail transaction in which the mail data is delivered to one or more mailboxes. The argument field contains a reverse-path.

RECIPIENT (RCPT)
This command is used to identify an individual recipient of the mail data; multiple recipients are specified by multiple use of this command.

DATA (DATA)
The receiver treats the lines following the command as mail data from the sender. This command causes the mail data from this command to be appended to the mail data buffer. The mail data may contain any of the 128 ASCII character codes.

SEND (SEND)
This command is used to initiate a mail transaction in which the mail data is delivered to one or more terminals. The argument field contains a reverse-path. This command is successful if the message is delivered to a terminal.

SEND OR MAIL (SOML)
This command is used to initiate a mail transaction in which the mail data is delivered to one or more terminals or mailboxes. For each recipient the mail data is delivered to the recipient's terminal if the recipient is active on the host (and accepting terminal messages), otherwise to the recipient's mailbox. The argument field contains a reverse-path. This command is successful if the message is delivered to a terminal or the mailbox.

SEND AND MAIL (SAML)
This command is used to initiate a mail transaction in which the mail data is delivered to one or more terminals and mailboxes. For each recipient the mail data is delivered to the recipient's terminal if the recipient is active on the host (and accepting terminal messages), and for all recipients to the recipient's mailbox. The argument field contains a reverse-path. This command is successful if the message is delivered to the mailbox.

RESET (RSET)
This command specifies that the current mail transaction is to be aborted. Any stored sender, recipients, and mail data must be discarded, and all buffers and state tables cleared. The receiver must send an OK reply.

VERIFY (VRFY) This command asks the receiver to confirm that the argument identifies a user. If it is a user name, the full name of the user (if known) and the fully specified mailbox are returned.

QUIT (QUIT)
This command specifies that the receiver must send an OK reply, and then close the transmission channel.

4.2. SMTP REPLIES

Replies to SMTP commands are devised to ensure the synchronization of requests and actions in the process of mail transfer, and to guarantee that the sender-SMTP always knows the state of the receiver-SMTP. Every command must generate exactly one reply.

An SMTP reply consists of a three digit number (transmitted as three alphanumeric characters) followed by some text. The number is intended for use by automata to determine what state to enter next; the text is meant for the human user. It is intended that the three digits contain enough encoded information that the sender-SMTP need not examine the text and may either discard it or pass it on to the user, as appropriate. In particular, the text may be receiver-dependent and context dependent, so there are likely to be varying texts for each reply code. Formally, a reply is defined to be the sequence: a three-digit code, <SP>, one line of text, and <CRLF>, or a multiline reply Only the EXPN and HELP commands are expected to result in multiline replies in normal circumstances, however multiline replies are allowed for any command.

4.2.1. REPLY CODES BY FUNCTION GROUPS

500 Syntax error, command unrecognized [This may include errors such as command

line too long]

501 Syntax error in parameters or arguments

502 Command not implemented

503 Bad sequence of commands

504 Command parameter not implemented

211 System status, or system help reply

214 Help message [Information on how to use the receiver or the meaning of a particular non-standard command; this reply is useful only to the human user]

220 <domain> Service ready

221 <domain> Service closing transmission channel

421 <domain> Service not available, closing transmission channel [This may be a reply to any command if the service knows it must shut down]

250 Requested mail action okay, completed

251 User not local; will forward to <forward-path>

450 Requested mail action not taken: mailbox unavailable [E.g., mailbox busy]

550 Requested action not taken: mailbox unavailable [E.g., mailbox not found, no access]

451 Requested action aborted: error in processing

551 User not local; please try <forward-path>

452 Requested action not taken: insufficient system storage

552 Requested mail action aborted: exceeded storage allocation

553 Requested action not taken: mailbox name not allowed [E.g., mailbox syntax incorrect] 354 Start mail input; end with <CRLF>.<CRLF>

554 Transaction failed

APPENDIX 4: EXTRACTS OF CODE – E MAIL AGENT

(Adapted from original code by Harm Verbeek, in Watson (1997))

```
public class SendMail
{
  static Socket          socket;
  static InputStream     in;
  static OutputStream    out;
  static DataInputStream din;
  static PrintStream     prout;


  public SendMail(String mailhost, String domain, String my_address,
                  String to_address, String data)
  {
    int             SMTPport =  25;
    String          incoming = new String();

    String          MailHost = mailhost;

    String          HELO     = "HELO " + domain;
    String          MAILFROM = "MAIL FROM:<" + my_address + ">";
    String          RCPTTO   = "RCPT TO:<" + to_address + ">";
    String          DATA     = "DATA";

    /* you must terminate your message string with
       "\r\n.\r\n" to indicate end of message.
    */
    String          Msg      = data + "\r\n.\r\n";

    /* mailhost returns either "220" or "250"
       to indicate everything went OK
    */
    String          OKCmd    = "220|250";


/* connect to the mail server */
    System.out.println("Connecting to " + MailHost + "...");
    System.out.flush();

    try {
      socket = new Socket(MailHost, SMTPport);
    } catch (IOException e) {
      System.out.println("Error opening socket.");
      return;
    }

    try {
      in    = socket.getInputStream();
      din   = new DataInputStream(in);

      out   = socket.getOutputStream();
      prout = new PrintStream(out);
    }
    catch (IOException e) {
      System.out.println("Error opening inputstream.");
      return;
    }
```

```
 /* OK, we're connected, let's be friendly and say hello to the mail
server... */
    prout.println(HELO);
    prout.flush();
    System.out.println("Sent: " + HELO);

    try {
      incoming = din.readLine();
    }
    catch (IOException e) {
      System.out.println("Error reading from socket.");
      return;
    }
    System.out.println("Received: " + incoming);

 /* let server know YOU wanna send mail... */
    prout.println(MAILFROM);
    prout.flush();
    System.out.println("Sent: " + MAILFROM);

    try {
      incoming = din.readLine();
    }
    catch (IOException e) {
      System.out.println("Error reading from socket.");
      return;
    }
    System.out.println("Received: " + incoming);

/* let server know WHOM you're gonna send mail to... */
    prout.println(RCPTTO);
    prout.flush();
    System.out.println("Sent: " + RCPTTO);

    try {
      incoming = din.readLine();
    }
    catch (IOException e) {
      System.out.println("Error reading from socket.");
      return;
    }
    System.out.println("Received: " + incoming);


/* let server know you're now gonna send the message contents... */
    prout.println(DATA);
    prout.flush();
    System.out.println("Sent: " + DATA);

    try {
      incoming = din.readLine();
    }
    catch (IOException e) {
      System.out.println("Error reading from socket.");
      return;
    }
    System.out.println("Received: " + incoming);

/* finally, send the message... */
    prout.println(Msg);
    prout.flush();
```

```
    System.out.println("Sent: " + Msg);

    try {
      incoming = din.readLine();
    }
    catch (IOException e) {
      System.out.println("Error reading from socket.");
      return;
    }
    System.out.println("Received: " + incoming);

/* we're done, disconnect from server */
   System.out.print("Disconnecting...");
    try {
      socket.close();
    }
    catch (IOException e) {
      System.out.println("Error closing socket.");
    }

    System.out.println("done.");
  }

}
```

APPENDIX 5: EXTRACTS OF CODE : INFO CLIENT/SERVER

infoServer.java

```
class infoServer {

  infoServer (int port) throws IOException {

  ServerSocket server = null ;
  Socket connection = null ;
  boolean listening = true ;
  Vector infoStore = new Vector() ;


    System.out.println("Running InfoSERVER\n" + "Waiting ...") ;
    try {
      server=new ServerSocket(port) ;
    } catch (IOException e) {
      System.err.println("Error : Port " + port + "unavailable") ;
    }

    while (listening) {
      try {
        connection=server.accept() ;// waits for connection

      } catch (IOException e) {
        System.err.println("Error : " + e) ;
      }
      System.out.println("Connection received on Port " + port + "
...\n") ;
      new infoServerThread(connection,infoStore).start() ;
    }
  }
}

class infoServerThread extends Thread {
  private Socket connection ;
  private PrintWriter out ;
  private BufferedReader in ;
  private Vector store ;

  public infoServerThread(Socket connection, Vector store) {
    this.connection=connection ;
    this.store=store ;

    try {
      out=new PrintWriter(connection.getOutputStream(),true) ;
      in=new BufferedReader(new
InputStreamReader(connection.getInputStream())) ;
    } catch (IOException e) {
      System.out.println("Error : " + e) ;
    }
  }

  public void run() {
    String input, output ;

    try {
      out.println("Hurray we're connected") ; // server greetings
```

```
        input=in.readLine() ;                    // get client
request//System.out.println(input) ;
        infoProtocol(input) ;                    // act on request
      } catch (IOException e) {
        System.out.println("Error : " + e) ;
      }

      try {
          System.out.println("Closing socket ...") ;
          out.close() ;      // cleanup
          in.close() ;
          connection.close() ;
      } catch (IOException e) {
        System.out.println("Error : " + e) ;
      }
    }

    private void infoProtocol (String action) {
      String echo ="Server :" ;
      String output ;

      if (action.equalsIgnoreCase ("PUT")) {
        out.println(echo+ "Ready to receive data") ;
        readInfo() ;
      }
      else if (action.equalsIgnoreCase ("GET")) {
        out.println(echo+ "Sending Data ...") ;
        writeInfo() ;
      }
      else out.println(echo+ "Error command not recognised") ;
    }

    private void readInfo() {
      LocalInfo newInfo ;

      try {
        ObjectInputStream infoIn=new
ObjectInputStream(connection.getInputStream()) ;
        newInfo=(LocalInfo)infoIn.readObject();
        System.out.println("Object Serialized !!!") ;
        System.out.println(newInfo.getAll()) ;

        // Need to store a list (Array) of LocalInfo objects.
        store.addElement(newInfo) ;

      } catch (IOException e) {
        System.out.println("Error :" + e) ;
      } catch (ClassNotFoundException Ce) {
        System.out.println("Error :" +Ce) ;
      }

    }

      private void writeInfo() {
      try {
        Enumeration e=store.elements() ;
        while (e.hasMoreElements()) {
          LocalInfo i=(LocalInfo)e.nextElement() ;
          //System.out.println(i.getAll()) ;
        }
```

```
        ObjectOutputStream infoOut=new
ObjectOutputStream(connection.getOutputStream()) ;
        infoOut.writeObject(store);
        infoOut.flush();
     } catch (IOException e) {
        System.out.println("Error :" + e) ;
     }
   }

}
```

infoClient.java

```
class infoClient {
  private Socket theclient=null ;
  private InetAddress host ;

  infoClient (int port) throws IOException {
    InetAddress local=InetAddress.getLocalHost() ;
    theclient=this.createsocket(local,port) ;
    this.Do(null) ;
  }

  infoClient (String hostString, int port) throws IOException {
    InetAddress host = InetAddress.getByName(hostString) ;
    theclient=this.createsocket(host,port) ;
    this.Do(null) ;
  }

  private Socket createsocket(InetAddress host, int port) throws
IOException {
    Socket client = null ;
    try {
      client= new Socket(host,port) ;
      //out= new PrintWriter(client.getOutputStream(), true) ;
      //in= new BufferedReader(new
InputStreamReader(client.getInputStream())) ;


    } catch (UnknownHostException e) {
       System.err.println("Can't find host") ;
    }  catch (IOException e) {
       System.err.println("Error : " + e) ;
    }
    System.out.println("Running infoCLIENT") ;
    System.out.println("Created socket : " + client.toString() +
"\n") ;
  return client ;
  }

  private void Do(String action) throws IOException {
    PrintWriter out=new PrintWriter(theclient.getOutputStream(),
true) ;
    BufferedReader in=new BufferedReader(new
InputStreamReader(theclient.getInputStream())) ;
    BufferedReader userin=new BufferedReader(new
InputStreamReader(System.in)) ;
    Vector info=new Vector() ;
    //String userInput ;

    System.out.println(in.readLine()) ;      // get welcome message
    action=userin.readLine() ;
    out.println(action) ;                    // send client request
    System.out.println(in.readLine()) ;      // get sever response

    if (action.equalsIgnoreCase ("PUT")) {   // write object
      ObjectOutputStream infoOut=new
ObjectOutputStream(theclient.getOutputStream()) ;
      infoOut.writeObject(new LocalInfo());
    }
```

```
    if (action.equalsIgnoreCase ("GET")) {     // or read object(s)
      ObjectInputStream infoIn=new
ObjectInputStream(theclient.getInputStream()) ;
      try {
        info=(Vector)infoIn.readObject();
        System.out.println("Data returned from Server ....\n") ;
      } catch (ClassNotFoundException Ce) {
        System.out.println("Error :" +Ce) ;
      }

      Enumeration e=info.elements() ;
      while (e.hasMoreElements()) {
        LocalInfo i=(LocalInfo)e.nextElement() ;
        System.out.println(i.getAll()) ;
        System.out.println() ;
      }
    }


    userin.close();
    out.close();     // cleanup
    in.close();
    theclient.close();

  }

}
```

LocalInfo.java

```
class LocalInfo {

   private String _hostName="Unknown" ;
   private String _userName="Unknown" ;
   private String _osName="Unknown" ;
   private String _osVersion="Unknown" ;
   private String _javaVersion="Unknown" ;
   private Date _localTime ;

  LocalInfo() {
   try {
      InetAddress localHost=InetAddress.getLocalHost() ;
      _hostName=localHost.toString() ;
   } catch (Exception e) {
         System.err.println("Error : " + e) ;
   }

   _userName=System.getProperty("user.name") ;
      _osName=System.getProperty("os.name") ;
   _osVersion=System.getProperty("os.version") ;
   _javaVersion=System.getProperty("java.version") ;
   _localTime=new Date() ;
   }

   String getHostName() {
  return _hostName;
  }

  String getUserName() {
  return _userName;
  }

   String getOsName() {
  return _osName;
   }

   String getOsVersion() {
   return _osVersion;
   }

   String getJavaVersion() {
   return _javaVersion;
   }

   Date getLocalTime() {
   return _localTime;
   }
   String getAll() {
      String str =
         "Host Name    :" + _hostName + "\n" +
         "User Name    :" + _userName + "\n" +
         "OS Name      :" + _osName + "\n" +
         "OS Version   :" + _osVersion + "\n" +
         "Java Version:" + _javaVersion + "\n" +
         "Local Time   :" + _localTime ;
   return str;
  }
}
```

APPENDIX 6: EXTRACTS OF CODE : GATHER INFO AGENT

```
import com.ibm.aglet.*;
import com.ibm.aglet.util.*;
import com.ibm.agletx.util.SeqPlanItinerary;
import java.util.* ;

public class GatherInfo extends Aglet {
     StringBuffer buffer;
     SeqPlanItinerary itinerary;

   public void onCreation(Object ini) {
      itinerary = new SeqPlanItinerary(this);
      itinerary.addPlan("atp://marcus:500/", "getLocalInfo");
      itinerary.addPlan("atp://marcus:501/", "getLocalInfo");
      // return to host and print result

   itinerary.addPlan(getAgletContext().getHostingURL().toString(),"pr
intResult");
   this.start() ;
   }

   public boolean handleMessage(Message msg) {
      if (msg.sameKind("getLocalInfo")) {
        getLocalInfo(msg);
        return true;
      } else if (msg.sameKind("printResult")) {
        System.out.println(buffer);
      return true;
      }
   return false;
   }

  public void run() {
      System.out.println("Hello, just visiting...") ;
   }

  public void oncemore() {
      try {
        itinerary.startTrip();
      } catch (Exception ex) {
        ex.printStackTrace();
      }
   }

  public void start() {
      buffer = new StringBuffer();
      oncemore();
   }

  public void getLocalInfo(Message msg) {
      AgletContext ac = getAgletContext();
      setText("Here I am about to gather info, and waiting ...") ;
      // waitMessage(2 *  1000);
      buffer.append("I've been to " + ac.getHostingURL().toString() +
"\n") ;
      buffer.append(new LocalInfo().getAll()) ;
      buffer.append("\n") ;
   }
}
```

APPENDIX 7: AGLETS API CLASS HIERARCHY

- class java.lang.Object
    - class com.ibm.aglet.Aglet (implements java.io.Serializable)
        - class com.ibm.agletx.patterns.Messenger
        - class com.ibm.agletx.patterns.Notifier
        - class com.ibm.agletx.patterns.Slave
    - interface com.ibm.aglet.AgletContext
    - class com.ibm.aglet.event.AgletEventListener
    - class com.ibm.aglet.AgletID (implements java.io.Serializable)
    - class com.ibm.aglet.AgletInfo
      (implements java.io.Serializable, java.lang.Cloneable)
    - interface com.ibm.aglet.AgletProxy
    - class com.ibm.aglet.system.AgletRuntime
    - class com.ibm.aglet.AgletStub
    - class com.ibm.aglet.system.Aglets
    - class com.ibm.aglet.event.CloneAdapter
(implements com.ibm.aglet.event.CloneListener)
    - interface com.ibm.aglet.event.CloneListener
(extends java.util.EventListener, java.io.Serializable)
    - class java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
        - class java.awt.Container
            - class java.awt.Panel
                - class com.ibm.aglet.util.AddressChooser
                  (implements java.awt.event.ActionListener)
            - class java.awt.Window
                - class com.ibm.aglet.util.AddressBook (implements java.awt.event.ActionListener, java.awt.event.ItemListener, java.awt.event.FocusListener)
    - class com.ibm.aglet.system.ContextAdapter
(implements com.ibm.aglet.system.ContextListener)
    - interface com.ibm.aglet.system.ContextListener
(extends java.util.EventListener)
    - class java.util.Dictionary
        - class java.util.Hashtable
(implements java.lang.Cloneable, java.io.Serializable)
            - class com.ibm.aglet.util.Arguments
    - class java.util.EventObject (implements java.io.Serializable)
        - class com.ibm.aglet.event.AgletEvent
            - class com.ibm.aglet.event.CloneEvent
            - class com.ibm.aglet.system.ContextEvent
            - class com.ibm.aglet.event.MobilityEvent
            - class com.ibm.aglet.event.PersistencyEvent
    - class com.ibm.aglet.FutureReply
    - interface com.ibm.aglet.util.ImageData
    - class com.ibm.agletx.patterns.Meeting (implements java.io.Serializable)
    - class com.ibm.aglet.Message (implements java.io.Serializable)
    - interface com.ibm.aglet.MessageManager
    - class com.ibm.aglet.event.MobilityAdapter (implements com.ibm.aglet.event.MobilityListener)
        - class com.ibm.agletx.util.AlternateItinerary
(implements java.io.Serializable)
        - class com.ibm.agletx.util.MetaItinerary
(implements java.io.Serializable)
        - class com.ibm.agletx.util.SeqItinerary

(implements java.io.Serializable)

      - class com.ibm.agletx.util.MeetingsItinerary
      - class com.ibm.agletx.util.MessengerItinerary
      - class com.ibm.agletx.util.SeqPlanItinerary
      - class com.ibm.agletx.util.SlaveItinerary
    - class com.ibm.agletx.util.SimpleItinerary

(implements java.io.Externalizable)

  - interface com.ibm.aglet.event.MobilityListener

(extends java.util.EventListener, java.io.Serializable)

  - class com.ibm.agletx.patterns.NetUtils
  - class com.ibm.aglet.event.PersistencyAdapter

(implements com.ibm.aglet.event.PersistencyListener)

  - interface com.ibm.aglet.event.PersistencyListener

(extends java.util.EventListener, java.io.Serializable)

  - class com.ibm.aglet.ReplySet
  - class com.ibm.agletx.util.Task (implements java.io.Serializable)
    - class com.ibm.agletx.util.MeetingTask
  - class java.lang.Throwable (implements java.io.Serializable)
    - class java.lang.Exception
      - class com.ibm.aglet.AgletException
        - class com.ibm.aglet.AgletNotFoundException
        - class com.ibm.aglet.InvalidAgletException
        - class com.ibm.aglet.MessageException
        - class com.ibm.aglet.NotHandledException
        - class com.ibm.aglet.RequestRefusedException
      - class java.io.IOException
        - class com.ibm.aglet.ServerNotFoundException

REFERENCES

Agents to roam the Internet [WWW document].
URL http://www.sunworld.com/swol-10-1996/swol-10-agent.html

Aglets - Mobile Java Agents. [WWW document].
URL http://www.trl.ibm.co.jp/aglets/whitepaper.html

Booch, Grady (1993). "Object-oriented Analysis and Design with Applications". Benjamin Cummings.

Cheng, D.T, & Covaci, S. (1997). "The OMG Mobile Agent Facility: A Submission, Mobile Agents, Springer Verlag.

Grand, Mark (1999). "Patterns in Java Volume 2". John Wiley & sons.

Gray, R.S. (1995). "Agent TCL: A transportable agent system". Proceedings of the CIKM Workshop on Intelligent Information Agents, Baltimore, MD.

Harrison, C., Chess, D., and Kershenbaum (1995). "Mobile Agents: Are they a good idea?" IBM T.J. Watson Research Centre. NY.

IBM Aglets Software Development Kit (1998). Home page [WWW document].
URL http://www.trl.ibm.co.jp/aglets/

Ince, Darrel & Freeman, Adam (1997). "Programming the Internet with Java". Addison-Wesley.

JAT*Lite* Introductory FAQ [WWW document].URL http://java.stanforrd.edu

JavaWorld Magazine [WWW]
URL http://www.javaworld.com

Karjoth, G., D.B. Lange and M. Oshima (1997), "A Security Model for Aglets," *IEEE Internet Computing* 1, 4, 68-77.

Kautz, H.B., Selmen, & Coen (1994). "Bottom –up Design of Software Agents". Communications of the ACM , 37, 7.

Lange, Danny B (1997). "Java Aglet Application Programming Interface (J-AAPI) White Paper - Draft 2"
[WWW document].URL http://www.trl.ibm.co.jp/aglets/JAAPI-whitepaper.html

Maes, Pattie (1995). "Artificial Life Meets Entertainment: Life like Autonomous Agents". Communications of the ACM, 38, 11, 108-114.

Niemeyer, P and Peck, J (1996). "Exploring Java". O'Reilly and Associates.

ObjectSpace: Voyager Overview [WWW document].
URL [http://www.objectspace.com/products/vgrOverview.htm]

Oshima, M and Karjoth, G (1997). "Aglets Specification (1.0)". [WWW document].
URL http://www.trl.ibm.co.jp/aglets/spec10.html

Redmill, Felix (1997). "Software Projects Evolutionary vs Big Bang Delivery". John Wiley & Sons.

Russell, Stuart. J. and Norvig, Peter (1995). "Artificial Intelligence: A Modern Approach". Eaglewood Cliffs, NJ. Prentice Hall.

Smith, Michael (1999). "Java an Object-Orientated Language". McGraw-Hill (UK).

Sommers, Bret (1997). "Agents: Not just for Bond anymore.". [WWW document].
URL http://www.javaworld.com/javaworld/jw-04-1997/jw-04-agents.html

Sunsted, Todd (1998). "Agents talking to agents". [WWW document].
URL http://www.javaworld.com/javaworld/jw-09-1998/jw-09-howto.html

Sunsted, Todd (1998). "An introduction to Agents". [WWW document].
URL http://www.javaworld.com/javaworld/jw-06-1998/jw-06-howto.html

Tanenbaum, Andrew S (1996). "Computer Networks" 3rd Ed. NJ. Prentice Hall.

The Java Tutorial. [WWW document].
URL http://java.sun.com/docs/books/tutorial/index.html

Venners, Bill (1997). "Solve real problems with aglets, a trype of mobile agent". [WWW document].URL [http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html]

Venners, Bill (1997). "Under the Hood: The architecture of aglets". [WWW document].URL [http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html]

Vilet, Hans van (1997). "Software Engineering Principles and Practice". John Wiley & sons.

Watson, Mark. (1997). "Intelligent Java Applications for the Internet and Intranets". Morgan Kauffman Publishers.

White, J. (1997) "Telescript Technology: Mobile Agents". *Software Agents*. MIT Press.