Enhancing Network Management using Mobile Agents

WJ Buchanan, M Naylor and AV Scott
Napier University, Edinburgh, UK
{ bill | andrea } @dcs.napier.ac.uk; mrmarcus@mail.com

Abstract

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. There has been a great deal of attention on the potential productivity gains expected from so-called intelligent agents. These however require complex artificial intelligence (AI) functionality. Agents can realistically be of benefit in those areas concerned with autonomy and mobility. This is especially true of network management applications and this will be the focus of this paper. The paper discusses the usage of mobile agents and the advantages that these have over traditional client/server applications. It discusses the main characteristics of an agent, and shows how Java has the main components that allow mobile agents to be easily development. To show how agents are implemented it gives a practical implemented of an agent. Finally, the paper also discusses the main Java agent development systems, which are IBM aglets, Object Space Voyager and JATLite and outlines the advantages of using each of them.

1. Introduction

Agents are programs that automate user tasks. Their great advantage is that they are designed to run over a distributed computing system, whereas traditional utility programs typically run on a peer-to-peer type connection [1].

The requirement for agents increases for many reasons, such as:

- Increased requirement for management information for system administrators.
- Increased requirement for reliability for networked applications.
- Increased requirement for a certain quality of service from the network.
- Increased usage of main different types of computer systems, operating systems, networked operating systems, network technologies, and networking protocols.

An agent which can be made portable and mobile can be the answer to these increasing demands, particularly for network management applications.

Traditional client/server architectures are typically wasteful in their usage of bandwidth. Agent mobility overcomes this by minimising bandwidth consumption, as they support:

- Adaptive network load balancing.
- Solve problems caused by intermittent or unreliable network connections.

The main area that agents will bring benefits is for autonomy and mobility are required, such as in network management applications.

Most software development systems use object-oriented methods, and a mobile agent can be viewed at the next step in the evolution of the object-oriented (OO) paradigm, where an agent is also given a location. The work on mobile agents thus relates to object-oriented technology, as well as to networks and distributed computing systems.

The aim of this research is to adopt mobile agent architecture and to provide the framework on to build a variety of network management tasks, as well as investigating the advantages of mobile agent architecture over a client/server architecture by developing a number of small applications to automate common network management tasks.

Great gains will come from *intelligent* agents, which require complex artificial intelligence (AI) functionality.

2. Agents

Agents are commonly used in other technological areas, such as:

- **Artificial intelligence**. For improving collaborative on-line social environments.
- **Distributed systems**. Enhancements to the client/server architecture.
- **Electronic gophers**. They can 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.

Accurate definitions for agents are:

"An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors" [2].

"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"

"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." [3]

3 Agent properties

These definitions, whilst informative, appear to result from specific examples of agents, but are not truly representative as a global definition. The best definition is provided by Franklin and Graesser [4] who have classified agents in which agents posses several or all of the following characteristics (Figure 1):

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.

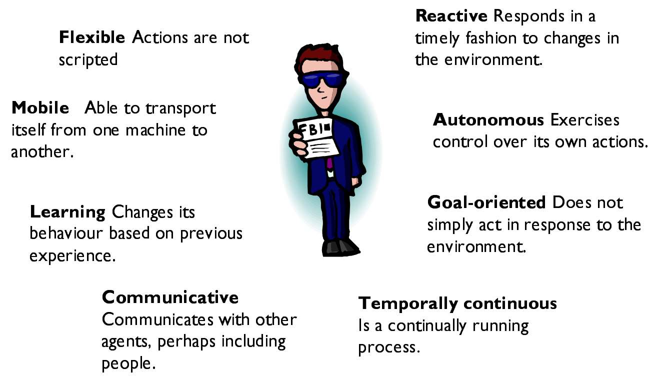
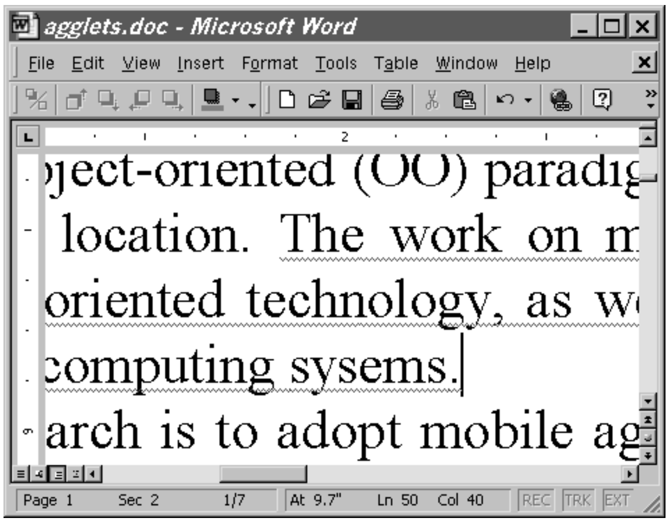

Flexible Actions are not scripted
Reactive Responds in a timely fashion to changes in the environment.
Mobile Able to transport itself from one machine to another.
Autonomous Exercises control over its own actions.
FBI
Learning Changes its behaviour based on previous experience.
Goal-oriented Does not simply act in response to the environment.
Communicative Communicates with other agents, perhaps including people.
Temporally continuous Is a continually running process.

Figure 1 Agent properties

Every agent satisfies the first four properties (reactive, autonomous, goal-oriented and temporally continuous). Other properties are defined on adding hierarchical classification. Agent research encapsulates main areas of computer science, such as objects and distributed object architectures, adaptive learning systems, artificial intelligence, expert systems, distributed processing and collaborative social online environments, as well as others.

4. Agent applications

Many software packages already use agents, such as Office Assistants in Office 97/2000 (Figure 2). These comply with the previous definitions of agents. For example, the spell checker agent 'watches' the user when typing and makes corrections on the fly. Agents such as this rely on AI for their functionality. However, one key property which these agents do not possess is *mobility*, and at present they are unable to operate in a distributed environment.


agglets.doc - Microsoft Word
File Edit View Insert Format Tools Table Window Help
Page 1 Sec 2 1/7 At 9.7" Ln 50 Col 40 REC TRK EXT

ject-oriented (OO) paradig
location. The work on m
oriented technology, as w
computing sysems.
arch is to adopt mobile ag

Figure 2 Office agents for spell checking

Mobility is a key property when considering the implication of mobile agent architectures against traditional client/server architectures. Client/server applications consist of the client and the server, usually on separate machines communicating over the network. When the client requires data or access to resources that the server provides, the client sends a request to the server. The server then responds by sending a response to the client. This handshake is continuous, where each request/response requires a complete round trip across the network. The fundamental difference in mobile agent architecture is in how this communication takes place; the client does not talk to the server over the network. The key difference is that the client actually migrates to the server's machine. Once on the server's machine, the client makes its requests of the server directly.

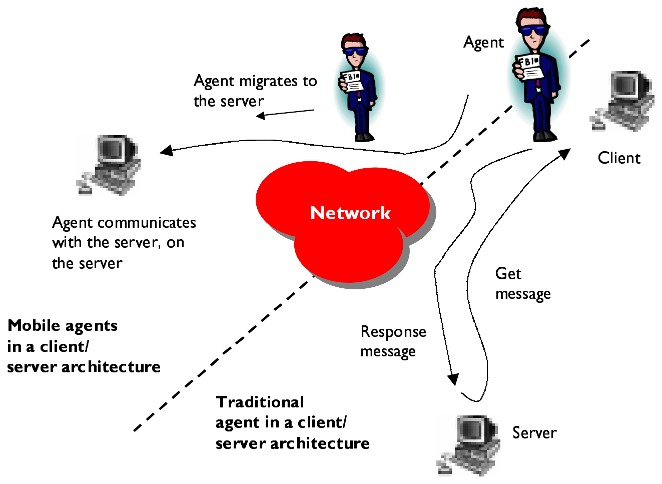
This represents a significant advantage over the client/server model and Sundsted [5, 6] states three reasons to adopt mobile agent architecture:

1. They reduce client/server network bandwidth problems. By moving a query or transaction from the client to the server, this eliminates repetitive request/response handshakes (see Figure 3).
2. They allow decisions about the location of code (client against server) to be made at the end of the development cycle when more is known about how the application will perform. This reduces the design risk.
3. They solve the problems created by intermittent or unreliable network connections. Agents can be easily created to work *off-line* and communicate their results when the application is back *on-line*.

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

1. Network bandwidth is a valuable resource. Client/server transactions use up network bandwidth. It is clearly an advantage to create an agent which handles queries, 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. Thus, instead of intermediate results and information being passed back and forth, only the agent needs be sent across the network.



Agent migrates to the server
Agent
Client
Network
Agent communicates with the server, on the server
Get message
Mobile agents in a client/ server architecture
Response message
Traditional agent in a client/ server architecture
Server

Figure 3 Mobile agents

2. In the design of traditional client/server applications, the programmer must have a clear idea about the roles of the client and server at design time, where 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, agent architecture gives a greater degree of flexibility, where an agent can be allowed to visit several nodes on the network in succession, carrying out some task. The only prerequisite is that at any node it visits, is that it provides the agent with an environment to operate. The means of communication, usually with message passing, is clearly defined and the programmer need only concentrate on the tasks to which a particular agent is assigned.
3. 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 of a transaction. On losing the network connection, 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. 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. Also, parallel execution (or load balancing) where a large computation can be divided dependent on resources. All these offer compelling reasons to adopt agent architecture for network management tasks.

Agents are also well adapted to heterogeneous environments, which are typical in today's networks.

5. Agents characteristics

Agents should support several characteristics. Sundsted (1998) identified these as:

- **Tactile**. Mobility and persistence.
- **Social**. Communication and collaboration and the
- **Cognitive**. Adaptation, learning and goal orientation. This has enormous potential and is the focus of much of the hype surrounding 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.

Realistically, the first two characteristics are currently more realistic goals. 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 WWW, and allows portable components to be distributed over a network. Such mobile code is dynamically loaded and executed by standalone programs, for applets this is the WWW browser. Unlike the applet however:

- An agent takes with it its state of execution.
- 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, the potential attack increase, especially with mobile code as a potential source of attack. If agent architecture 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 is be a convenient way to spread viruses and a security mechanism must be developed to handle both trusted code, which is safe, and untrusted code, which are not safe.

Every WWW browser implements security policies to keep applets from compromising system security. The following restrictions apply:

- An applet cannot read from or write to 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 security measures are very necessary, but inhibiting. When considering 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. Java provides a highly customisable security model, which allows us to go some way in achieving this aim. Thus, just as the applet requires a WWW browser in which to execute, so an agent needs a safe environment in which to be hosted.

As a minimum, any agent operating within its host must have a:

- Unique identity.
- Means of identifying that other agents are also operating within the host.
- 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 each other 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' it so as to allow its execution to continue after transfer.
- Inhibit agents from directly interfering with each other.

As well as giving the agent mobility, the agent must also be given a workplace (a context), where it gets access to only the resources and data that it requires. The context is: a stationary object residing on the host computer. It:

- Provides a means for maintaining and running agents in a uniform execution environment, securing the host against possible malicious agent attacks.
- Prevents 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 straightforward to build adequate security models to protect against malicious agents, it is not as easy to protect agents 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, as, a host could upload an agents class files and state of execution, and then have access to such sensitive information. The agent's context must then include some mechanism where rules can be set stating what privileges that an agent has within its context. These are likely to be based on knowledge of the agent's origin.

Agents interact with each other and applications through message passing. Message handling must be able to support both synchronous messages and asynchronous messages. The two types of messages are:

- **Now-type message**. This is synchronous and blocks the execution until the receiver has completed the handling of the message.
- **Future-type message**. This 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.
- **Multicast**. This is where a message is simultaneously sent to all agents within a context. Only those agents who have subscribed to the message will receive it.

6. Why Java?

Mobile agents can use one of two fundamental development technologies: *mobile code* or *remote objects*. Java is well suited to these environments are it supports both of these through Object Serialization Remote Method Invocation (RMI) [7].

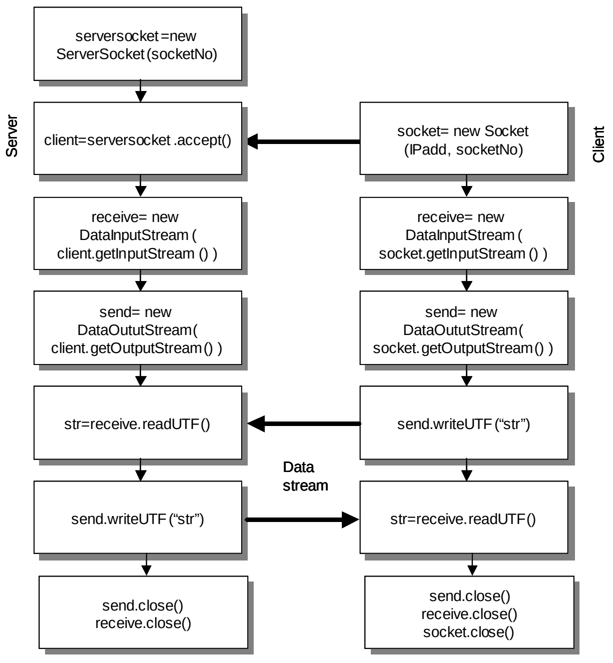
Java has become the development language of choice for building distributed application components. It offers:

- Modular, dynamic, class-orientated compilation units.
- 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.

The main sections of code required for a Java agent are:

- **Sockets**. In Java, 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. 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.
- **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 which 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-coordinating threads relatively simple. The main advantages of threads are:
    - o They make better use of the processor, where different threads can be run when one or more threads are waiting for data. For example, a thread

could be waiting for keyboard input, while another thread could be reading data from the disk.
- They are easier to test, as each thread can be tested independently of other threads.
- They can use standard threads, which are optimised for given hardware.



serversocket=new ServerSocket(socketNo)
Server
client=serversocket.accept()
socket= new Socket (IPadd, socketNo)
Client
receive= new DataInputStream (client.getInputStream ())
receive= new DataInputStream (socket.getInputStream ())
send= new DataOututStream(client.getOutputStream())
send= new DataOututStream(socket.getOutputStream())
str=receive.readUTF()
send.writeUTF ("str")
Data stream
send.writeUTF ("str")
str=receive.readUTF()
send.close()
receive.close()
send.close()
receive.close()
socket.close()

Figure 4 Java sockets

- **RMI**. RMI supports the information interchange between the server and client. It uses a distributed object application, where Java objects may be accessed and their methods called remotely to take advantage of a distributed environment and thus spread a workload over a number of network nodes. It also provides a means in which agents may communicate each another.

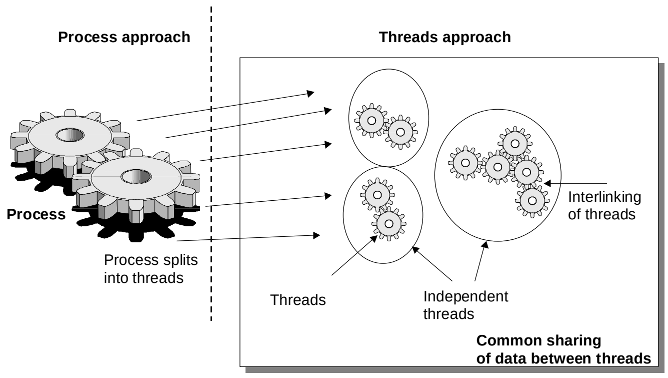


Figure 5 Process splitting into threads

- **Object Serialization**. This is a process which enables the reading and writing of objects, and has many uses, such as RMI and object persistence. In developing agent applications it is Serialization that can provides *mobility*. An object (an agent) may be serialised (converted to a bit stream), and moved (passed over the socket) to another host where it continues its execution. Thus, the agent is no longer bound to the host, but has the whole network as a resource. 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). Reading from and writing to a socket across a network is as easy as reading and writing any standard I/O stream. Before 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 thus supports sockets, threads, RMI and object serialization, and is thus the ideal development environment for mobile agents, and any distributed application. In summary, Java also provides:

- Modular, dynamic, class-orientated compilation units.
- Portability and mobility of compiled code (class files).
- On-demand loading of functionality.
- Built-in support for low-level network programming.

One important area is security. Agent architecture ultimately allows agents to move around a network, where the boundaries of hosts become blurred and access to local resources is available, security becomes a major concern. It could be easy to write agents which could corrupt the data on the systems connected to a network. Java offers a fine-grained and highly configurable security control that provides acceptable levels of security to protect hosts from malicious agents. A more difficult question, which is not so straightforward to address, is that of malicious hosts. As hosts upload an agent's class files and state of execution, it could potentially have access to sensitive information. Agent architecture must then include an environment in which agents can operate; known as a context. Rules can be adopted on what privileges an agent may be granted within this context based on knowledge of its origin. These rules are of considerable interest, and are currently being investigated as part of this research.

The original security model provided by Java (the *sandbox* model) allowed a very restricted environment in which to run untrusted code obtained from the open network. In this, local code is trusted to have full access to vital system resources, whereas 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. Security has changed over the versions of Java with:

- **Signed applet**. JDK 1.1 introduced this, where a digitally signed applet is treated like local code, with full access to resources, only if the public key used to verify the signature is trusted. Unsigned applets are still run in the sandbox.
- **Security policy**. JDK 1.2 introduced this in an attempt to integrate all aspects of security into a consistent

approach. All code, regardless of whether it is local or remote, can now be subject to a security *policy*. This 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 mobile agent architecture.

7. Agent development tool

Java is an excellent choice for a development language, but it is not so easy to determine an environment in which to develop mobile agents. It would of course be possible to develop from scratch a suitable environment, as Java supports all the necessary requirements, but this would be a massive undertaking. Instead, it is necessary to examine what is already available. There are a number of agent development environments [8], such as:

- **JATLite** [9]. This allows the development of agents that exchange messages and agent router functionality. Agent routing 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. This provides a robust message passing system, in which agents communicate through passing messages. It does not however benefit from allowing agent mobility.
- **Object Space Voyager** [10]. This provides a system which takes existing Java class and *treat* them it to create a new class with an identical interface, known as a proxy. The proxy is an image of another class and operates just as the original to 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, and its strength mainly lies in the integration and support of other distributed technologies, such as CORBA.
- **IBM Aglets** [11]. This allows for the development of its mobile agent, called aglets, which are Java objects which can move from one host to another, over a network. When the aglet moves, it takes the program code as well as all the objects it is carrying.

Distributed applications can be classified into two groups: those where applications are partitioned among participating nodes and the other computation is partitioned towards resources. JATLIte and Object Space Voyager use the first type, whereas aglets migrate computation toward resources. This mobility offers an evolution of the OO paradigm where an object can be given a location and for this reason the Aglet Workbench has been chosen as the best choice of development system.

The aglet framework provides useful generic aglet pairs from which development of network management tasks may evolve. These are:

- Messenger – receiver.
- Master – slave.
- Notifier – notification.

Using the aglet framework is a convenient way for user-defined agent to inherit default properties and functions for mobile agents.

8. Practical agent implementation

Rather than having a client/server type communication, an aglet (a mobile agent) is deployed which actually travels to the server, from the client and carries out some action on the machine where the server resides.

The infoserver class handles multiple requests from any client. It responds to a PUT or GET message by collecting local information sent by the client, or by sending all the information gathered so far.

```
import java.net.* ;
import java.io.*;

import java.util.Vector;
import java.util.Enumeration;

class infoClient {
   private Socket theclient=null ;
   private InetAddress host ;
   //private PrintWriter out ;
   //private BufferedReader in ;

   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();
   }
}
```

The infoClient class is the client side of the application. When the user issues a PUT command the local info is obtained and sent to the server as a LocalInfo object. A GET command receives an array (a Java vector) of LocalInfo objects gathered so far.

```
import java.net.* ;
import java.io.* ;

import java.util.Vector;
import java.util.Enumeration;

class infoClient {
   private  Socket theclient=null ;
   private  InetAddress host ;
   private  PrintWriter out ;
   private  BufferedReader in ;

   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) ;
      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();
   }
}
```

The LocalInfo class handles the information local to a host.

```
package myAglets;

import java.util.Date;
import java.io.*;
import java.net.*;
import java.lang.System ;

class LocalInfo implements Serializable {

   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;
 }
}
```

9 Mobile agent development tool

The Aglets Software Development Kit (ASDK) from IBM's Tokyo Research Laboratory is a mobile agent framework written in pure Java that allows the development of distributed applications using 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)

Aglets derive their name from a combination of agent and applet. Applets are event-driven and provide methods that the programmers may override in order to control their life cycle, whereas, 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:

`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.

Aglets communicate by message passing. 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.

The agent host supplied in the ASDK is called Tahiti. It offers a graphical user interface in which to run aglets and provides a customisable security interface configurable at each aglet host. 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. Nonetheless, it provides a highly customisable security environment. Within each host, the aglet is given a unique identity based on its class name and code base. The owner of the aglet may also be identified by his/her name and e-mail address. Aglets are further categorised as *trusted*, where their code base is local or *untrusted* where the code base originated elsewhere. Security options are defined for each category (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. R*eserves ports on which network connections may be made
- *Properties*. Defines which system properties are made available

10. Conclusions

This paper has outlined the advantages that mobile agents have over traditional client/server applications, and has given Java code examples for the implementation of agents. The advantages of agents include a reduction in the network communications, especially in transmitting network language information.

The paper has clearly shown that Java is the best choice for a programming language as it directly supports sockets, RMI, threads and object serialization. The paper has also outlined the main Java agent development system has gives the advantages of using each of them. It concludes that the IBM aglet system has the advantage over other types in that aglets migrate computation toward resources, whereas other methods partitioned applications among participating nodes. This mobility offers an evolution of the OO paradigm where an object can be given a location.

11. References

[1] Watson, Mark. (1997). *Intelligent Java Applications for the Internet and Intranets*. Morgan Kauffman Publishers.

[2] Russell, Stuart. J. and Norvig, Peter (1995). *Artificial Intelligence: A Modern Approach*. Eaglewood Cliffs, NJ: Prentice Hall, pp 33.

[3] Chang, IBM (1998). [http://activist.gpl.ibm.com:81/WhitePaper/e2.html]

[4] Franklin, Stan and Graesser, Art (1996). *Is it an Agent, or just a Program? A taxonomy for Autonomous Agents*. Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag. [http://www.mcsi.memphis.edu/~franklin/AgentProg.html]

[5] Sunsted, Todd. *Agents on the move*, Java World, July 1998". [http://www.javaworld.com/javaworld/jw-07-1998/jw-07-howto.html]

[6] Sunsted, Todd. *An Introduction to agents – Java World* – June 1998.

[7] Niemeyer, P and Peck, J (1996). *Exploring Java*. O'Reilly and Associates, pp 235.

[8] www.cs.nccu.edu.tw/~jong/agent/IA/ma.html

[9] JATLite [htt p://java.stanforrd.edu]

[10] ObjectSpace: Voyager Overview [http://www. objectspace.com/products/vgrOverview.htm]

[11] IBM Aglets SDK (1998). [http://www.trl.ibm.co.jp/aglets/]

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

[13] Kautz, Selmen and Coen (1994). *Bottom–up Design of Software Agents*. Communications of the ACM, 37, 7.

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

[15] Aglets - Mobile Java™ Agents [http://www.trl.ibm.co.jp/aglets/whitepaper.html]

[16] Cetus-Links (links to agent resources) [http://pent21.infosys.tuwien.ac.at/cetus/oo_mobile_agents.html]