DYNAMIC DETECTION AND IMMUNISATION OF MAL-WARE USING MOBILE AGENTS

HUSSAIN ALI AL SEBEA

Submitted in partial fulfilment of the requirements of
Napier University for the degree of
MSc Networking and Web Technology

School of Computing

June 2005

Authorship Declaration

I, Hussain Al Sebea, 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: 01008950

Abstract

At present, malicious software (mal-ware) is causing many problems on private networks and the Internet. One major cause of this includes outdated or absent security software to countermeasure these anomalies such as Antivirus software and Personal Firewalls. Another cause is that mal-ware can exploit weaknesses in software, notably operating systems. This can be reduced by use of a patch service, which automatically downloads patches to its clients. Unfortunately this can lead to new problems introduced by the patch server itself.

The aim of this project is to produce a more flexible approach in which agent programs are dispatched to clients (which in turn run static agent programs), allowing them to communicate locally rather than over the network. Thus, this project uses mobile agents which are software agents which can be given an *itinerary* and migrate to different hosts, interrogating the static agents therein for any suspicious files. These mobile agents are deployed with a list of known mal-ware signatures and their corresponding cures, which are used as a reference to determine whether a reported suspect is indeed malicious. The overall system is responsible for Dynamic Detection and Immunisation of Mal-ware using Mobile Agents (DIMA) on peer to peer (P2P) systems. DIMA is be categorised under Intrusion Detection Systems (IDS) and deals with the specific branch of malicious software discovery and removal.

DIMA was designed using Borland Delphi to implement the static agent due to its seamless integration with the Windows operating system, whereas the mobile agent was implemented in Java, running on the Grasshopper mobile agent environment, due to its compliance with several mobile agent development standards and in-depth documentation.

In order to evaluate the characteristics of the DIMA system a number of experiments were carried out. This included measuring the total migration time and host hardware specification and its effect on trip timings. Also, as the mobile agent migrated, its size was measured between hops to see how this varied as more data was collected from hosts.

The main results of this project show that the time the mobile agent took to visit all predetermined hosts increased linearly as the number of hosts grew (the average inter-hop interval was approximately 1 second). It was also noted that modifications to hardware specifications in a group of hosts had minimal effect on the total journey time for the mobile agent. Increasing a group of host's processor speeds or RAM capacity made a subtle difference to round trip timings (less than 300 milliseconds faster than a slower group of hosts). Finally, it was proven that as the agent made more hops, it increased in size due to the accumulation of statistical data collected (57 bytes after the first hop, and then a constant increase of 4 bytes per hop thereafter).

Table of Contents

Authorship Declaration 2
Abstract 3
Table of Contents 4
List of Tables 7
List of Figures 8
List of Figures 8
Acknowledgements 9

1. INTRODUCTION 10

1.1 Aims and Objectives 10

1.2 Background 11

2 THEORY 12

2.1 Introduction 12

2.2 What is an agent? 12

2.3 Mobile Agents 13
2.3.1 Characteristics 13
2.3.2 Framework 14

2.4 Mobile Agent Environments 14

2.5 Client Server Communication 15
2.5.1 Delphi Sockets 16
Sample Client/Server Application in Delphi 17
2.5.2 Java Sockets 19
Java Client 20

3 LITERATURE REVIEW 23

3.1 Introduction 23

3.2 Similar Research Directions 23
3.2.1 Definition of Intrusion Detection 23
3.2.2 First Generation IDSs 23
3.2.3 Present Day IDSs 24
3.2.4 Why Use Mobile Agents? 25
3.2.5 Agent Architecture 26
3.2.6 Using Mobile Agents for Intrusion Detection 26
3.2.7 Related Work 27

3.3 Summary 29

4 REQUIREMENTS ANALYSIS AND DESIGN 31

4.1 Introduction 31

4.2 System Component Definition: Overall System Requirements 31

4.3 FileMon Requirements and Design 32
4.3.1 Introduction 32
4.3.2 Static Agent Requirements 33
Main Requirements 33
Programming Requirements 33
4.3.3 Static Agent Design 33
File Modification Monitor 33
Network Communication 34
Agent-to-Agent Protocol 35
Local Administrative Functions 36
Remedy Application 36
User Interface Design 38

4.4 MalDes Requirements and Design 40
4.4.1 Introduction 40
4.4.2 Mobile Agent Requirements 40
Main Requirements 40
Programming Requirements 40
4.4.3 Mobile Agent Design 40
Multi-host Migration Ability 40
Active Intelligence 41
Protocol Compliance 41
System Status Report Generation 41
Failsafe Design 42

4.5 Analytical Experiment Design 44
4.5.1 Aims of Experiments 44
4.5.2 Experiment Construction 44
Return Trip Timer 44
Host Performance Timer 44
Agent Size Measurement 44
Internet Return Trip Timer 45

4.6 Summary 45

5 IMPLEMENTATION 47

5.1 Introduction 47

5.2 Grasshopper Programming Briefing 47

5.3 Implementation Details: FileMon 48
5.3.1 File Modification Monitor 48
5.3.2 Network Communication 49
5.3.3 Protocol Integration 51
Protocol Processor 52
Command: ANY SUSPECTS 53
Command: NUMBER OF SUSPECTS 53
Command: LIST SUSPECTS or NEXT SUSPECT 53
Command: REMEDY 53
Command: NO REMEDY FOUND 53
5.3.4 Local Administrative Functions 54
Process Termination 54
File Deletion 54
Registry Value Removal 55
5.3.5 Remedy Application 55

5.4 Implementation Details: Mobile Agent 55
5.4.1 Built-in Intelligence 55
5.4.2 Itinerary Preparation 56
5.4.3 Network Connectivity 56
5.4.4 Protocol Integration 56
Command Templates 56
Protocol Processing 56
Command: NEGATIVE 56
Command: POSITIVE 56
Command: RESPONSE NUMBER *X* 57
Command: SUSPECT *filename* 57
Command: REMEDY RECEIVED 57
Command: QUIT 57
5.4.5 Agent Lifecycle States 57

5.5 Analytical Experiment Implementation 58
5.5.1 Experiment 1: Return Trip Timer 58
5.5.2 Experiment 2: Host Performance Timer 59
5.5.3 Experiment 3: Agent Size Measurement 59
5.5.4 Experiment 4: Internet RTT Test 60

5.6 Summary 60

6 RESULTS 62

6.1 Introduction 62

6.2 Return Trip Timer (RTT) 62

6.3 Host Performance Timer 62

6.4 Agent Size Measurement 64

7 EVALUATION 66

7.1 Evaluation of Objectives 66

7.2 Main Findings 67

7.3 Conclusions 67

7.4 Recommendations for Future Work 68

8 APPENDICES 70

8.1 Appendix 1: Project Management (GANTT Chart) 70

8.2 Appendix 2: Project Management (Project Diaries) 71

8.3 Appendix 3: Source Code (MalDes) 80

8.4 Appendix 4: Code Extracts (FileMon – Remedy Application) 89

8.5 Appendix 5: ECIW Research Paper 91

9 REFERENCES 99

List of Tables

TABLE 4.1 A TYPICAL INTER-AGENT CONVERSATION 35
TABLE 4.2 RETURN TRIP TIMER TESTS 44
TABLE 5.1: PROTOCOL COMMANDS 52
TABLE 6.1: RTT TEST RESULTS FOR 1, 2, 3 AND 4 HOSTS 62
TABLE 6.2 SUPERIORS RTT TEST RESULTS 63
TABLE 6.3 INFERIORS RTT TEST RESULTS 63
TABLE 6.4 CONVERSATION SIZE RESULTS AS HOPS INCREASE 64

List of Figures

FIGURE 2.1: SERVER PROGRAM GUI 17
FIGURE 2.2: CLIENT PROGRAM GUI 18
FIGURE 2.3: TEXT RECEIPT GUI 19
FIGURE 4.1: CONNECTION SUMMARY BETWEEN FILEMON AND MALDES 32
FIGURE 4.2 REMEDY PACKAGE STRUCTURE 37
FIGURE 4.3 STATIC AGENT ROLES 38
FIGURE 4.4: FILEMON SCREENSHOT 38
FIGURE 4.5 SETTINGS WINDOW SCREENSHOT 39
FIGURE 4.6 MOBILE AGENT ROLES 43
FIGURE 4.7 AGENT INTERACTION 46
FIGURE 5.1 SAMPLE NETWORK SERVER 1 50
FIGURE 5.2: SAMPLE NETWORK SERVER 2 IENT. 50
FIGURE 6.1: RTT HOST PERFORMANCE TEST 63
FIGURE 6.2: CONVERSATION SIZE CHART 65

Acknowledgements

I would like to express my thanks and gratitude to my parents who supported me while I was studying abroad. Also many thanks go to Bill Buchanan and Nikos Migas who provided the motivation this thesis required to lift off the ground.

1. Introduction

Mobile agents have been in the limelight for some time now. Their use in sophisticated applications provides powerful advantages that can be employed to devise flexible and autonomous distributed systems. Agents possess distinguishing characteristics which differ from traditional software systems. Often these software systems have many aims which makes the overall system complex and, typically, the more complex a software entity becomes, the more prone it is to errors. A software agent has the advantage that it is typically goal-oriented, and can thus be simpler to write, and easier to test. Classed as mobile, these agents hold the capability to independently traverse the network they inhabit, carrying out software tasks they have been assigned. It is believed that network awareness of MAs could contribute to overcoming the contemporary issue of detecting and responding to intrusions in the network. Some of the design elements from this project contributed to the writing and acceptance of a paper accepted in the ECIW 2005 conference (Buchanan et al., 2005). This chapter's purpose is to introduce the project, state its aims and objectives and presents scope and background information. Having defined the research question of *whether IDSs for the detection of mal-ware can be enhanced with the usage of an agent-based approach in P2P systems*, Chapter 2 moves on to define all the background information required to be able to understand the rest of the dissertation. Topics covered include agents, mobile agents, and mobile agent environments. Chapter 3 moves on to state the advantages of using mobile agents, followed by a critical review of achieved work in the field of agent based IDS. Chapter 4 elucidates the design phase of the project by providing analysis and specification of the system proposed, followed by Chapter 5 that details the steps taken to implement a prototype and design the relevant experiments required for evaluation. The experiments are carried out in Chapter 6 and their results are displayed in an explanatory fashion. Finally, Chapter 7 attempts to evaluate project objective achievement, critically assess the work done, and highlight scope for future work.

1.1 Aims and Objectives

1. To conduct a critical review of the literature in the field of using mobile agents in IDSs, including the hierarchical and also the peer-to-peer (P2P) approach.
2. To design and implement a framework for mal-ware detection and immunisation using mobile and static agents.
3. To evaluate the performance of a mobile agent approach and propose strengths and weaknesses of this approach as compared with traditional mal-ware detection/immunisation systems.
4. To document findings and to critically analyse the project and examine whether the advantages of adopting the P2P architecture over the centrally co-ordinated version are indeed worthwhile.

1.2 Background

First generation IDSs consisted of data collection components and a centralised analyser. Data collected from different sources was sent to the central analyser, which took the appropriate action. In these types of IDSs, the central point can be a *point* of failure. Presently used IDSs are hierarchically structured. Processed data from host and network levels are passed up the ladder until they reach the central coordinator, which in turn conducts analysis to determine the health of the overall system. These systems lack flexibility and making changes to the way things work is difficult (Ramachandran & Hart, 2004).

Ramachandran and Hart (2004) have proposed a *distributed* mobile agent based IDS. Their system overlooks the need for a central hub, can be adapted easily and additionally scales well to large networks. The aim of this project is to further research and implement certain aspects of their present work (which was still at its initial stages as of April 2004) by designing and implementing a set of agents to perform specific system file monitoring tasks and inject them into a "virtual network laboratory." These agents keep records of critical system file data at strategic nodes in the network. Once these agents are set up, a set of pre-defined intrusion hazards will be simulated to provoke these agents. The agents should be able to identify the any present hazards and eliminate them. Performance and output results from each of the agents will be logged and findings shall be reported through comparative analysis.

2 Theory

2.1 Introduction

This chapter attempts to define all necessary background information required to be able to comprehend the sections that are to follow. Agents in general and mobile agents are introduced, followed by client and server communications. Since the project involves client/server programming, two code samples in two eligible languages have been shown and clarified (Java and Delphi). This chapter is concluded with mobile agent platforms.

2.2 What is an agent?

"An agent is a program that assists people and acts on their behalf. Agents function by allowing people to delegate work to them" (Lange & Oshima, 2003). A software agent can be defined as:

> A software entity which functions continuously and autonomously in a particular environment ... able to carry out activities in a flexible and intelligent manner that is responsive to changes in the environment ... Ideally, an agent that functions continuously ... would be able to learn from its experience. In addition, we expect an agent that inhabits an environment with other agents and processes to be able to communicate and cooperate with them...
>
> (Bradshaw, 1997, cited by Balasubramaniyan, Garcia-Fernandez, Isacoff, Spafford& Zamboni, 1998)

Jansen (2001) further clarifies the definition of an agent as a piece of software that can exercise authority on behalf of an entity (be it a person or organisation) and autonomously progress to achieve a predefined goal or set of goals. An agent may also - at its own will – interact with other agents and also with its own environment. Helmer, Wong, Honavar, Miller and Wang (2002) describe 'lightweight' agents as those that achieve their tasks with minimal code. These agents are dynamically upgradeable, less complex, smaller, and hence faster to transport. According to Tanenbaum and Van Steen (2002) there has been a lot of controversy concerning what exactly an agent is. The definition given by Lange and Oshima is quite broad, and many different types of applications can be easily categorised as agents. Balasubramaniyan et al.'s explanation is somewhat short from being comprehensive and is rather an attempt to illustrate the specific examples of agents that the group had in mind. Jansen attempts to build on the former two but falls short of providing a thorough picture. Instead of narrowing down the definition (as Helmer et al. had attempted), it makes more sense to describe the characteristics of the category of agent that are of interest.

The list below shows these characteristics, some of which are common to all agents:

- *autonomous* can act on its own
- *reactive* responds timely to changes in its environment
- *proactive* initiates actions that affect its environment

- *communicative* can exchange information with users and other agents
- *continuous* has a relatively long life span
- *mobile* can migrate from one site to another
- *adaptive* capable of learning

The latter three are not common to all agents (Tanenbaum & Van Steen, 2002).

2.3 Mobile Agents

2.3.1 Characteristics

Mobile agents (also called migrating processes) are derived from *standard* agents and have the ability to physically travel across a network and perform tasks on machines that provide agent hosting services. This allows processes to migrate across computers and allows for splitting processes into multiple instances that execute on different machines, and to return to their point of origin, the *launch pad*. Unlike remote procedure calls (RPC), where a process invokes procedures hosted at a remote machine, process migration allows executable code to travel and interact with databases, file systems, information services and other agents (Reilly, 1998). Reillyalso mentions that the following things are prerequisites that enable mobile agent migration:

- *Establish a common execution language*. In a situation where an agent was to migrate from one machine to the next, it must be ensured that both machines share a common execution language. This can be guaranteed only in a homogenous networking environment. However, such a system would not prove scalable. In the real world, networking environments are heterogeneous, where systems of many different architectures are interconnected. In this case, meeting the common execution language requisite would obviously be impossible. The most suitable approach to this obstacle would be the use of an interpreted scripting language, via a mobile agent environment such as Grasshopper.
- *Ensure process persistence*. Following successful migration to a remote machine, a process would need to be provided with its runtime-relevant information in order to resume execution. Usually, the agent execution environment would be responsible for this, by saving the process execution state and packing it along with the agent before dispatch.
- *Setup a communication mechanism between agent hosts*. Some mechanism for communication must be established to allow agent transferral across networks. TCP/IP is a commonly used protocol to transfer processes. An alternative higher level of communication is Remote method Invocation (RMI), which according to Shekhar (2003) provides a simple and direct model for distributed computation with Java objects.
- *Enforce security to protect agents and agent hosts*. In situations where executable code is allowed to travel across a network, badly written code or misbehaving processes could wreak havoc amongst hosts if not hindered by preventative measures. Similarly, well-behaved processes could be exploited by their hosts which could cripple the agent's functionality.

2.3.2 Framework

Referring to the previously mentioned characteristics, presented next (Code Extract 2.1) is the basic code framework of a mobile agent which migrates to a single host (in this case to 192.168.0.1 on port 7000) and returns back to its origin.

The source code shown in Code Extract 2.1 is represented by a Java class that may be executed on a suitable mobile agent environment such as Grasshopper.

```
package examples.simple;
import de.ikv.grasshopper.communication.GrasshopperAddress;
public class GrainHopper extends de.ikv.grasshopper.agent.MobileAgent {
       int state;

       public void init(Object[] creationArgs) { state = 0; }
       public void live()
         {
        switch(state)
       {
        case 0:

          GrasshopperAddress location =
              new GrasshopperAddress( "socket://192.168.0.1:7000" );
          if (location != null)
          {
          state = 1;
          log( "Trying to move..." );
          try {
             // Migrate!
              move(location);
              }
          catch (Exception e) { log( "Migration failed: " , e); }
          state = 0;
          }

        break;

        case 1:
          log( "Arrived at destination, heading back home now." );
          state = 2;
          log( "Trying to move..." );
          try
          {
          move(getInfo().getHome());
            }
          catch (Exception e) { log( "Return trip failed: " , e); }
       break;
       }
  }
}
```

Code Extract 2.1: Sample Agent Code
Basic code framework of a mobile agent which migrates to a single host.

2.4 Mobile Agent Environments

With reference to the preceding Section 2.3, agent execution environments are responsible for maintaining process persistence as agents migrate from host to host. Their responsibilities also include handling inter-agent communication, agent transport and a agent proximity detection service, which keeps track of agent locations as they travel. Interactions can be performed using plain socket connections (TCP/IP) or Java RMI as was mentioned earlier.

On some environments, RMI and plain socket connections can be protected with encryption. Mobile agent environments are able to interact with each other using a range of methods such as synchronous, asynchronous and multicast communication (IKV++ GmbH [A], 1999). A few examples of agent environments include Grasshopper and IBM Aglets SDK.

2.5 Client Server Communication

Several benefits distinguish TCP/IP over other networking protocols. Being an open standard that is widely published, TCP/IP is totally independent of any hardware or software manufacturer. Belonging to the transport and network layers of the ISO OSI seven layer model, this protocol can be run on many types of network including Ethernet, Token Ring, and telephone lines. This allows sending of data between systems that may differ in specifications, size and operating system. The IP portion of TCP/IP makes the protocol routable, balancing load on different parts of the network. Courtesy of IP addressing, TCP/IP is scalable, allowing it to be run on networks of varying sizes, even those as large as the Internet. When computers communicate on a TCP/IP network, network data travels from a port on the sending computer to a port on the receiving computer. Using ports is required to identify the application that is associated with the data and is represented by a unique 16-bit number in the range of 0 through 65535. The source port represents the application that sends the data. Likewise, the destination port represents the application that receives the data. Many of these ports are standardised and well known. Thus, during actual communication, computers automatically determine which port to connect to for a specific service. The automation of this procedure results in ports being utilised transparently to the user (Groth, 2001).

In a client/server distributed system, processes are divided into two groups. A *server* is a process that implements a specific service, for example, a database service. A *client* is a process that requests a service from a server by sending a request (in an agreed upon manner) and subsequently waiting for the server's reply. There are two methods of communication between a client and a server. A simple connectionless protocol can be implemented if the underlying network is considered reliable (as in many local area networks). In this scenario, when a client requests a service, it simply constructs a message for the server, identifying a known service it wants along with required data. The message is then dispatched to the server. The server waits for an incoming request, processes it, and constructs a reply message that is sent back to the client.
Using a connectionless protocol boasts an advantage of being efficient (less communication to achieve a required goal). Provided that messages arrive correctly (and in a timely fashion), this method works very well. There are however setbacks to using this approach. Transmission failures can come in the way and providing a workaround for the protocol is not easy. If an acknowledgement reply was lost, then resending the request may result in the server performing the operation more times than needed. An example of this would be a request to deduct a sum of money from the client's bank account. In this case, sending a duplicate request would be catastrophic. Transmission failures cannot be detected by the client and the remedy would need to involve extra programming effort. Before making a choice on which communication method to use in this project, the second option shall be explored.

Alternatively, many client/server systems use a reliable connection-oriented protocol. Although not entirely relevant in local-area networks due to their nature, the connection-oriented protocol works very well in larger network infrastructures (where communication can be unreliable). This communication method requires that whenever a client requests a service, it must first set up a connection to the server, before sending its request. The server usually uses the same connection to send the reply message. After the server replies, the connection is torn down. Setting up and tearing down a connection can be costly, especially if the data to be transmitted is small in size.

Conceptually, a *socket* is a communication endpoint to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read. A socket is actually an abstraction for a specific transport protocol (Tanenbaum & Van Steen, 2002). The connection-oriented approach shall be used as a base for this project, bearing in mind that the client and server may reside on the same physical machine. The costliness of connection set up will not be considered a setback since this project is to provide a prototype application and will not involve much communication. If the project were to be practically applied, the connectionless method would be ideal since client/server conversations will not exceed one request and one reply.

2.5.1 Delphi Sockets

Referring to Gajic (2005), Delphi provides several components that allow data exchange over the network. Two quite common objects are TServerSocket and TClientSocket. Both are designed to let applications read and write data over a connection-oriented channel (TCP/IP).

Microsoft Windows provides an open interface for network programming, called *Winsock*. Winsock acts as a link between Delphi network applications and underlying protocol stacks by providing necessary classes and events required to access network services.

Delphi includes socket components, which are wrappers for the aforementioned Winsock classes that let applications communicate with each other using TCP/IP, without the developer having to worry about underlying network software or protocols.

Initiating a socket connection using the socket component requires a host and a port to be specified. Generally, host specifies an alias for the IP address of the server system; port specifies the endpoint number that identifies the server socket connection.

Sample Client/Server Application in Delphi

Delphi Server

To create a network server program, the *TServerSocket* component is required. The port property must be specified and the component must be ordered to listen for incoming connections as shown in Figure 2.1.

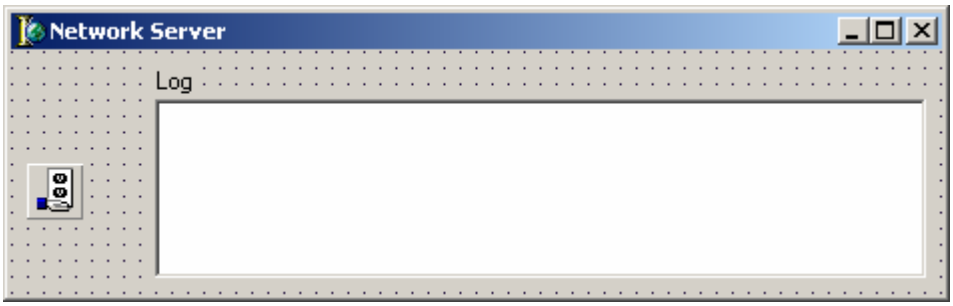


Figure 2.1: Server Program GUI
GUI for the server program, showing the TServerSocket component and a memo control to display received text.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
        ServerSocket1.Port := 110; // Port reserved for POP3
        ServerSocket1.Active := True;
end;
```

Code Extract 2.2: TServerSocket Settings
TServerSocket must be programmed to listen to incoming connections on a specific port, for example, 110 (POP3). This is done in the program's OnCreate event.

The *Active* method shown above orders the TServerSocket component to start listening. Once this program is compiled and executed, it will actively listen for incoming connection requests on port 110.

The listening process must be ended upon termination of the program. This is implemented in the program's *OnClose* event (which actually belongs to the main form).

```
procedure TForm1.FormClose
(Sender: TObject; var Action: TCloseAction);
begin
        ServerSocket1.Active := false;
end;
```

Code Extract 2.3: Terminating TServerSocket
Terminating the socket when the program exits.

Delphi Client

The client program is created in the same way, except using the *TClientSocket* component, as shown in Figure 2.2 and Code Extract 2.4.

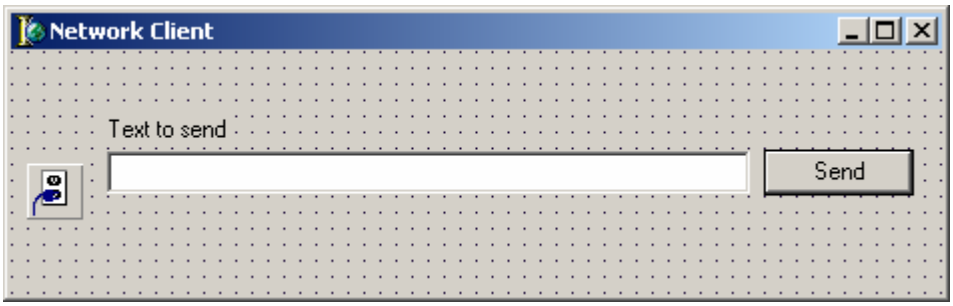


Figure 2.2: Client Program GUI
GUI for the Delphi client program, showing the TClientSocket component and an edit box along with a button to send text to the server.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ClientSocket1.Port :=  10;
  // Host to connect to is the server's IP address in this case.
  ClientSocket1.Host :=  '192.168.0.1';
  ClientSocket1.Active :=  true;
end;

procedure TForm1.FormClose
(Sender: TObject; var Action: TCloseAction);
begin
  ClientSocket1.Active := false;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if ClientSocket1.Active then
    ClientSocket1.Socket.SendText(Edit1.Text);
end;
```

Code Extract 2.4: Client Socket Settings
The client program must be programmed to connect to the server (at its IP address) on the predefined port upon start-up and to send the specified text when the Send button is clicked (Button1.OnClick). The socket must be closed when the program is terminated

A slight addition to the server program must be made in order for it to show the text received by clients. This is shown in Code Extract 2.5.

```
procedure TForm1.ServerSocket1ClientRead
(Sender: TObject; Socket: TCustomWinSocket);
var
  i:integer;
  sRec : string;
begin
  for i := 0 to ServerSocket1.Socket.ActiveConnections-1 do
  begin
    with ServerSocket1.Socket.Connections[i] do
    begin
      sRec := ReceiveText;
      if sRecr <> '' then
      begin
        Memo1.Lines.Add(RemoteAddress + ' sends :') ;
        Memo1.Lines.Add(sRecr);
      end;
    end;
  end;
end;
```

Code Extract 2.5: Message Exchange Using Sockets
This makes the server iterate through its list of open connections and add any text received to the memo control, as a means of feedback to the user.

After this modification, the server will log any text received from clients. The GUI below shows the final program's functionality (Figure 2.3).

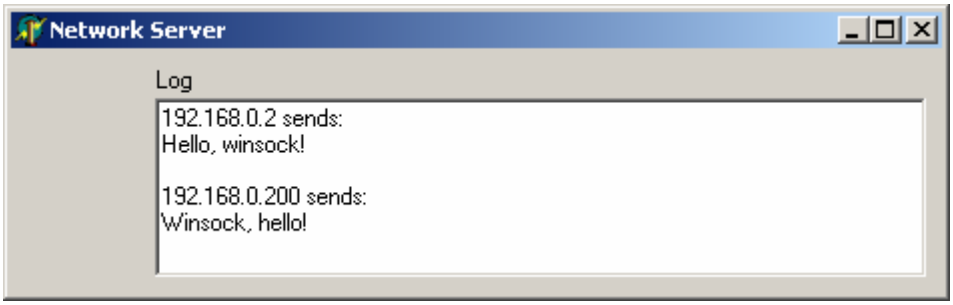


Figure 2.3: Text Receipt GUI
The server GUI showing received text from several clients

2.5.2 Java Sockets

Mahmoud (1996) emphasises that all the classes related to sockets are in the *java.net* package, which needs to be imported when using sockets.

Java Client

To program a client in Java, a socket would need to be created as follows (Code Extract 2.6).

```
Socket MyClient;
try {
            MyClient = new Socket("Machine name", PortNumber);
}
catch (IOException e) {
    System.out.println(e);
}
```

Code Extract 2.6: Java Client Socket Setup
'Machine name' represents the server machine's name or IP address. 'Port' is the number of the endpoint to connect to on the server. The rest of the code is for exception handling – in case an exception occurs, an explanatory message is displayed.

Reading and writing to a socket is demonstrated below in Code Extract 2.7.

```
DataInputStream input;
try {
   input = new DataInputStream(MyClient.getInputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

Code Extract 2.7: Intercepting TCP/IP Input
The DataInputStream class is used to create an input stream to receive response from the server, which can be recorded in a variable.

DataInputStream allows the client to read lines of text and Java primitive data types in a portable way. It provides methods such as read, readChar, readInt, readDouble, and readLine, as shown in Code Extract 2.8.

```
String response_line
response_line = input.readLine();
```

Code Extract 2.8: User Input Extraction
Reading user input into a string variable.

The DataOutputStream class allows writing of Java primitive data types. Many of its methods write a single Java primitive type to the output stream, such as *writeBytes*, as demonstrated in Code Extract 2.9.

```
DataOutputStream output;
try {
   output = new DataOutputStream(mysocket.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}

output.writeBytes("Hello there!");
```

Code Extract 2.9: Console Output
Writing output to the console.

Java Server

Creation of a Java network server requires binding the program to a socket and listening for and accepting incoming connections. A server socket is created as follows in Code Extract 2.10.

```
ServerSocket MyService;
try {
   MyService = new ServerSocket(Port);
        }
        catch (IOException e) {
           System.out.println(e);
        }
```

Code Extract 2.10: Java Server Socket Setup
Creating a server socket in Java.

Starting a server also requires the creation of a socket object from the ServerSocket class in order to listen for and accept connections from clients (Code Extract 2.11).

```
Socket clientSocket = null;
try {
   serviceSocket = MyService.accept();
        }
catch (IOException e) {
   System.out.println(e);
}
```

Code Extract 2.11: Java Connection Acceptance
Accepting incoming connections in Java.

Sending and receiving data is done in much the same way as with Java clients, except the *MyClient* socket variable is replaced with the accepted socket variable '*serviceSocket*' as shown next in Code Extract 2.12.

```
DataOutputStream output;
try {
   output = new DataOutputStream(serviceSocket.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}

output.writeBytes("Hello there!");

DataInputStream input;
try {
      input = new DataInputStream(serviceSocket.getInputStream());
}
catch (IOException e) {
   System.out.println(e);
}

String response_line
response_line = input.readLine();
```

Code Extract 2.12: Java Server Message Exchange
Sending and receiving data with a Java server.

Closing sockets involves terminating all created stream objects in both the client and the server. This is done with the *close()* function (Code Extract 2.13).

```
// Closing the client socket
try {
            output.close();
            input.close();
            MyClient.close();
}
catch (IOException e) {
   System.out.println(e);
}

// Closing the server socket
try {
            output.close();
            input.close();
            serviceSocket.close();
            MyService.close();
}
catch (IOException e) {
   System.out.println(e);
}
```

Code Extract 2.13: Closing sockets in Java

3 Literature Review

3.1 Introduction

This chapter aims to conduct a review of research related to the project topic, especially mobile agents and IDS. It is started off by delving into similar research directions. Research projects that are relevant to the thesis are discussed and compared where possible, in addition to discussing how this project relates to others, elucidating areas in which they meet and differ.

3.2 Similar Research Directions

3.2.1 Definition of Intrusion Detection

There are several different perspectives on the issue of secure computer systems. According to Janakiraman, Waldvogel and Zhang (2003), intrusion can be classified as the perpetration (or attempted perpetration) of taking advantage of a computer (or its associated resources) without rightful credentials, causing direct or indirect damage. IDSs are responsible for identifying intruding entities.

Other researches depict computer system security by describing what violates this security - intrusion. Eid (2004) defines intrusion as an attempt to exploit a computer network. According to Eid's work, intrusion detection comprises of several aspects, primarily detecting unauthorised and exploitive access to any device on the network. Secondly, IDSs have the capability of pointing out security vulnerabilities in a system. As well as administrating security policies and system "health" monitoring.

Experts may disagree on what topics need to be included in defining intrusion detection, however Helmer, Wong, Honavar, Miller and Wang (2002) provide a more comprehensive definition by stating that in order for these systems to be classified as secure, they must guarantee dependability on them. This dependability consists of availability, reliability, and confidentiality of its data and services. Since these systems are designed by humans, who are – by nature – prone to error, these systems end up with integrated design and implementation flaws, which culminates in security vulnerabilities. This in turn violates the guarantees set by secure computer systems. IDSs are responsible for detecting some set of intrusions and carrying out some necessary predefined action when a deviance is detected.

3.2.2 First Generation IDSs

Eid reports that first generation IDSs were entirely centralised. A single machine was responsible for monitoring and detecting intrusions throughout the network. This single host monitored data flow at certain points in the network and analysed data extracted from collected log files.
Ramachandran and Hart (2004) and Jansen (2001) extend Eid's narration of early IDSs by elucidating that these systems consisted of not only a single centralised host, but also data collection components. This implied that the central host's role was solely analytical, and was not responsible for data collection from logs and network monitoring.

It is also explained that that "while this architecture is effective for small collections of monitored hosts, centralised analysis limits the ability to scale up to handle larger collections." (Jansen, 2001). Researchers agree that the main limitation with these early systems was that the central host may be a single point of failure. If an attacker manages to successfully destabilise the host, the IDS would ground to a halt and the network would become vulnerable. A different approach was needed to tackle the task of developing an effective and robust IDS. This is presented in the following section.

3.2.3 Present Day IDSs

According to Hart and Ramachandran, the majority of IDSs currently in use are based on a hierarchical structure. Processed data from host and network levels are passed up the ladder until they reach the central coordinator, which in turn conducts analysis to determine the health of the overall system. These systems lack flexibility and making changes to the way things work is difficult.

"Present-day IDSs are less than perfect" (Martino, 1999, cited by Jansen, 2001).

Jansen (2001) states that some shortcomings in modern IDSs are inherent in the way these systems are constructed. Jansen gives a list of the most common shortcomings, shown below, which are essentially supported by Kruegel and Toth (2002).

- *Lack of efficiency:* Host-based systems may slow down when faced with a large number of network events. Network-based systems may drop packets that cannot be evaluated in time. Network events need to be processed in real time, and this requirement may be hard to keep if the events keep pouring in at a very high rate.
- *High number of false positives:* Since attack recognition is not perfect, some attacks may go undetected (by possibly lowering the alarm threshold). Switching the system to a more paranoid approach (by raising the alarm threshold) will generate many false positives.
- *Burdensome maintenance:* Configuration and maintenance of IDSs often requires expertise and extra effort. Special signatures must be registered for different types of attacks and additional adjustments must be made to cope with deviations in behaviour.
- *Limited flexibility:* In most cases, IDSs designed for a specific environment cannot be shifted to work in another. Adapting a system can be time consuming and problematic.
- *Vulnerability to direct attacks:* Any IDS is susceptible to attacks. Current systems lack many features that could make an IDS survive attacks, such as redundancy.
- *Vulnerability to deception:* Due to the way current systems are designed, an attacker can modify packets at a network level to fool the IDS into ignoring the attack.
- *Limited response capability:* IDSs were mainly designed to detect attacks in real time. Analysis of these detections and responding to them is not done dynamically, which gives attackers more time to freely operate.
- *No generic building methodology:* A lack of an agreed-upon framework for building IDSs acts as a setback to development in the arena.

3.2.4 Why Use Mobile Agents?

Several benefits motivate developers to delve into the world of using mobile agent technology. Researchers have highlighted numerous advantages that inspire the adoption of mobile agents.

Firstly, mobile agents assist in *network load reduction*. According to Lange and Oshima (2003), distributed systems by nature rely on communications protocols that involve multiple interactions to accomplish a given task. This undoubtedly results in a lot of network traffic. Mobile agents provide a mechanism that allows the developer to package a conversation and dispatch it to a known destination host, where interactions can be carried out locally. The result is the reduction of the flow of raw data in the network. When a process requires access to large volumes of data, it makes more sense to execute the given transactions in the locality of the data instead of it being transferred over the network. Sundsted (1998) further clarifies this issue by mentioning that a transaction or query in a client/server situation may require many round trips over the wire in order to complete. Each trip creates network traffic and consumes bandwidth. In a situation where network traffic would be quite extreme due to a high number of ongoing transactions (which may occur when a large number of clients are involved), the total bandwidth requirements to operate may exceed what is available, which would result in degraded performance for the application as a whole. By employing an agent to handle the transaction, and sending the agent from the client to the server, network bandwidth consumption is reduced. So instead of requests and replies being sent back and forth in order to complete an operation, only the actual agent needs to be sent. Jansen (2001) also points out a benefit that where confidentiality is a main concern, it is more efficient to move an encrypted agent and its refined data rather than moving all of the raw data in encrypted form.

Secondly, *mobile agents are intelligent and dynamically adapt*. Tanenbaum and Van Steen (2002) report that mobile agents are autonomous and can actively interact with their environment. This implies that if an agent's execution was paused and it was sent to a different host, it would resume execution from where it left off when it left its origin. This is further reinforced by Lange and Oshima who mention that mobile agents possess the ability to sense their execution environment and react autonomously to changes. Jansen also claims that agents may move elsewhere to avoid danger, clone themselves for redundancy and parallelism, or ask for assistance from other agents.

Finally, mobile agents are known for their *robustness and fault tolerance*. Lange and Oshima believe that mobile agents boast the ability to improvise when faced with critical situations. Sundsted supports this by demonstrating that unreliable network connections cause no panic to mobile agents. Most network applications nowadays require the network connection to be alive the entire time a transaction is taking place. The moment a connection goes down, the client often must restart the transaction from the beginning, if allowed to do so at all.

Sundsted also mentions that if a client were to dispatch an agent into a network environment, and then went offline, the agent would have the capability to handle the transaction autonomously. The result would be stored with the agent and sent back to the client once the network connection comes up again. Eid supports the aforementioned advantages and adds that implementing such systems in languages such as Java provide system and platform independence as well as plenty of security features which are essential to IDS.

Thus, the features mentioned above can dramatically improve the prospects of a system, and are compelling reasons to adopt a mobile agent based approach for use in different applications, including intrusion detection.

3.2.5 Agent Architecture

Firstly, there is a need to establish how to design and build an agent architecture. Designing and implementing support for several of the agent characteristics mentioned earlier is of high importance. The key requirements an agent architecture must satisfy are as follows (Sundsted, 1998).

- *Uniqueness.* An agent must have its own unique identity
- *Co-habitation.* An agent host must allow multiple agents to co-exist
- *Environment awareness.* Agents must be able to determine what other agents are executing in the agent host
- *Peer service consciousness.* Agents must be able to determine what messages other agents accept and send
- *Established communication channels.* Agent hosts must allow agents to communicate with each other and the agent host
- *Agent exchangeability.* Agent hosts must be able to negotiate the exchange of agents
- *Agent handling awareness.* An agent host must be able to freeze an executing agent and transfer it to another host
- *Agent operation reactivation.* An agent host must be able to thaw an agent transferred from another and allow it to resume execution
- *Inter-agent barrier.* The agent host must prevent agents from directly interfering with each other

The architectural requirements above provide support for the tactile and social characteristics of supported agents. Explicit support is not provided for the cognitive characteristics, as this is up to the agent developers to improvise.

3.2.6 Using Mobile Agents for Intrusion Detection

In an effort by Jansen (2001), it was stated that while mobile agents do not directly improve the techniques for detection, on the other hand, Helmer et al. (2002) believe that mobile agents have many advantages and solve critical problems in intrusion detection. However, Jansen mentions that mobile agents *can* reshape the way intrusion detection techniques are applied, resulting in improved efficiency and effectiveness. Helmer et al. agree and add that other critical problems can also be solved, like bandwidth and reliability. The two research groups also agree that mobile agents solve several problems due to their characteristic of being at the right place at the right time. They can monitor network activity at each host which ensures that no packets are missed (which may happen if the agent was centralised on a specific network segment). Attack resistant architectures can be facilitated thanks to an agent's ability to operate autonomously and make decisions when needed an also to share knowledge with other agents.

Jansen reckons that there are many advantages of applying mobile agents to responding to an intrusion:

- *Tracing an attacker.* Makes the task much easier due to distribution of agent platforms and using a shared knowledgebase.
- *Responding at the target.* When an attack is detected, it is crucial to automatically and swiftly respond. A quick response can prevent the matter from deteriorating if the attacker establishes a better foothold and uses the exploited host to further compromise the network.
- *Evidence gathering.* Data is audited from different hosts no matter what operating system or hardware platform is used. This information can form a knowledgebase for the agents.
- *Source and target isolation.* Mobile agents have the ability to travel to all network elements to carry out remedial work (according to defined strategies). This allows them to execute network level responses if other automated methods fail. The three available strategies are block the attacker's communications, block the target's communications, and block communications between the attacker and the target.

3.2.7 Related Work

Not many research projects have attempted to incorporate mobile agent technology into IDSs. Balasubramaniyan, Garcia-Fernandez, Isacoff, Spafford, and Zamboni (1998) proposed a system called AAfID that is claimed to be scalable by employing a hierarchical structure. The system architecture uses static agents that are distributed over a number of hosts. Each one of these agents carries out monitoring tasks and reports its findings to a higher authority agent called a *transceiver*. DIMA aims to adopt the same static agent concept by installing permanent processes on each host to be monitored. The transceiver agent analyses and compacts the data it receives and reports it to one or more *monitor agents*. AAfID tackles the central point of failure (CPF) shortcoming by ensuring that transceivers report to multiple monitors to provide redundancy in case a monitor fails (similar to the central hub IDS scenario). The main drawback of AAfID is its complexities that lead to design issues due to its complicated hierarchy.

An example would be in a situation if two monitor agents controlled the same transceiver, mechanisms must be employed to ensure consistency of information and behavior. Overall, out of the systems reviewed, AAfID is the most frequently referenced, mainly due to its early debut in the mobile agent IDS arena.

De Queiroz, Da Costa Carmo and Pirmez (1999) propose a novel system called Micael, that utilises five types of agents. Centralised system control is maintained by a *Head Quarter* agent, named the *QG*, which is usually stationed on a single host. The QG agent is considered the mother of all other agents in the system, due to its capability to create other types of active agents that carry out the tasks required for the system to operate. The QG is capable of migrating from its origin host only in threatening situations, like if the CPU load increases no the host due to frequent user activity, or when the host is invaded. The second type of agent in the system is the *Sentinel* that acts as a resident agent, collecting data at a host level and informing the QG about eventual anomalies. Periodically, the Sentinel backs up its data and execution state to the QG preventing data loss in the event of a system failure. The Sentinel is initially created by the QG and ordered to migrate to its destination host. After consulting external databases on its host, the Sentinel searches for anomalies and reports any threats to the QG, which in turn creates and sends a third type of agent, the *Detachment*.

A Detachment is a special type of agent that is created to combat anomalies detected by Sentinels. Upon creation, it is instructed by the QG to migrate to the threatened host, where it evaluates the situation and reacts accordingly, possibly by starting countermeasures. Once the threat is eliminated, Detachment migrate back to the QG, back up their data, and terminate. From time to time, the QG creates *Auditors*, which are protective agents that check up on the integrity of the active agents. These Auditors have the authority to reinstate other types of agents if they abort execution for any reason.

There are other kinds of agents in Micael, known as special agents that are created for special purposes such as detecting network attacks like flooding and spoofing. Although well designed, Micael resorts to the CPF approach, where the QG has the final word. I the event of QG failure due to an undefined exception, with the absence of *Auditors* in the scenario, the whole system would be crippled.

Helmer et al. propose another system that can be divided into several types of agents. At the lowest level are the data collector agents, which work in a similar fashion to the static agents in DIMA. Several types of data collectors exist, each monitoring different activities, such as failed logins and port scans. DIMA, being an early prototype, integrates several activity monitors into one single static agent. Next come the *low level* agents that monitor and classify ongoing events and activities reported from the data collectors. Different monitored activities require different types of low level agents. These agents are mobile and can migrate to other hosts to interrogate data collectors, and are assisted by *facet* agents in carrying out their roles. Finally, *data miner* agents use machine learning techniques to collect information that is passed up the hierarchy and processed by *higher level* agents. Unfortunately, no structure has been put in place for resolution of anomalies. DIMA is different in that its mobile agents are pre-programmed with intelligence and can take decisions swiftly on the spot and without referring to external entities.
Another unique proposal by Asaka, Okazawa and Taguchi (1999) designs a system to trace intrusions and gather information. The system consists of a *manager*, which resembles De Queiroz et al.'s QG agent in that there is only one of its kind and that it remains the highest authority. Reporting to the manager are *sensors*, which are basically static agents that monitor system logs for evidence of intrusion. In any intruder evidence is discovered, *tracing agents* (tracers) are invoked to trace the path of the intrusion, identifying its point of origin (the intrusion target host). Every time a tracer is started, an accompanying *information gathering agent* (infoagent) is activated, which analyses evidence that was collected and migrates back to the manager and reports its findings. Also, each time a tracer migrates to a new host, a new infoagent created. Although they arrive at very close proximity of the intrusion evidence, neither tracers nor infoagents have the capability of deciding whether an intrusion has actually occurred. On the other hand, DIMA's Maldes agent does its analysis before taking decisions at each target host by providing resolutions before migrating to its next host. Additionally, Asaka et al. devised a clever mechanism for local information exchange through the use of a shared *message board* that sits on each host. Tracers and infoagents post information relevant to decide whether a track has already been traced by other agents, therefore saving time and preventing trace route overlaps by two or more agents. A second centralised mechanism, called a *bulletin board*, resides on the manager machine and records more extensive information gathered by different agents from their target systems. Asaka et al.'s agents focus on tracing intrusions rather than detecting their presence or eliminating them. The analysis is conducted centrally by the manager, who judges and declares whether or not an intrusion has occurred. As previously mentioned, centralised solutions like Asaka et al.'s manager pose as a CPF setback.

A novel P2P approach by Ramachandran and Hart presents a distributed mobile agent based IDS with no central coordinator. The system design indicates that hosts in a network are divided in *sites* that are equal in authority (peers). These sites are logically arranged into virtual neighborhoods, and must all be running special IDS software. Intrusions at a site are either detected locally from within the site, or remotely by a nearby neighbor using mobile agents. Any information related to the safety of a site is distributed among neighbors. Upon detection of an intrusion, the neighbors of the suspected site are polled and a joint decision is made to take action.

Although the system framework is relatively robust by eliminating the need for a dependency such as the central processing station, it may raise questions by separating the agents from their required data. One merit of this system is its distributed decision making. All decisions are made after a majority vote is reached between neighbors. DIMA avoids this approach and opts for standard on-site intelligence, which overtakes the dependency on numerous external responses and frequent delays that affect the overall performance of the system.

Buchanan, Graves, Saliou, Al Sebea, and Migas (2005) outline an integrated framework for both data gathering, using mobile and static agents, and also in the creation of a data gathering system which logs data in a verifiable and open way. The framework for logging data for future investigations uses a formal approach where a forensics policy is defined, which is then compiled into an implementation which can run on IDS agents. The paper also proposes a system which uses mobile and static agents to formalise the investigation process. The mobile agent framework proposed defines an outline of the usage of the mobile agent in a forensics system. Within this, the investigator has no direct interface to the host-under-investigation (HUI). This thus keeps the integrity of the system, where a static agent is immediately installed on the HUI, and guards against any changes to the information stored on the HUI. The investigator then gives the mobile agent requirement, such as the names of the executable files on the HUI, and the mobile agent migrates itself to the HUI, and authenticates itself to the static agent, and vice-versa. The mobile and static agents can then intercommunicate with each other and pass information about the required case. Both the mobile and the static agent are programmed with a framework which supports both legal and moral constraints, thus any part of the investigation which breach the limits of the current investigation will be stopped.

3.3 Summary

First generation IDSs depended on a centralised approach, where a single host was responsible for monitor and analysis operations. This designed suffered from a severe drawback that surfaced when the central station came under attack from intruders. This destabilised the system and brought the whole IDS down. Researchers worked on improving this flaw by devising hierarchically structured systems that eliminated the need for a central host that could be compromised. Data processed from hosts passes through several higher layers until it reaches the topmost authorities that determine the level of threat and make countermeasure decisions when needed. More problems arose with these present systems, mainly lack of efficiency and burdensome maintenance.

A considerable replacement for client/server scenarios that require heavy data exchange between hosts is the use of mobile agents, where the required process migrates to the point closest to the data it requires, and performs its tasks locally. Depending on the situation, a mobile agent can be trained (with intelligence for example) and assigned a mission, which it will accomplish at its own autonomy and return with a report of its findings. Mobile agents provide numerous advantages over the client/server approach, including conservation of valuable bandwidth by reducing network load.

Introducing mobile agents to the field of IDSs is an innovative proposal, which according to researchers can solve stringent shortcomings, such as reliability. Most recently proposed IDS projects share a common design blueprint, where the system is hierarchical. Clever designs incorporated a certain level of security vertically to preserve system integrity and prevent tampering. The general structure consists of low level static agents that collect information and report to a higher level entity (AAfID). In other designs, the topmost authority is informed, which in turn takes appropriate action (De Queiroz et al.). Mobile agents have been utilised in some systems, responsible for traversing the network and picking up information from collector agents (Helmer et al.). Some proposals only concentrated on outlining tracing techniques, whilst others focused on creating diverse neighbourhoods of hosts that frequently exchanged updates to trap and disable intruders.

4 Requirements Analysis and Design

4.1 Introduction

The main aim of this project was to create a prototypical IDS that would tackle one of the problems that arise in almost all computer networked environments. The leakage of malicious software such as trojans and worms could prove devastating to such an environment if not countered by strict mechanisms.

This Section first defines the overall system requirements and then delves into the details of each of DIMA's components.

4.2 System Component Definition: Overall System Requirements

The proposed host-based system, named *DIMA*, consists of three components: static agent, mobile agent, and host. All three are described below.

The first component is the *static agent* (named FileMon, short for File Monitor) that resides on every machine in the virtual laboratory. This is an agent that acts as a server whose responsibility is to make note of any unusual file appearances within certain vulnerable system folders and report them to authoritative entities. Once started up, it reverts to a standby mode, where it monitors the host's file system in the background. The result of the monitoring process is a compiled list of files that have recently been created on the host machine. Upon arrival of an authoritative entity, FileMon reports this list and awaits advice on how to deal with each reported file individually.

The second portion of the system proposed is the authoritative entity mentioned above, the *mobile agent* (named MalDes, short for Mal-ware Destructor, used interchangeably with the term 'authoritative entity'), which acts as a client to FileMon. This agent has built in intelligence in order to correctly identify and combat certain intrusions caused by mal-ware (viruses, worms, trojans and spy-ware). This agent is allocated a list of local hosts to visit (an *itinerary* or *agenda* – used interchangeably) and is dispatched from the system administrator's machine (the investigator). It iterates through the list given, migrating to each host and performing a predetermined set of maintenance tasks. MalDes' intelligence is used to advise FileMon on how to remove malicious files and ignore what is known to be innocuous.

The third component is the *host*, which is the protected machine where the preceding two components run and all inter-agent communication transactions take place.

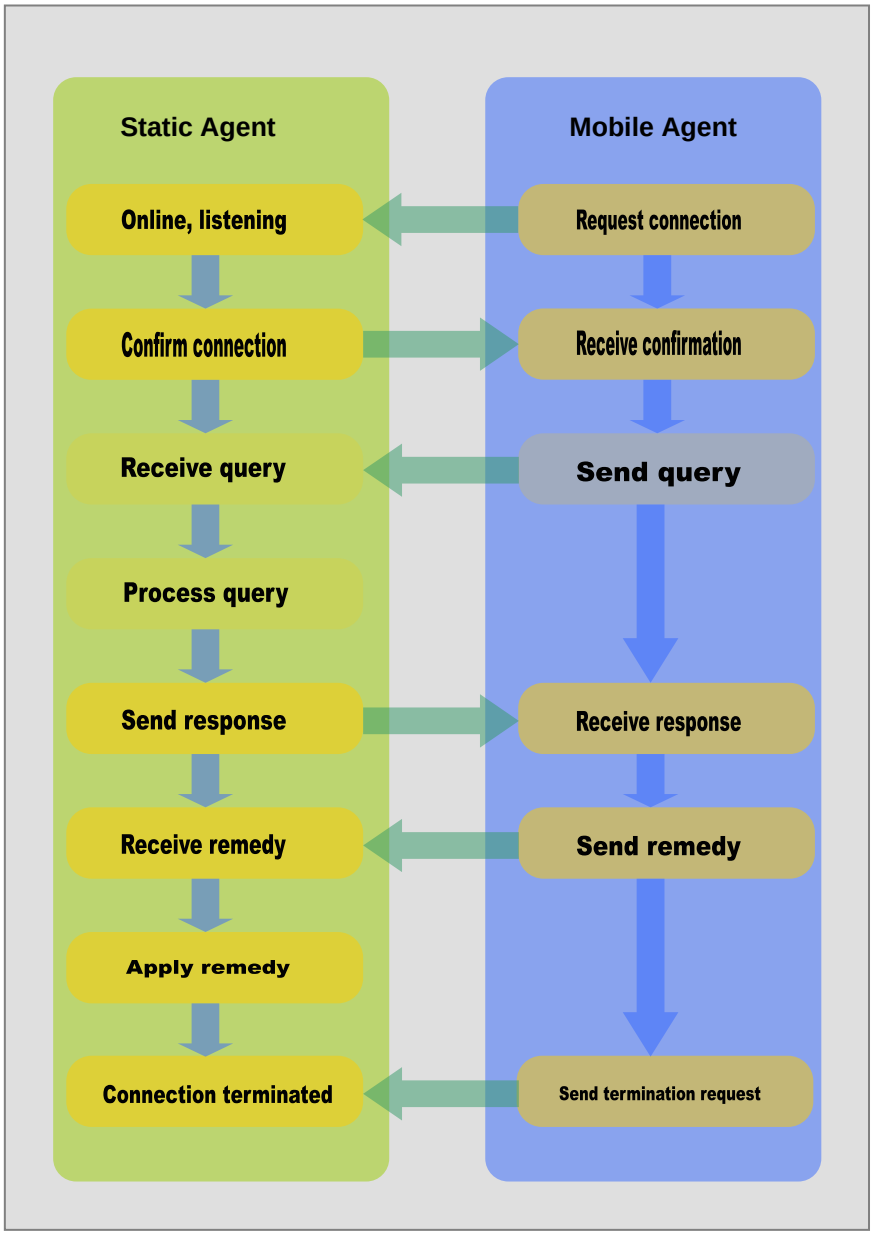


Figure 4.1: Connection Summary Between FileMon and MalDes

4.3 FileMon Requirements and Design

4.3.1 Introduction

FileMon has several roles that contribute to helping tackle potential intrusions perpetrated by mal-ware. Being located at the target machine to be monitored, it is considered the closest it can get to the local viral playground. This agent must have administrator's credentials in order to be able to have direct and unconditional access to the machine's file system and registry. FileMon's roles will be outlined next, along with sample natural language *pseudo code* demonstrating how they would be carried out.

4.3.2 Static Agent Requirements

Main Requirements

The core requirement of this program is the capability of monitoring the host system for any newly created files in a specific group of folders. If any file matches the set criteria, file information is collected and used to build a list of suspects. The agent also binds itself to a specific TCP port and listens for incoming connections from anticipated clients. This procedure is called opening a socket. Once a socket is opened, external authoritative entities will need to exchange messages with FileMon, which consequently reports the suspect list it has built. Message exchange implies the need for a custom communication protocol. This protocol is required to include commands relevant to all parties in the conversation, including FileMon, allowing them to express themselves adequately, to reach the required solution. During communication, exchanged messages as well as executed actions are required to be logged. This should be provided in the form of onscreen feedback to the investigator. After a communication session ends, FileMon must have the ability to execute any advice given by MalDes.

Programming Requirements

Finding the most suitable programming language and development environment for FileMon is an important decision that takes into account several factors. Among these factors are rapid development capability, performance, third party component availability, and executable robustness.

Therefore, numerous programming options have been identified, including Visual Basic, and Borland Delphi. The most appropriate development environment will be justified in Section 4.6.

4.3.3 Static Agent Design

File Modification Monitor

Of the first roles FileMon performs is being a *file modification monitor*, keeping an eye out to any additions or changes to executable files in specific system directories. This very much works like a mouse trap, catching anything that treads across its path. The agent is set to guard a specific set of folders, and any executable that is created or modified within those folders gets noted in a suspect list. This consequently implies that the population of this list may not necessarily suggest the prevalence of malicious files, and may indeed contain false positives. Since the size of this list may grow over a period of time - depending on user and system activity – the contents of this list must be filtered frequently to completely purge the system of any hostilities. This is where MalDes plays its role by informing FileMon of which files are infected and helping it remove these *infections*.

The pseudo code for this function would be as follows:

```
Set folder to be monitored as the Windows System folder
Enable the monitor

If a file has been created in this folder and extension is .exe
        Add file to suspect list and note its parent folder
Endif
```

Network Communication

As previously mentioned, FileMon, which sits on its monitored machine, needs to communicate with MalDes once it arrives. This is done on a *local host* basis. Since FileMon contains a server that opens TCP port 2005 upon starting up and binds itself to the local IP address, it listens for any incoming messages from roaming MalDes agents that may visit. In turn, mobile agents that arrive at the host's machine attempt to establish a connection on the local host address. FileMon understands a protocol that is spoken by MalDes, allowing it to state if it has noticed any changes through its file monitor. Upon arrival of a MalDes agent, it initiates the conversation by asking FileMonif it has noticed any file alterations recently. FileMon responds to its client, the newly arrived MalDes agent, depending on its situation. In the event of the existence of possible suspects, the client/server conversation continues until FileMon is informed how to deal with each individual file's case. MalDes will stay on the host machine, until all files have been *remedied*. The pseudo code for this role is shown below:

```
Set TCP server port to 2005
Activate TCP server

If mobile agent arrives
        Respond according to defined protocol
Endif

If mobile agent has finished and has left
        Do cleanup process
Endif
```

Agent-to-Agent Protocol

Communication between the static and mobile agents is built on a simple protocol that allows FileMon to report suspects to MalDes, and for the latter to respond with each suspect's possible *remedy* (process of removal) in a format understood by both parties. A demonstration of a typical agent-to-agent conversation is shown below as a *pseudo protocol* in Table 4.1.

Mobile/Static Agent Conversation	
Agent	**Message**
Mobile Agent (M)	Hello, do you have any suspects to report?
Static Agent (S)	Indeed, I do. May I report them please?
M	How many suspects do you have?
S	I have two suspects here.
M	OK, list your first suspect.
S	File name: msnull32.exe
M	I recognise that file, it's a spy-ware program.
M	You need to end a task running on your machine, delete the file, and remove some registry entries.
S	OK, I got that.
M	Any more suspects to report?
S	Yes, file name: iexplore.exe
M	That file is not malicious, you may ignore it.
S	OK, I got that.
M	Any more suspects?
S	No, that's it for now.
M	OK, goodbye.

Table 4.1
A Typical Inter-agent Conversation

The protocol shown above may at first seem quite verbose with regard to the conversation overhead before a solution is negotiated. If it were to be transformed into a proper protocol to be applied, almost a handful of messages are exchanged prior to MalDes getting to the point.

The reason this approach was used was to simplify the task of programming agent-to-agent interaction, by separating the conversation into several commands, in order for each program to prepare for the next step. In the event of no files being noted by FileMon, it simply responds by stating that its suspect list is empty, giving MalDes permission to leave.

Local Administrative Functions

It was outlined in the introduction to FileMon that it *must* have administrator's credentials. At the end of a successful agent-to-agent conversation, once FileMon receives its required remedies, it needs to apply what it has been told. Suppose FileMon was started on a machine where a restricted user was logged on. To perform remedial routines - such as deleting a file - it would be of no use if the program was greeted by an 'access denied' exclamation from the operating system. For that reason, the user that is responsible for running FileMon must have administrative rights.

There are three individual operations that FileMon is capable of executing:

- *File deletion*: to delete a file from anywhere on the local machine's file system.
- *Ending tasks*: to *kill* a program – ending all instances of its process
- *Registry entry removal*: to remove a programs entry from the operating system registry. This entry could be used to launch the program every session.

The pseudo code for these operations would be as follows:

```
Get remedy advice from mobile agent

If remedy exists for file
        Kill file process
        Delete file
        Delete registry entry for file
Endif

If no remedy exists for file
        Ignore file
        Remove file from suspect list
        Display feedback saying file is harmless
Endif
```

Remedy Application

In DIMA, mobile agents send remedies to their static counterparts in the form of *remedy packages*. A remedy package is simply a string of key bits of information relevant to the removal process of the infection (see Figure 4.2).

This includes the file's *human name*, such as 'Sasser worm', the process image name, such as 'msprc.exe', and the registry key and corresponding entry that corresponds to the file, allowing it to start automatically when the operating system loads at every session.

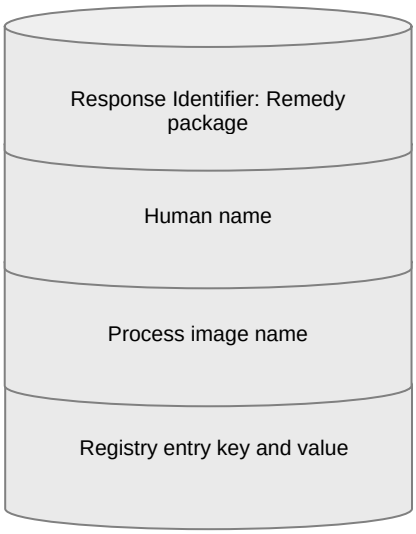


Figure 4.2
Remedy Package Structure

In the two way conversation, requesting a remedy for a file results in MalDes sending back a single reply that contains the comprehensive remedy package. Whether or not the remedy is actively applied is up to FileMon, not the MalDes agent. In the event of the file being harmless, MalDes will simply state that it has no remedy available for the file being reported. The pseudo code for remedy application would be as follows:

```
Get remedy package from mobile agent
Parse remedy package
Extract contents of remedy package.
```

The pseudo code continuation is shown in the preceding *Local Administrative Functions* subsection.

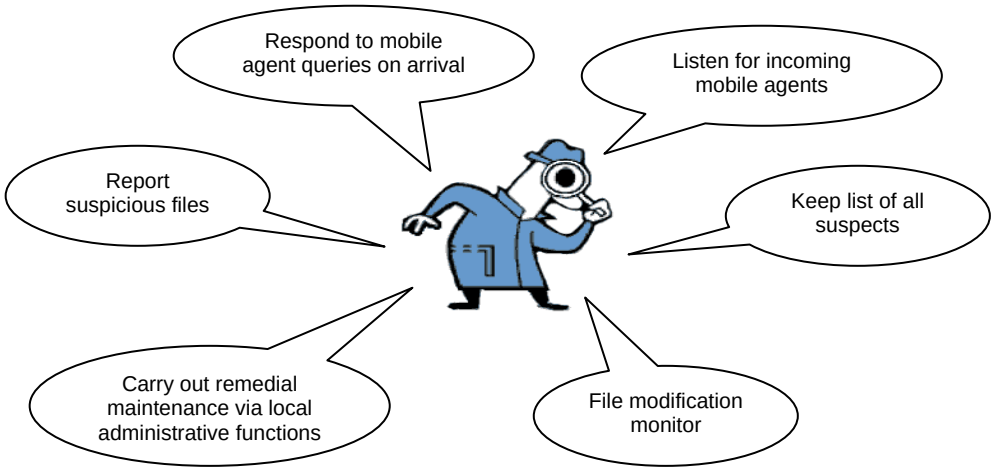


Figure 4.3
Static Agent Roles

User Interface Design

Based on FileMon's requirements that were listed in Section 4.3.2, the following prototypical user interface was proposed:

Main Window

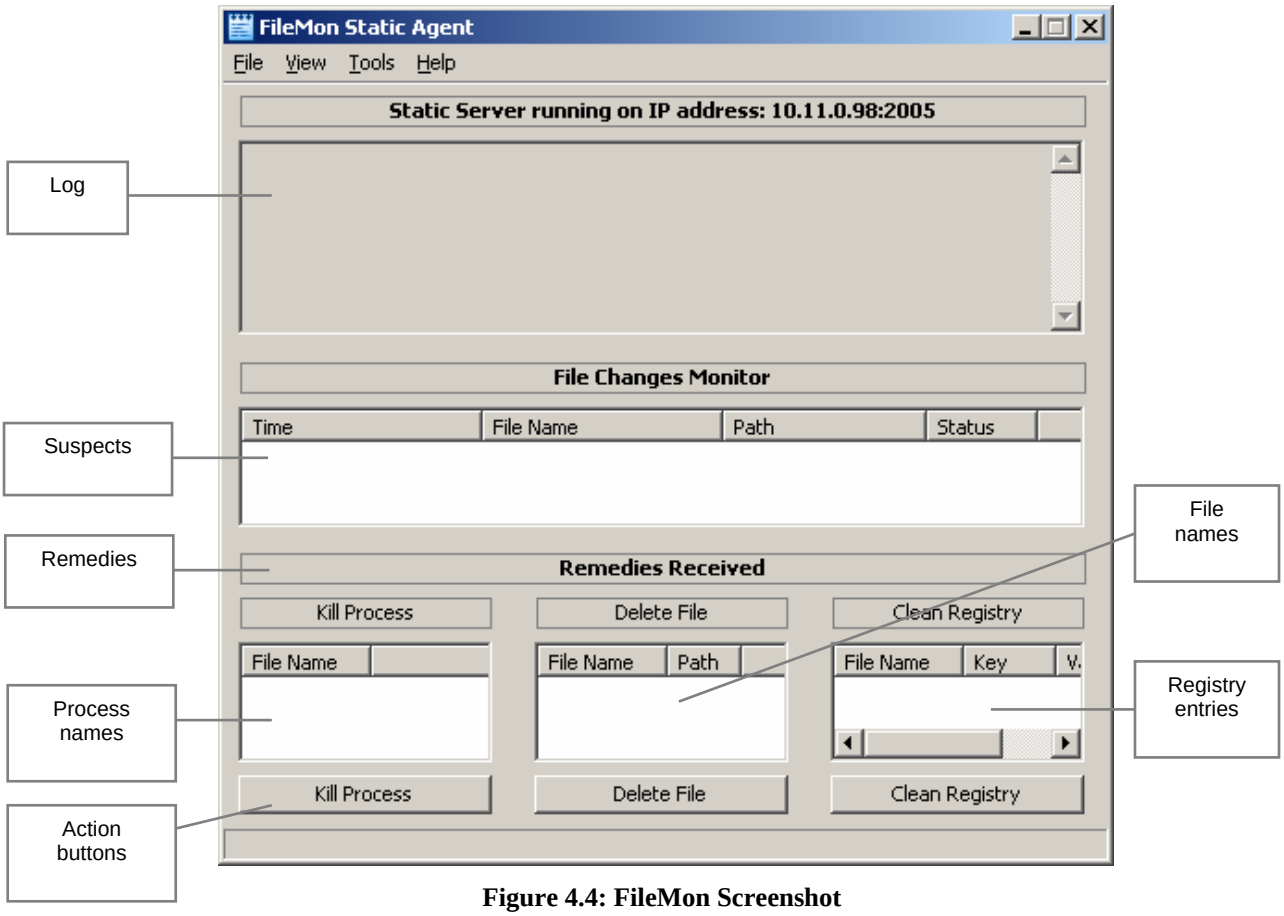


Figure 4.4: FileMon Screenshot

FileMon has one main window where several pieces of information are displayed. This information is categorised as follows (according to the preceding Figure 4.4):

- Any communication made through the TCP socket is displayed in the *log*.
- Files that have been noted as possible suspects are added to the suspect list, along with their corresponding path. A timestamp is also included with the suspect, and the file's *status* stays blank until MalDes updates FileMon on whether this file is harmless or infected.

Settings Window

The settings window is accessible through the menu system (Tools | Settings).

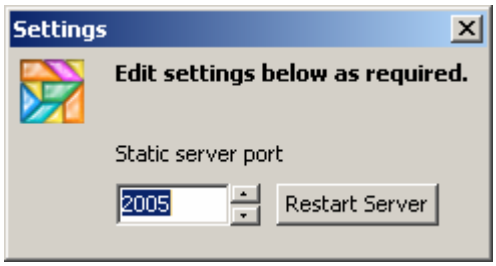


Figure 4.5
Settings Window Screenshot

For flexibility purposes, a port change option was incorporated into FileMon, to alleviate conflicts with other network applications and firewalls. The default FileMon *server port* is 2005.

Upon the receipt of a remedy from MalDes, information is extracted and distributed among three lists:

- *Process names*: all processes in this list are terminated.
- *File names*: all files in this list are destroyed.
- *Registry entries*: all values in this list are removed from the registry.

Once all remedies have been received, the three remedy lists will have been populated. The *action buttons* execute the required actions on the contents of their corresponding lists:

- *Kill process*: terminates all processes in the *process names* list.
- *Delete File*: deletes all files in *file names* list.
- *Clean Registry*: removes all values in the *registry entries* list.

4.4 MalDes Requirements and Design

4.4.1 Introduction

MalDes is the core component of DIMA. With its ability to actively migrate to predefined hosts, it ensures that these monitored machines are kept free of infections by utilising its intelligence to instruct each FileMon process on how to react to problematic file appearances. MalDes' roles will be described next, including sample pseudo code.

4.4.2 Mobile Agent Requirements

Main Requirements

Fed with a list of target hosts to monitor, the agent sets out on its journey to seek out malicious files and eradicate them. A unified protocol is used for communication between the static and mobile agents. The nature of the conversation between the two programs is interrogatory at first, and then informative. FileMon uses MalDes' experience to identify and remove any files that are considered dangerous to the overall health of the host.

MalDes is required to have the ability to migrate to any hosts it is allocated, provided that these hosts are reachable network-wise. Any unexpected communication errors (such as subnet invalidity) must be handled gracefully. The agent is responsible for determining its own agenda, which makes it autonomous in many aspects.

Programming Requirements

An important note on mobile agents is that they cannot be directly executed on an operating system like Microsoft Windows. In fact, they run on special software execution environments (mobile agent platforms). This platform software must normally be installed and configured on a Windows system before mobile agents can be run on it. Several platforms exist for Windows, such as Grasshopper and IBM Aglets Software Development Kit (ASDK). The most suitable platform will be chosen in Section 4.6.

4.4.3 Mobile Agent Design

Multi-host Migration Ability

Being a mobile agent requires the ability to relocate often. The design of the MalDes portion of DIMA, along with the powerful underlying Grasshopper Mobile Agent Environment accomplishes this required mobility. Invoked from its launch pad (the investigator's machine, also called *base*), MalDes can be given an agenda, and it will comply accordingly by paying visits to as many hosts as needed. An internal tracking system (built into the agent) is used to keep a *checklist* of the hosts that have already been visited, and to determine where to go next.

After the checklist has been iterated through, it will return to its launch pad. The pseudo code for this characteristic is shown below:

```
Get list of hosts to migrate to
Initialise total_hosts to zero
Process list and set total_hosts to number of items in list

Initialise current_host to zero
Determine next host address using current_host as index to the list

Do While current_host  is less than or equal to total_hosts
        Migrate to host
        Contact FileMon
        Carry out remedial maintenance
        Close connection with FileMon
        Increment current_host
        Determine next host address
EndDo
```

Active Intelligence

When interrogating FileMon, this agent must be able to determine – on the fly – whether the file reported as a suspect is indeed dangerous to its host, or whether it was a false alarm. To be capable of fairly convicting a reported suspect, a quick but precise comparison must be made. The file's identity (file name) is searched within a list of criminal process names (signatures). If this search yields a match, then the agent looks up the corresponding remedy method and sends it to FileMon. Below is an example of how these signatures would look like:

```
        Process name: msblast.exe
        Human name: Blaster Worm
        Registry Value: HKEY_LOCAL_MACHINE\...\Run\Blaster
```

Sample infection signature

Protocol Compliance

Since FileMon understands the aforementioned protocol, MalDes must speak the same *language.* Therefore the pair complement each other and know what to next expect in the spoken conversation in the same manner as completing a jigsaw puzzle.

System Status Report Generation

DIMA would be incomplete if no type of reporting mechanism was put in place. Sitting from a remote location, the investigator would dispatch a MalDes agent, assigning it a checklist of hosts.

The agent would traverse the network, stopping at certain hosts to conduct its interrogations and give remedial advice. Once the checklist has been successfully completed, the agent returns home, to its launch pad. If no report generation mechanism were in place, the investigator would have no clue of whether the agent had encountered any problems along its journey, which hosts were infected, and which were unharmed. For that reason, a log – similar to a field trip diary – is kept and updated each time the agent completes work at one of its assigned hosts. This means that once it returns to its launch pad, MalDes will *remember* in detail what had been encountered during its round trip. With the reporting mechanism in place, the status of each of the machines visited is displayed, along with the human names of any infections found. A sample is shown below:

```
Machine:  "accounts-server" is infected with the Blaster worm.
Machine: "accounts-client-1" is infected with the Pro-Rat trojan.
Machine: "proxy-1" has no problems.
```

Sample generated report, listing machine names and any infections found.

The pseudo code for report generate would not differ much from that of 'Multi-host Migration Ability' except for what is shown in *bold*. See below:

```
Get list of hosts to migrate to
Initialise total_hosts to zero
Reset reports to blank
Process list and set total_hosts to number of items in list

Initialise current_host to zero
Determine next host address using current_host as index to the list

Do While current_host  is less than or equal to total_hosts
        Migrate to host
        Contact FileMon
        Carry out remedial maintenance
        Close connection with FileMon
        Generate report on this host
        Increment current_host
        Determine next host address
EndDo
```

Failsafe Design

Due to the nature of some operating systems, and possibly the underlying computer networks, inconsistencies may arise from time to time. To demonstrate, assume a mobile agent was assigned to a group of computers, linked by a switch.

One of the group's members IP address or subnet mask may be modified (mistakenly or intentionally), isolating it from the rest of the group. This would naturally prevent the agent from reaching the isolated host. In a similar situation, network connections may go down temporarily, preventing the agent from reaching a host at the other side of the switch. The default reaction to these contingencies would be to halt code execution and exclaim an error. These situations require the agent to stay robust and improvise. For that reason, the agent must be able to skip a host if it cannot reach it and continue with the rest of its checklist. Additionally, any destinations that were skipped need to be reported to the investigator as well. The pseudo code for this feature is as follows:

```
If cannot contact next host
        Mark this host as unreachable
        Increment current_host
        Migrate to next host
Endif
```

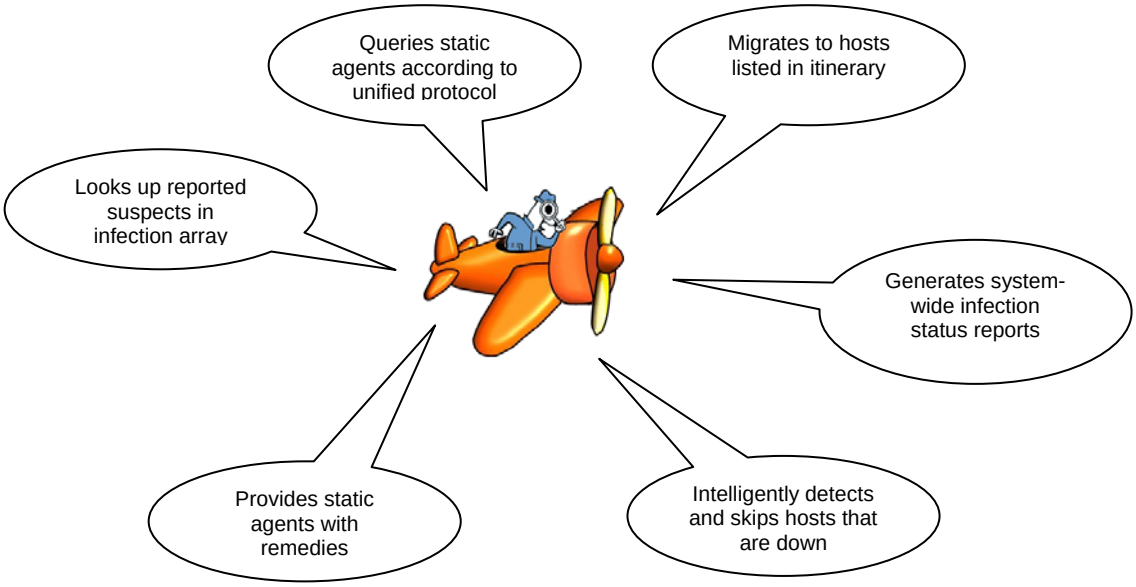


Figure 4.6
Mobile Agent Roles

4.5 Analytical Experiment Design

4.5.1 Aims of Experiments

Now that the overall design of DIMA has been described, a set of experiments need to be devised and carried out on the system to prove and test its functionality. The basic aim of these experiments is to test DIMA's performance in different situations. Without these experiments, it would not be possible to evaluate the design of the project. Naturally, discovery of design issues and inconsistencies is a by-product of these tests. Below is an outline of the experiments that will be conducted:

4.5.2 Experiment Construction

For the following experiments, the mobile agent environment needs to be installed correctly and it must be ensured that a copy of FileMon is operational on each host to be visited.

Return Trip Timer

This experiment will attempt to measure the amount of time an agent takes to migrate to a different number of hosts (each visit is called a *hop*) and return to its launch pad on a local area network. It will be assigned a different number of hosts for each test, and will calculate how long it takes to migrate to all allocated hosts and return successfully to its origin. The timer functionality will be built into the agent, so no external tools will be required for this experiment. The agent will be tested 4 times, according to the following table:

Return Trip Timer	
Test ID	**Number of Hops**
A	1
B	2
C	4
D	8

Table 4.2
Return Trip Timer Tests

Host Performance Timer

For this series of tests, the number of hops will be kept the same. Two groups of hosts will be organised. The first group shall consist of four computers of the same hardware specification (Pentium 4 2.8 GHz machines with 512MB RAM), to be called the *superior group*. The second group shall consist of the same number of computers, except their hardware specifications shall be relatively lower than the first group (Early Pentium processors with a maximum of 128MB RAM), to be called the *inferior group*. A return trip test shall be conducted on both groups of machines, and the amount of time taken shall be compared. The timer functionality will be built into the agent, so no external tools will be required for this experiment.

Agent Size Measurement

As mentioned in Section 4.4.3 (System Status Report Generation) as more hosts are visited, more data is stored in the agent. This leads to assume that agent size increases as more hops are made. This test shall attempt to prove that assumption. An external tool called Ethereal is required for this experiment. Ethereal is an open source network protocol analyzer. It allows precise location of the packets that are exchanged between hosts to achieve agent migration.

Once these packets are pointed out, it is possible to manually calculate the size of the data being sent (the actual agent itself).

Internet Return Trip Timer

This experiment very much resembles the Return Trip Timer series of tests, except that it is conducted over the Internet. A mobile agent will be assigned a host to visit that resides on another network. This would require the agent to traverse through the Internet in order to reach its destination, and return back to its launch pad.

4.6 Summary

To conclude this phase of the project, the design foundations can be outlined as a series of blueprints. FileMon, with its administrative credentials, has been assigned multiple tasks which it is to execute autonomously without the interference of any external entities. These tasks are considered pertinent to its overall purpose and functionality, and can be divided into various categories. The file modification monitor is responsible for eavesdropping on certain Windows API calls to detect whenever executable files are created or modified. By keeping record of these files, FileMon files a report on each file and requests advice from the visiting mobile agent. The program's local administrative methods allow it – with the help of mobile agent intelligence – to carry out remedial chores to rid its host of any known *mal-ware*, leaving these infectious files at a colossal disadvantage. For FileMon application development, Borland Delphi has been chosen due to its excellent integration with the operating system and readily available third party software components. Even though the key strength of FileMon is its ability to cripple mal-ware and hinder its propagation, its prime weakness lies in it being a process itself, vulnerable to termination at any time, spontaneously.

On the other hand, MalDes, equipped with infection signatures, roams the network visiting hosts and interrogating FileMon processes therein. It is capable of communicating with FileMon processes through a unified protocol, and can make instantaneous decisions on the criminal status of the files in question, defeating mal-ware presence. For mobile agent development, both Grasshopper and IBM ASDK were found to support Object Management Group's Mobile Agent System Interoperability Facility (OMG MASIF) standard. Taking a closer look, Covaci (1998) states that Grasshopper is the first MASIF compliant mobile agent platform, as it has built the basis for several agent-related European research projects. In addition, in-depth documentation is available, which serves as a great help in building a strong foundation in mobile agent development on this particular platform. For the aforementioned reasons, Grasshopper has been picked as the mobile agent platform of choice.

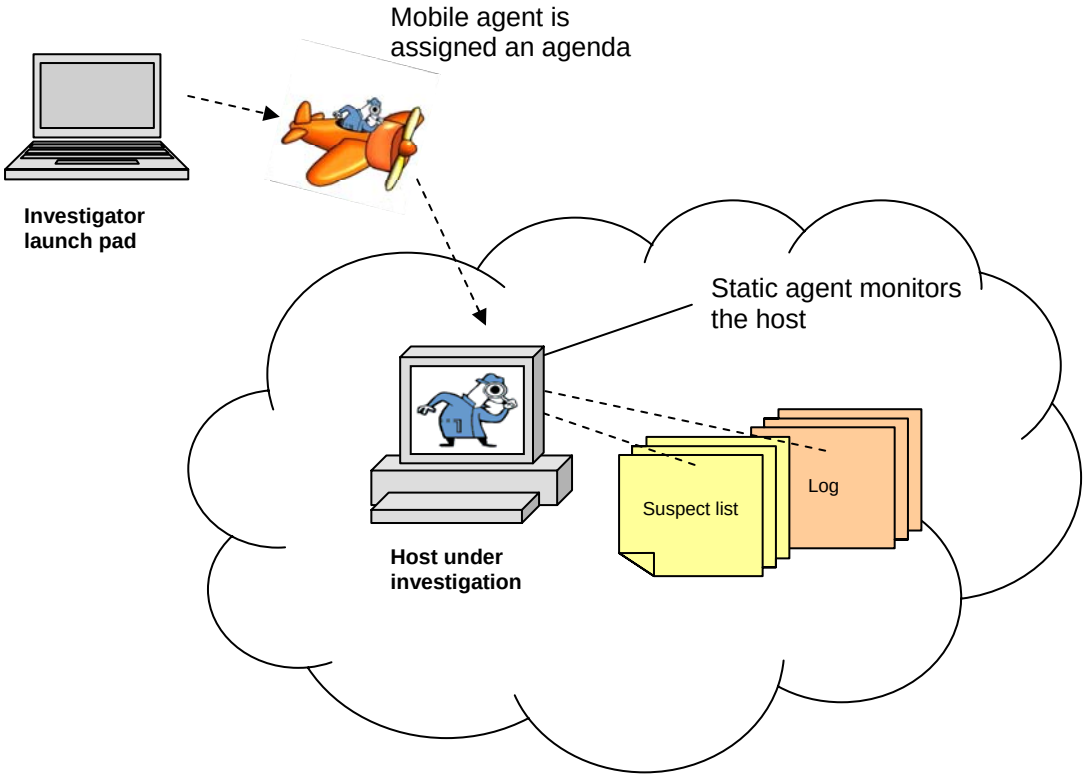


Figure 4.7 Agent Interaction
Mal-ware detection and immunisation system using mobile agents

The mobile agent approach can prove tremendously convenient, especially to system administrators but comes bundled with a drawback that can be spotted immediately. If the agent platform software is disabled or shut down on a particular host, this would prevent mobile agents from migrating to and from that host, posing a threat to its security. This threatening problem can be eliminated by ensuring all software run in the hidden in the background and are started as system services if possible (granting them higher credentials than administrators and thwarting mischievous user attempts to hamper the system's smooth operation).

5 Implementation

5.1 Introduction

This Section describes the phase of the project in which DIMA was implemented and tested. An introduction is given to the mobile agent development environment from a programming perspective, followed by the static and mobile agent implementation processes.

5.2 Grasshopper Programming Briefing

Since agents are not standalone programs, but instead Java classes, they must run on a specialised platform – Grasshopper, allowing support of running and maintaining agents. It must be noted that in order to enable agents to run on a set of hosts, Grasshopper and the Java Runtime Environment must be correctly installed on each machine where an agent may run. Grasshopper takes care of inter-host communication that involves agent migration from between hosts. In addition to standard Java code support, Grasshopper-specific functions exist in the form of a special Application Programmer's Interface (API).

In a typical situation, the Grasshopper GUI would be running on a host and would declare an *agency*. Agencies must be named to differentiate between them. Agents can move around hosts by knowing the names of the different available agencies.

To allow agent identification, agents must be given human names. In addition, a unique agent identifier is automatically generated by running agency during the creation of an agent. From a general perspective, as previously stated, a Grasshopper agent is implemented by means of a Java class or a set of classes. Each agent has one agent class which characterises the agent and which must be derived from one of the predefined *super-classes*. Several super-classes exist, including mobile-agent and stationary-agent. Due to the nature of this project, with the involvement of mobile agents, the mobile-agent class shall be adopted. (IKV [B], 1999).

Through its super-class, each Grasshopper agent has access to the following methods (IKV [B], 1999):

- **action( ):** This method is automatically invoked by the agency if a user performs a double-click on the corresponding agent entry in the agency GUI.

- **getInfo( ):** This method returns a set of information that is associated with the agent. Among others, this set of information comprises the agent's identifier, type, and name.

- **getName( ):** This method returns the name of the agent. In contrast to the unique agent identifier which is automatically generated by an agency during the creation of an agent, the name can be specified by the agent programmer during the implementation phase or by the user when creating the agent, provided that this is supported by the agent implementation

- **init( ):** This method is automatically called by the hosting agency when an agent is created. It offers the possibility to provide creation arguments to the agent.

- **live( ):** This is the core method of each Grasshopper agent, since its implementation realises the agent's active, autonomous behaviour.

- **log( ):** Allows an agent to print textual messages onto the text console of the local agency.

- **move( ):** With this method, an agent is able to migrate to another agency

5.3 Implementation Details: FileMon

The development phase of FileMon was divided into several segments, to ease implementation. The local administrative functions were tackled first, followed by the network connectivity and the protocol compliance segment. Finally, the remedy application portion was implemented. Details of each of these stages of development are elucidated below.

5.3.1 File Modification Monitor

In order for the program to detect any recently created files, specific Windows API calls must be referenced. A third party component written by Shevin (2000) provided this functionality. It allows the appropriate Windows events to be captured when a new file is created and hence acts as a wrapper to the Windows API calls needed to implement the file modification monitor. By default, the component monitors all folders on the machine it's running on, including removable drives. Since the project requirements indicate the need to monitor a specific folder on the host machine, the program needs to be adapted to filter the captured events and only log the files that were created in that folder.
The two most important events required are *ChangeNotify* and *UpdateDir*, both triggered when any new file is created in the monitored folder.

The events *ChangeNotifyRenameItem*, and *ChangeNotifyCreate* are triggered when a new file is created in the monitored folder and are hence handled by their corresponding procedures. These procedures subsequently call the main procedure *ScanForChanges* to detect whether the file mentioned fits the criteria or not. The criteria set ensures that the file extension is of type 'exe' and that it has been created in the folder that the component has been told to monitor.

```
procedure TfrmMain.ChangeNotifyCreate(Sender: TObject; Flags: Cardinal;
  Path1: string);
Begin
  ScanForChanges(Path1);
end;

procedure TfrmMain.ChangeNotifyRenameItem(Sender: TObject; Flags: Cardinal;
  Path1, Path2: string);
begin
  ScanForChanges(Path2);
end;

procedure TfrmMain.ScanForChanges(FileAndPath: String);
Var
  FolderName, TheFileName, DirToMonitor: String;
  RightNow: TDateTime;
begin
  // Check if created file is of type EXE
  FolderName:=ExtractFilePath(FileAndPath);
  FolderName:=Truncate(FolderName);
  DirToMonitor:='c:\laboratory';
  If Lowercase(FolderName) = DirToMonitor then
  Begin

    If Lowercase(ExtractFileExt(FileAndPath)) = '.exe' then
    Begin

        TheFileName:=Lowercase(ExtractFileName(FileAndPath));
        If FindDuplicate(TheFileName) = false then
        Begin

          With ListViewFiles.Items.Add do
          Begin

                RightNow:=Now; // Get time and date
                Caption:=DateTimeToStr(RightNow);
                SubItems.Add(TheFileName);
                SubItems.Add(UpperCase(FolderName));

          End;

        End;
    End;
  End;
End;
```

Code Extract 5.1
File Monitor Function

The ScanForChanges procedure takes one parameter which specifies the full file name of the file captured by the monitor events. This procedure extracts the file's parent folder and determines if it matches the folder it has been instructed to monitor. If the folders differ, the file is ignored, since it has been created in a different folder. If the folder's indeed match, execution continues. The next step is to check the file's extension. Only files of type 'exe' are considered. When a file matches the criteria set - by being a child of the monitored folder and being an executable - it is stripped of its path and added to the suspect list, along with the current time and date, and folder name mentioned separately. At this point, the file monitor has completed its role.

5.3.2 Network Communication

At the heart of FileMon's communication is the TServerSocket component that comes standard with Borland Delphi.

This non-visual component gives applications networking functionality by establishing a network socket on the host machine. Incorporating this component into one of the project's forms instantly grants it network connectivity. The following code snippet and screenshots demonstrate a sample program that acts as a server that listens on port 2005.

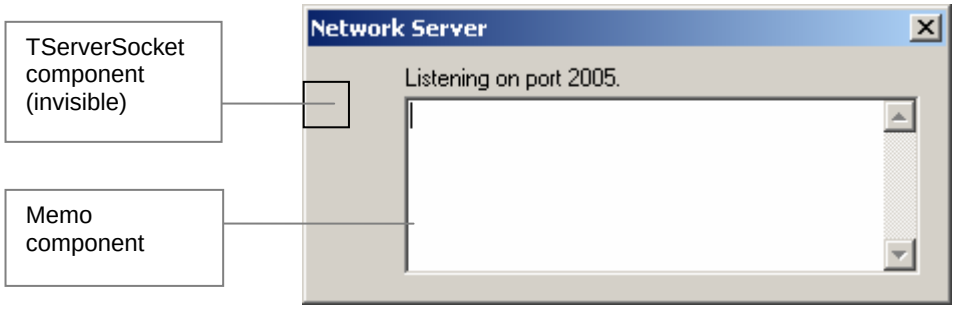


Figure 5.1 Sample Network Server 1
Sample network server listening on port 2005, with no clients.

By running the network server program and starting a telnet session on port 2005, it is possible to communicate with the program and witness the TServerSocket's functionality. From the command console, by typing:

```
telnet localhost 2005
```

Telnet connects to the server program on the specified port (2005). Once connected, typing text into the console will be reflected in the server Memo component. This is shown below:

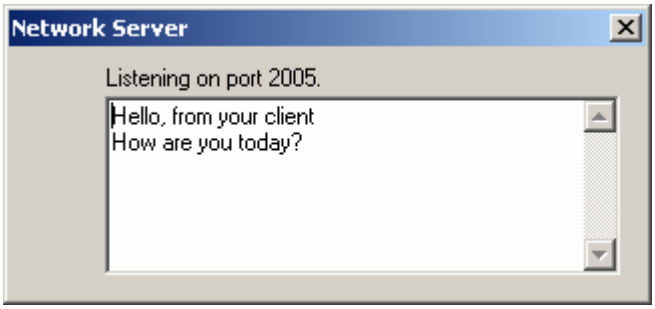


Figure 5.2: Sample Network Server 2
Sample network server listening on port 2005, with one connected client.

The sample network server demonstrated in Figure 5.2 enables a program to open a socket on a certain port and listen for incoming connections from clients.

Returning back to FileMon program, implementing network connectivity can be achieved by using the same source code as the aforementioned sample, creating a built in network server. This code is shown below:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
        Serversocket.Active:=True;
end;

procedure TMainForm.ServerSocketClientRead(Sender:
TObject; Socket: TCustomWinSocket);
begin
        Memo.Text:=Memo.Text + Socket.ReceiveText;
end;
```

Code Extract 5.2 Enabling FileMon Network Connectivity
Code required to allow network connectivity between a server and its clients

The TServerSocket is activated upon creation on the program's main form, and starts listening for incoming connections. The event *ClientRead* is triggered when text is received from the client. The program adds received text to the memo's contents, creating a log of the conversation.

At this point, two issues can be observed. Firstly, the conversation is one way only, from the client towards the server (being FileMon). This is not a major issue, since the server has not been programmed to process what the client sends and subsequently reply back. Secondly and more importantly, receiving text and adding it to a pre-existing stack is not of much use to the server. Understandably, this text is meant to represent commands, embedded in the text that is sent to the server. Therefore, it would be a wise decision to devise a *template* of commands and teach the client and server how to speak the protocol, over which the commands can be exchanged. In addition, in order to alleviate confusion between commands, there must be a method to be used to delineate between individual commands. This can be realised by using artificial terminators, to mark the end of a command when sent by a client. In turn, the server recognises these terminators, and processes the commands individually. A sample command is shown below:

```
Client:          Do you have any suspect files?
```

In the command above, a question mark was used as a terminator to the command (which is in this case, a question). According to the Network Virtual Terminal (NVT) interpretation of US-ASCII control characters, it was noted that the terminator universally in use was the *carriage return, line feed* (CRLF) control character (Tabor, 1995). Therefore, the same control character will be used as a terminator in accordance to the NVT standard. Thus, the server will store all received text from the client in a buffer variable, to construct the full command, until it encounters CRLF, where it would stop accumulating text and process the command adequately.

5.3.3 Protocol Integration

As mentioned in Section 4, inter-agent communication occurs via a common protocol. This protocol consists of numerous commands that are described in the table below. Every time a command is sent, it is either initiated from MalDes or FileMon. The sender associated with the command is also shown.

Command Templates		
Command	**Sender**	**Description**
ANY SUSPECTS?	MalDes	Asks static agent to check if it has detected any suspects
POSITIVE	FileMon	Positive response stating that one or more suspects have been detected
NEGATIVE	FileMon	Negative response stating that no suspects have been detected
NUMBER SUSPECTS	MalDes	Asks static agent to indicate the total number of suspects detected
RESPONSE NUMBER *X*	FileMon	Response sent to NUMBER SUSPECTS query, where *X* represents the total number of suspects detected
LIST SUSPECTS	MalDes	Asks FileMon to present a list of the names of the files it has to report
SUSPECTFILE *filename*	FileMon	Reports the suspect to MalDes, where *filename* is the suspects actual file name
NEXT SUSPECT	MalDes	Confirms receipt of previous file name and asks FileMon for next file in suspect list
NO MORE SUSPECTS	FileMon	States to MalDes that there are no more suspects to report
REMEDY	MalDes	Comprehensive string of text containing the suspect's file and human name, along with its corresponding registry values involved with the suspect's start-up procedure
NO REMEDY FOUND	MalDes	Indicates to FileMon that the file reported is not known to be harmful and may therefore be safely ignored
REMEDY RECEIVED	FileMon	Acknowledgment of receipt of previously sent remedy
QUIT	MalDes	Indicates that MalDes wishes to terminate the connection

Table 5.1: Protocol Commands
Note: all commands are terminated by CRLF

Protocol Processor

As mentioned in Section 5.3.2, the ClientRead event is triggered every time data arrives at FileMon's associated port. This data (in the form of text) goes through several layers of processing to allow the program to adequately decide on how to react. A major portion of the ClientRead code is shown in Code Extract 5.3.

The first step is to capture the received text, which is placed in a global variable that retains its contents throughout the connection's lifecycle.

Due to the nature of TCP/IP, text that is sent between the two agents does not arrive at its destination in one go, but may well be pushed in two or possibly more batches. This requires that the global variable, named *Received*, stacks all received input in an accumulative fashion. This gives a complete picture of all received data. Each time a batch of data is received, the program scans the text for well-known termination characters (previously defined as terminators), such as CRLF. An additional terminator is also defined, which is the forward slash character (/).

This is used for the sole purpose of communicating with FileMon via a manually-initiated telnet session when testing the program during its development stage (since using the *enter* key generates a carriage return character instead of a full CRLF. Once a terminator is detected, the *Received* variable copies itself to a new string called *Command*, and then clears itself. At this point, *Command* contains the previously received text up until the terminator was detected (essentially representing the command that has been received). As a precaution, all received text is automatically converted into uppercase to ease comparison against pre-defined (uppercase) command templates, which are mentioned shortly.

Once the received command has been established, the program can compare it against a list of know command templates, listed in Table 5.1 above. These commands are sent by MalDes and intended solely for FileMon. If a match is found, the corresponding code is executed. Otherwise, the command is ignored until the next command is received. A detailed description of the command templates is presented next.

Command: ANY SUSPECTS

Receiving an *ANY SUSPECTS* command provokes the program to check whether or not it has detected any suspects. Since the suspects are held in their associated list, checking whether the *Count* function of the *Items* property is greater than nil verifies whether any suspects have been detected, in which case a *POSITIVE* response would be sent back. If no suspects have been found, the program would respond with a *NEGATIVE* command.

Command: NUMBER OF SUSPECTS

When a *NUMBER OF SUSPECTS* command is received, FileMon must determine the number of suspects that have been detected during the program's lifetime. As mentioned above, these are held in the suspect list, and calling the *Count* function of the *Items* property yields this number. This number is sent through the socket back to MalDes in the form of a *RESPONSE NUMBER* command template.

Command: LIST SUSPECTS or NEXT SUSPECT

This command requires the program to iterate through its suspect list to report each file detected, which keeps track of how many files have been mentioned so far. Each suspect is reported individually in the format of the command *SUSPECT filename*. MalDes sends the appropriate response back and asks for more suspects, to which the program would repeat the suspect reporting procedure. Once the number of reported files is equivalent to the number of files in the suspect list, then it can be assumed that all suspects have been reported, and the program responds with a *NO MORE SUSPECTS* command.

Command: REMEDY

After a suspect is reported by FileMon, MalDes processes the file name and immediately responds with either a *REMEDY* (called a remedy response). For more details on the format of the remedy response, see Table 5.1 (Command Templates). Remedy responses must be acknowledged by the program by sending a *REMEDY RECEIVED* command.

Command: NO REMEDY FOUND

In the event that no remedy exists for the reported suspect, a *NO REMEDY FOUND* string is sent to FileMon, which must also acknowledge this response in a manner identical to that of which if a proper remedy had been received.

```
procedure TfrmMain.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
Var Command, Temp: String; // Local temporary variable
iIndex: Integer;

begin
 Received:=Received + socket.ReceiveText;
 If ((Received[length(received)] = #10) or (Received[length(received)] =
'/')) then
 Begin
  // Convert to uppercase
  Received:=Uppercase(Received);
  Log('Agent: ' + Received);

  // Fill command buffer, clear received for next arrival of data.
  Command:=Received;
  Received:='';

  // Delete \n (#10) from command to process the following statements
  Delete(Command, Length(Command),1);

  // If this is initial MA contact, respond with positive or negative
  If Command = ANY_SUSPECTS then
  Begin
    // Calculate number of suspects and store in global integer
    NumFilesRecorded:=ListViewFiles.Items.Count;
    // RESPOND WITH + OR -

    If NumFilesRecorded=0 then
    Begin
     ServerSocket1.Socket.Connections[0].SendText('NEGATIVE' + CRLF);
     Log('Server: NEGATIVE');
    End;

    If NumFilesRecorded>0 then
    Begin
     ServerSocket1.Socket.Connections[0].SendText('POSITIVE' + CRLF);
     Log('Server: POSITIVE');
    End;
End;
                          // … more code … //
End;
End;
```

Code Extract 5.3: ClientRead Procedure

5.3.4 Local Administrative Functions

An essential role of FileMon is the ability to execute remedies received by MalDes. This role can be split into three functions, described below.

Process Termination

This function takes a file image name as an argument and consequently kills all corresponding processes matching the file name. It makes reference to the Windows API call named *TerminateProcess* and returns (1) if successful or (0) if it fails.

File Deletion

Being the most straightforward function, file deletion takes a single argument pointing to the name of the file, and commences the deletion process by calling the *DeleteFile* Delphi function.

Registry Value Removal

A preliminary routine is executed before this function starts, where the given argument is segmented into the parent registry key and the actual value name. To extract the key name, all characters are copied from the argument string up until the final backslash character. The remainder of the string represents the value name. These two portions are used to locate the corresponding registry key and value and to remove it from the system registry. This function uses the *DeleteValue* method included in the *TRegistry* Delphi library.

5.3.5 Remedy Application

Upon receipt of a remedy string from MalDes, it must be broken down into several components in order for FileMon to be able to make use of it. The remedy string (remedy package) is made up of a set of substrings, each separated by a pipe delineation character (|). A sample remedy is shown below.

```
REMEDY|SASSER WORM|MSNULL32.EXE|HKEY LOCAL MACHINE|Worms\Sasser
```

To extract the remedy packages contents, the program must first detect the number of pipe delimiters contained within the string. Once this is determined, a loop is started to locate each element in the package and add it to an *infection data array*. Once this is complete, a second routine is initiated which filters the arrays contents among three different remedy lists. These lists individually correspond to the names of processes to terminate, files to delete, and registry values to remove.

5.4 Implementation Details: Mobile Agent

As with FileMon, the MalDes implementation phase was also segmented into several roles. The agent's effective decision-making was a result of intelligence built into the code, as well as dynamic calculation of which hosts to visit and efficient network communication in compliance to a unified protocol.

5.4.1 Built-in Intelligence

MalDes is designed to be equipped with the information necessary to allow it to take swift decisions on the scene. The version implemented, being prototypical, includes a few signatures relating to different infections. This information details the infections' metadata, including the file names and registry values. These signatures are an inseparable part of MalDes, and are carried around with it as it travels from host to host. The agent refers to this information when invoked to investigate whether or not a suspicious file is infected. This is done by dynamically searching its signature array while interrogating a host. A sample signature is shown below:

```
String signatures[][] = {

{"Blaster Worm", "msblast.exe", "HKEY_CURRENT_USER",
"software\worms\blaster"}

                        };
```

Code Extract 5.4
Sample Mal-ware Signature

5.4.2 Itinerary Preparation

To make the system more flexible, MalDes can be given a list of destinations to visit in the form of a list in a text file called "*hosts.txt*". This file must follow a simplified format, where each line represents a Grasshopper agency address. When first executed, the agent will open the associated text file and convert its contents into an array called *hosts*, where each element represents a single line from the text file (hence pointing to a destination Grasshopper address). A sample text file is shown below.

```
Socket://10.11.0.1:7000/Glasgow
Socket://10.11.0.2:7000/Aberdeen
Socket://10.11.0.3:7000/Edinburgh
Socket://10.11.0.4:7000/Fife
```

Contents of Hosts.txt Text File

5.4.3 Network Connectivity

When MalDes arrives at its host, it expects FileMon to be active and ready to respond to its queries. This process is started by initiating a TCP socket on the local host IP address (127.0.0.1) on the FileMon server's associated port (2005). Once the socket is successfully opened, MalDes may freely communicate with FileMon until it decides to terminate the connection.

5.4.4 Protocol Integration

Command Templates

The protocol command templates for MalDes are identical to those of FileMon (see Table 5.1).

Protocol Processing

The first message that MalDes sends is an opening *ANY SUSPECTS*? query. As soon as this is dispatched, the agent enters a loop state where it will gather any messages received until it encounters a carriage return terminator (CR). Detection of this terminator provokes the agent code to treat the received text as a command and extract its contents. After the command is received, the agent looks it up against a list of known command templates, listed in the Table 5.1. If a match is found, then the agent can decide which code to execute next. If the command is unrecognised, it is ignored until the next command is received. A detailed description of MalDes' command templates is presented below.

Command: NEGATIVE

Agent will disconnect from its current host connection and decide where to migrate the next according to its itinerary.

Command: POSITIVE

In this case, the agent will start the interrogation process by querying FileMon for the number of suspects it has noticed. This is accomplished by sending a *NUMBER OF SUSPECTS* command. It expects a reply consisting of a command template and a number, greater than zero.

Command: RESPONSE NUMBER *X*

Upon receipt of this command, MalDes will receive the total number of files that have been caught by FileMon on its host. This number is not used in the actual program code, but instead contributes to the deliberate eloquence of the communication protocol. Once this number arrives, the agent will ask FileMon to list its suspects by sending it a *LIST SUSPECTS* command.

Command: SUSPECT *filename*

As a direct response to being asked to list its suspects, FileMon will start to report the files it has detected in the form of *SUSPECT file name*. The file name is extracted from the message and MalDes attempts to locate the entry within its infection data array. If it does not find the file, it will send FileMon a *NO REMEDY FOUND* reply. On the other hand, if the search finds a match, MalDes instantly packages the remedy into a specially formatted message and dispatches it to FileMon. This reply is prefixed with the command *REMEDY*. A sample message is shown in Section 5.3.5.

Command: REMEDY RECEIVED

Once FileMon receives the previously dispatched remedy, it automatically responds with an acknowledgement message in the form of a *REMEDY RECEIVED* reply. This invokes MalDes to prompt FileMon for any more suspects by sending a *NEXT SUSPECT* command.

Command: QUIT

This command wraps up the interrogation process and causes MalDes to disconnect from its current host and decide where to migrate the next according to its itinerary.

5.4.5 Agent Lifecycle States

While initialising, MalDes takes a snapshot of the current system time on the host it has been summoned on and a list of hosts to visit is retrieved from the local file system.

During migration, MalDes must be able to distinguish how much of its itinerary it has accomplished. This involves keeping a record of the agent's upcoming intentions before the migration process is invoked. A state variable is used to keep track of this piece of information. This variable's value is checked upon every time MalDes lands at a new host, to determine what should be done next.

Prior to dispatch for the first time, the state variable is set to zero. The initial source code checks this variable and recognises that MalDes is currently at its origin and about to start iterating through its list of destinations. At this point, the agent prepares a list of host addresses that were loaded during initialisation. Also, a special array called *reports* is initialised (that initially starts off as blank), which will hold details of each individual host's infection details. Once the itinerary is prepared, MalDes adjusts its state variable to the value of (1) and attempts to migrate to the first destination on its list

Upon arrival at its first host-under-investigation (HUI), the agent checks the value of the state variable - which migrated with the agent's data luggage – and notes that it has been set to the value of (1). This invokes the agent to start its interrogative process with its local static agent counterpart. Once interrogation is completed, a report is compiled and stored in the *reports* array. The agent then ticks the current host off its itinerary and hops to the next destination on its list, while keeping the state variable set to (1).

The aforementioned process is repeated, as the agent visits host after host, until the agent discovers that the current host it has just arrived at is in fact the final destination. MalDes will, as is the case with other hosts, conduct its interrogation and set the state variable to (2). Since the itinerary has been completed, MalDes sets its next destination back to its origin and hops back to its launch pad.

Once back at its starting point, the state variable is checked one last time and noticed to be at the value of (2), indicating that the current host is indeed the origin. A second snapshot is taken of the system time. By comparing the first and second time snapshots, MalDes is able to declare how much time the journey took. The system time mentioned is not the same as well known human time, but instead retrieved from a Windows API call named *GetTickCount* which returns the number of milliseconds that have lapsed since the operating system was last started. This system time used resembles human time in that it is ever-increasing. This time measurement method was used in early versions of MalDes, but was later replaced with the more portable *System.currentTimeMillis* instead. MalDes' function is concluded with report generation, which is conducted by sifting through the *reports* array and stating which hosts were infected, and which were not, giving details of any infections that were encountered.

5.5 Analytical Experiment Implementation

Following the brief description of the experiments to be conducted in Section 4.5 (Experiment Design), the following section shall define the setup of the experiments, detailing how these experiments are to be carried out in a step-by-step fashion. Any program code used to assist in the experiment is shown, as well as any third party software required to complete the experiments.

5.5.1 Experiment 1: Return Trip Timer

Measuring the time it takes for an agent to complete its agenda and return home was accomplished by instructing it to take note the system time at two points during its journey. A snapshot of the system time is taken once when it is about to set off visiting its list of hosts, and again when it completes its checklist and returns back to base. These two measurements are subtracted from each other, giving the return trip time (RTT).

```
// Get current time
time_departure = System.currentTimeMillis();

// Convert time to string
String strTime = (new Long(time_departure)).toString();

// Declare time of departure
log("Agent loaded at TimeStamp: " + strTime);
// ...
// other code
// ...
// Get current time (again) on return to base
time_return= System.currentTimeMillis();

// Convert time to string
String strTimeReturn = (new Long(time_return)).toString();

// Declare arrival
log("Agent is back. TimeStamp now: " + strTimeReturn);

// Subtract the two measurements
long totaltime = time_return - time_departure;

// Convert result to string
String strTotal = (new Long(totaltime)).toString();

// Declare return trip time
log("Work completed in " + strTotal + " milliseconds");
```

Code Extract 5.5
RTT Calculation Code

The RTT experiment does not require any extra software in order to be carried out. MalDes is loaded at the agency and it will automatically carry out the experiment as part of its duties. Upon return back to base, the RTT will be displayed.

As mentioned in Section 4.5, this experiment will be conducted four times, with a different number of hops varying from one to four. The aim of this experiment is to prove that MalDes takes longer to complete its journey as the number of hops increases.

5.5.2 Experiment 2: Host Performance Timer

This experiment does not require any third party tools. The aforementioned RTT test is conducted on two sets of computers (superior and inferior groups). Since this experiment is in fact a repetition of the RTT test, no additional code is required for it to be conducted. The basic aim of this experiment is to show how the trip times noticeably vary according to host hardware specifications.

5.5.3 Experiment 3: Agent Size Measurement

Slightly more complicated, this experiment requires a third party tool named Ethereal. By constricting the number of machines in the network laboratory (like disconnecting the Internet gateway), it is possible to isolate a group of machines from the outside world and scrupulously study the packets that are exchanged during communication. Throughout this experiment, the number of hosts is kept the same. Starting off with a small number of hosts would be wise (two hosts would suffice initially), since the aim is to minimise the amount of redundant packets captured, such as broadcasts, namely NetBIOS-specific announcements

which may in many cases waste valuable bandwidth (Evans, 2002). Using the third party tool, Ethereal, it is possible to sniff the exchanged packets between the two workstations (by running the tool on the destination host). Once the migration and return of MalDes is finished, data captured by Ethereal needs to be studied and filtered of any unnecessary packets, leaving the bare grasshopper conversation. Bearing in mind that the maximum frame payload is 1500 bytes (Fairhurst, 2001), it is possible to measure the number of frames sent from the origin to the destination in order to transfer MalDes. Hence, it is possible to calculate the size of the data transmitted during migration and note the frame time stamps to determine the total bandwidth consumed. Since MalDes made only one hop to a single destination, collected report data would correspond to a single host. This measurement would represent a *minimal size* mobile agent. Next, the number of hops is gradually increased by introducing new hosts to the network and adding them to MalDes' itinerary. Each time a host is added, the migration process is repeated and an Ethereal packet capture session is conducted simultaneously. Thus, this experiment allows measurement of the total size of the transported agent as it traverses through the network, swelling in size as it collects more report data.

5.5.4 Experiment 4: Internet RTT Test

As described in Section 4.5, this experiment is basically a RTT test conducted over the Internet.

5.6 Summary

To wrap up the Implementation phase, it must be emphasised that the Grasshopper agent platform provides a rich spectrum of features for mobile agent development. The learning curve involved to gain adequate competency was not at all steep, mainly due to the documentation provided being very in-depth and comprehensive.

Splitting the FileMon agent into several development stages according to Wier's strategies (2002) enabled each code portion to be crafted according to requirements, because using the software milestone strategy greatly simplified the task of getting the program airborne. Using a third party software component for the file monitor vastly reduced development time, by providing the precise requirements that fitted gracefully into the program structure, allowing more time to be spent focusing on other aspects of the project. The mechanism used to analyse exchanged network messages during inter-agent communication was a result of many days of experimentation, but getting the final version working was well worth all the effort. The protocol used between mobile and static agent was created to be simple and eloquent, and was kept unchanged throughout the development process to eliminate any potential problems that could arise.

With mobile agent development, the same 'divide and conquer' approach was used. Incorporation of multiple infection signatures into the code started off as an experiment, but later, as the progress was made, turned out to be a milestone in the software's design. In the beginning, agent migration was tested between two machines. After the core functionality of MalDes was completed, destination host addresses were built into the code as is the case with infection signatures. This later proved quite inconvenient, and led to the introduction of a more dynamic approach by loading the addresses from a locally-based text file. Discovering that a host was infected was of no use if no feedback mechanism was put in place. This led to the report collection method design that was later, after intensive testing, was merged with the current version of the agent.

Although the project went through many stages of experimentation during project - like determining the best approach to allocate MalDes its itinerary – some sort of post-production testing was required. These types of experiments would uncover facts about the overall system by giving measurements for different metrics that could later be used in showing the overall project findings.

6 Results

6.1 Introduction

Subsequent to the thorough design of the experiment strategy in the previous chapters, the following section attempts to briefly recap each experiment's specification, and then present the corresponding results in a descriptive fashion. For these experiments, it was ensured that all static agents had no suspects to report to prevent any extra delay induced from inter-agent interrogation.

6.2 Return Trip Timer (RTT)

This experiment measured the amount of time a mobile agent took to complete its itinerary and return back to its launch pad. The experiment was conducted with a different number of hops each time (from one to four). The aim was to prove that as more hosts were added to the itinerary, the RTT should increase. The results are shown below.

RTT Test	
Hop Count	**RTT (seconds)**
1	1.062
2	2.063
3	3.079
4	4.094

Table 6.1:
RTT Test Results for 1, 2, 3 and 4 Hosts

Table 6.1 shows the results for 1, 2, 3 and 4 hosts. It can be seen that the average for a single hop is 1.062 seconds, for 2 hops it is 2.063 seconds, for 3 hops it is 3.079 seconds and for 4 hops it is 4.094 seconds. Thus, the variation is linear, where the delay is fairly static over a number of hosts.

As expected, the RTT proportionally increases in relation to the number of hops that are made. Thus, it can be assumed that if the same experiment was conducted with fifty hops, the RTT would expectedly be greater than that of another experiment carried out with for example, forty-nine hops.

6.3 Host Performance Timer

For this experiment, two groups of workstations were prepared. The first group, named the Superiors, consisted of eight Pentium 4 (2800 MHz) machines with 512MB DDR memory running Windows XP. The second group, named the Inferiors, contained eight Pentium 3 (1000 MHz) machines with 128MB SDRAM also running Windows XP. A RTT test was run on both groups and the timings were logged in the following two tables.

Superiors	
Hop Count	RTT (seconds)
1	1.047
2	2.297
3	3.328
4	4.375
5	5.360
6	6.375
7	7.351
8	8.453

Table 6.2
Superiors RTT Test Results

Inferiors	
Hop Count	RTT (seconds)
1	1.297
2	2.406
3	3.593
4	4.563
5	5.703
6	6.172
7	7.765
8	8.843

Table 6.3
Inferiors RTT Test Results

The timings in tables 6.1 and 6.2 indicate that the RTT is directly proportionate in relation to the number of hops made. The differences between the two groups are noticeable and are justified by the performance inferiority of the second group. By analysing the graph next (Figure 6.1), the slight differences between the two competing groups can further be visualised.

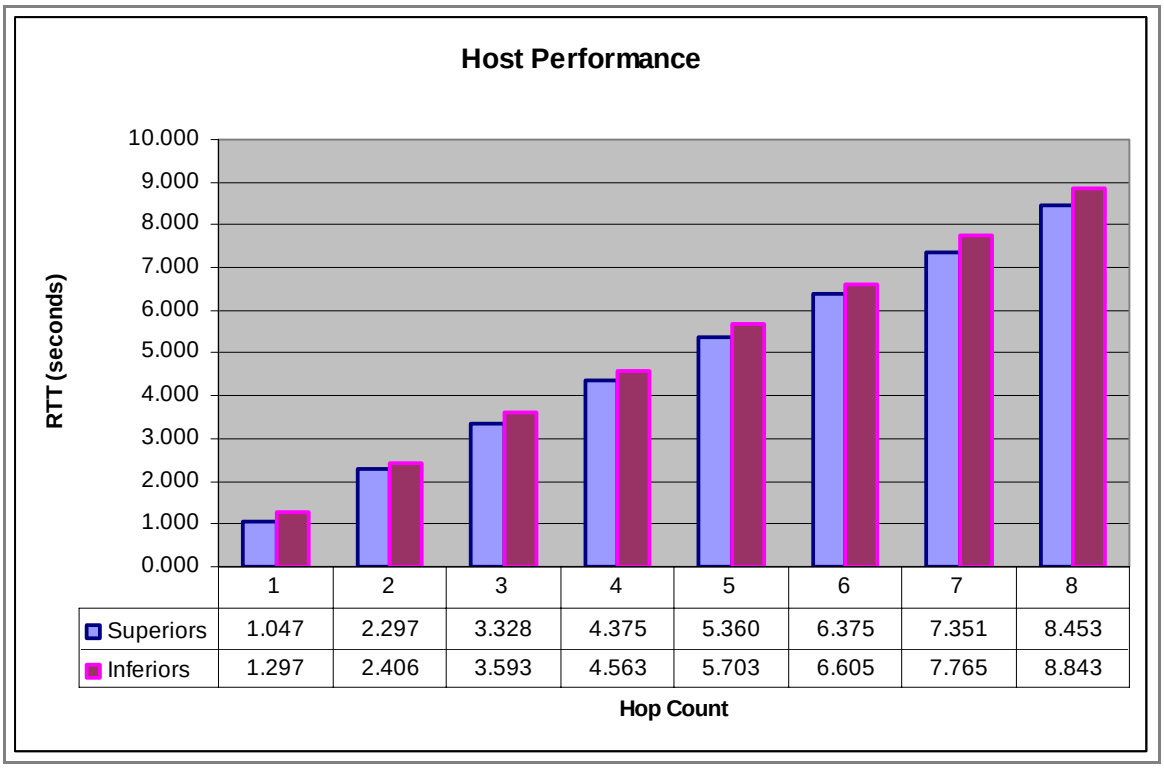


Figure 6.1: RTT Host Performance Test

The Superior group timings were always quicker than their inferior counterparts, usually by a constant average of approximately 274 milliseconds. After analysing the results above, it can be concluded that workstation hardware specifications have a slight yet noticeable effect on the total trip time. Effectively, MalDes was able to commute quicker through the faster hosts compared to the slower ones.

6.4 Agent Size Measurement

Referring to Section 5.4.5 (Agent Lifecycle), it was mentioned that prior to MalDes setting off on its journey, a special array named reports is initialised as blank. As the agent traverses through the network, its reports array gradually increases in size as more information is accumulated. This experiment was run to investigate this issue, which thus far, lacked evidence. A program called Ethereal was used to eavesdrop on packets communicated between hosts as the agent navigated its way through them. The data captured by Ethereal was then analysed and a summary of results was generated to create a basis for evidence, further elucidating the reports array accrual incident. The results of this experiment are shown next in Table 6.4.

Hop Number	Conversation Size (bytes)
1	4664
2	4712
3	4745
4	4781
5	4797
6	4810
7	4818
8	4846

Table 6.4
Conversation Size Results as Hops Increase

Conversation Size represents the communication that occurred between the sending and receiving host as MalDes was in the process of travelling. A larger conversation size indicates that more data was exchanged in order to transport MalDes to the next destination host. As mentioned above, the reports array accumulates more data as more hops are encountered. To support this statement, from Table 6.4 notice that conversation size increases as more hops are made. The conversation between the launch pad agency and the first hop was 4662 bytes in size. Between hop 1 and hop 2, the conversation size dramatically soared to 4719 (an increase of 57 bytes). For the remaining hops, the conversation size inter-hop increase average was 4 bytes. This is further clarified in the chart in Figure 6.2.

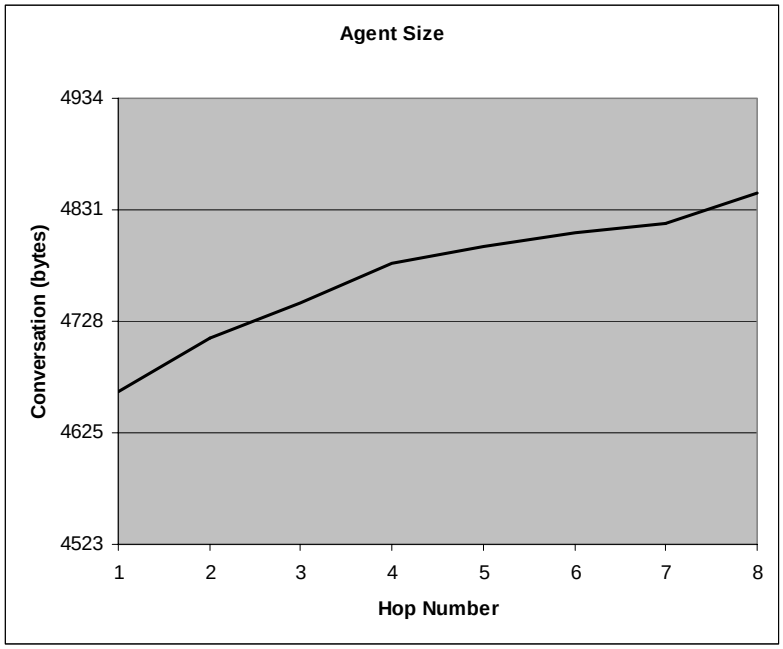


Figure 6.2: Conversation Size Chart
Chart shows conversation size increase as more hops are made.

7 Evaluation

The project is drawn to a close by presenting critical analysis of the work undertaken and more importantly to examine whether the advantages of adopting the P2P architecture over the centrally coordinated version are indeed worthwhile. Discussed next are the project objectives and how well they were met, as well as its strengths and weaknesses. Experimentation results are summarised, followed by future work recommendations. The project concludes by acknowledging that although obstacles still exist, such as inter-agent authentication, a P2P mobile agent based architecture can offer an extremely adaptable, robust and powerful environment that can contribute to the field of IDSs.

7.1 Evaluation of Objectives

Recalling the originally stated aims of this project in Section 1.1, the four main objectives that were declared have all been met. A critical review in the field of using mobile agents for IDSs was conducted in Section 3. The following two sections focused on evaluating and implementing selected elements of functionality that allowed mobile agents to fit into the P2P IDS model (fulfilling the framework design and implementation objective). Section 6 and 7 shifted to document the project's finding and to conduct critical analysis. When comparing between adopting a mobile agent-based P2P architecture and a centrally co-ordinated version, the advantages become obvious. Mobile agents provide autonomy, dynamic adaptation, intelligence and portability that when merged with the field of IDSs, can provide an effective arsenal to combat anomalies such as mal-ware. The mobile agent approach's performance has been evaluated and compared with traditional mal-ware detection/immunisation systems.

At the moment, the monitoring capabilities of FileMon are limited to capturing newly created executables. Existing files that are modified easily slip past the monitor and go unnoticed. This flaw is due to the way the Delphi VCL component in use handles Windows events, and can be rectified by adding event handlers that would capture specific Windows file system messages. On the other hand, local administrative functions such as mal-ware process termination work smoothly. A possible advantage would have been gained if Maldes instructed Filemon to restart the Explorer Shell application to terminate any malicious processes residing under the Explorer tree. Additional intelligence for FileMon is certainly needed, at least to memorise any given remedies for future reference.

From a system security point of view, none has been integrated yet. FileMon will interact directly with any process or human that successfully bonds a connection on the correct TCP port, and can speak the protocol required. This prototypical design assumes that any TCP/IP communication shall be from a trusted source, and is not acceptable in the real world. A method of authentication must be introduced, possibly using Challenge Handshake Authentication Protocol (CHAP).

Taking a look at MalDes, being a prototype, only a handful of mal-ware signatures were built into the agent. The approach of randomly feeding the mobile agent a group of signatures would not prove very effective in an IDS environment. Instead there should be a mechanism where the investigator on the launch pad host could select specific profiles to match the investigation that the agent is about to set off on.

Another approach could be to adopt a similar idea to the one proposed by De Queiroz et al. (1999) where an external set of distributed databases would contain a history of all know signatures, and Maldes would remotely query the databases if the given suspect is not within its intelligence barrier. A possible speed tune up for return times could be achieved if Maldes could intelligently construct a logical map of hosts on its agenda and spawn several other identical agents that set off on different routes and meet back at the launch pad.

7.2 Main Findings

From experimentation was derived a collection of useful results that could be used to support the claim that mobile agents could be utilised in IDSs (Section 3.2.6). On average, the time a mobile agent took to migrate to one host and back was approximately 1.062 second. This increased linearly as the number of hops grew, becoming 2.063 for 2 hops and 3.079 for 3 hops.

After comparing two groups of computers that differed in processor generation and amount of memory, it was noticed that host hardware specifications had a minimal effect on return trip times so long as the host was able to run the operating system smoothly. Over a fixed number of hosts, MalDes completed the trip on the superior group 274 milliseconds faster than the inferior group (on average).

Supporting the claim made in Section 5.4.5, it was proven that as MalDes makes more hops, it increases in size due to the accumulation of report-related data within its variables. The average increase in agent size was 4 bytes per hop.

The Internet RTT test could not be conducted due to lack of time and resources, such as a direct connection to the Internet without going through NAT and router port restrictions. These factors acted as obstacles in the process of attempting to migrate agents over the Internet.

7.3 Conclusions

The purpose of this project was to answer the question of whether IDSs for the detection of mal-ware could be enhanced with the usage of an agent-based approach in P2P systems. It started off by taking a bird’s eye view on the current state of mobile agent IDS applications (especially P2P) and it was noticed that little research has been initiated recently in this field. Different researchers in this arena came up with similar approaches to applying mobile agents in IDSs.

A gap was noticed for contribution to this field in designing and implementing a prototype system that would utilise intelligent mobile agents in protecting systems against mal-ware. DIMA’s design introduced the idea of static agents (much similar to proposals in Section 3.2.7) that would monitor their hosts for suspicious executables, and mobile agents that had built-in intelligence used in identifying malicious content and instructions on how static agents would handle each file detected. To evaluate the project, a group of experiments were devised and conducted to test DIMA and produce statistical results that would be complementary in answering the initial research question.

Ramachandran and Hart proposed a system where mobile agents collaborated in a P2P structure. In this work the model has been dissected and a specific branch of IDSs has been examined, namely mal-ware detection and elimination.

An important element of this work is to investigate the possibility of creating a mal-ware detection/mitigation system using mobile agents. This has resulted in a paper which was accepted in the ECIW 2005 conference. It presented a possible model of a mobile agent which could investigate malicious activities, and is partly based on the work conducted in this thesis.

DIMA was based on a P2P approach due to the various drawbacks of adopting a hierarchical structure, as detailed in Section 3.2.3. The project focused on a specific branch of intrusion detection, and avoided covering a broad range of IDS functions that would result in less detail covered per aspect, as with several systems reviewed in Section 3.2.7. Dozens of mobile agents could be devised for a variety of tasks and allowed to roam the network autonomously performing IDS tasks without the dependence on external entities.

A prototype was successfully implemented and tested on networks made up of up to 16 hosts. Completing this prototypical phase (which delved into a specific part of intrusion detection – mal-ware) established solid evidence that mobile agent's could indeed be of use in the field of IDSs.

7.4 Recommendations for Future Work

Overall, the implemented system showed good potential for expansion and improvement and it would provide a reasonable base for future work, suggestively. DIMA, overall, is far from complex in concept, and many improvements could be suggested to enhance the overall system specification. Currently, the mobile agent portion of DIMA carries built-in mal-ware signatures that are hard-coded into the agent. As an improvement – possibly using XML – mal-ware signatures could perhaps be loaded from an external source, to suit the type of environment the agents would run in. This would contribute to DIMA's flexibility, and poses as a step forward in functionality, empowering the overall system.

Also, FileMon - that currently monitors only Windows executables – was written in Pascal. Porting the program to another language such as Java would allow it to be run on any operating system, allowing it to be used more widely. In addition, if the need arose to run several instances of DIMA on neighboring networks, it would be highly of use if Mobile agents's would meet occasionally to exchange mal-ware signatures and *learn* from each other's knowledge.

Blueprints by researchers such as De Queiroz, et al. draw to attention interesting designs for IDSs using mobile agents, where the system starts off compact with one parent agent that intelligently spawns an army of diverse multi-purpose agents capable of defending hosts against anomalies, while the mother agent defends itself from invasion by evading intrusions by migrating away from threats. Such proposals could be used as a basis for future projects, where the design could be overhauled to step away from the centralised approach and have backup mother agents on standby, as well as bodyguard processes that defend their parent agents. Ramachandran and Hart innovated a design that draws attention by clustering hosts into virtual neighborhoods that watch out for each other, using a distributed decision making system to determine the health of a specific site. A possible amendment to this design could involve using distributed decision making amongst mobile agents that may roam in teams and take decisions on the spot instead of consulting neighboring sites. The choice of mobile over static agents has numerous advantages, firstly with system resource conservation, where mobile agents are resident in memory for a limited amount of time.

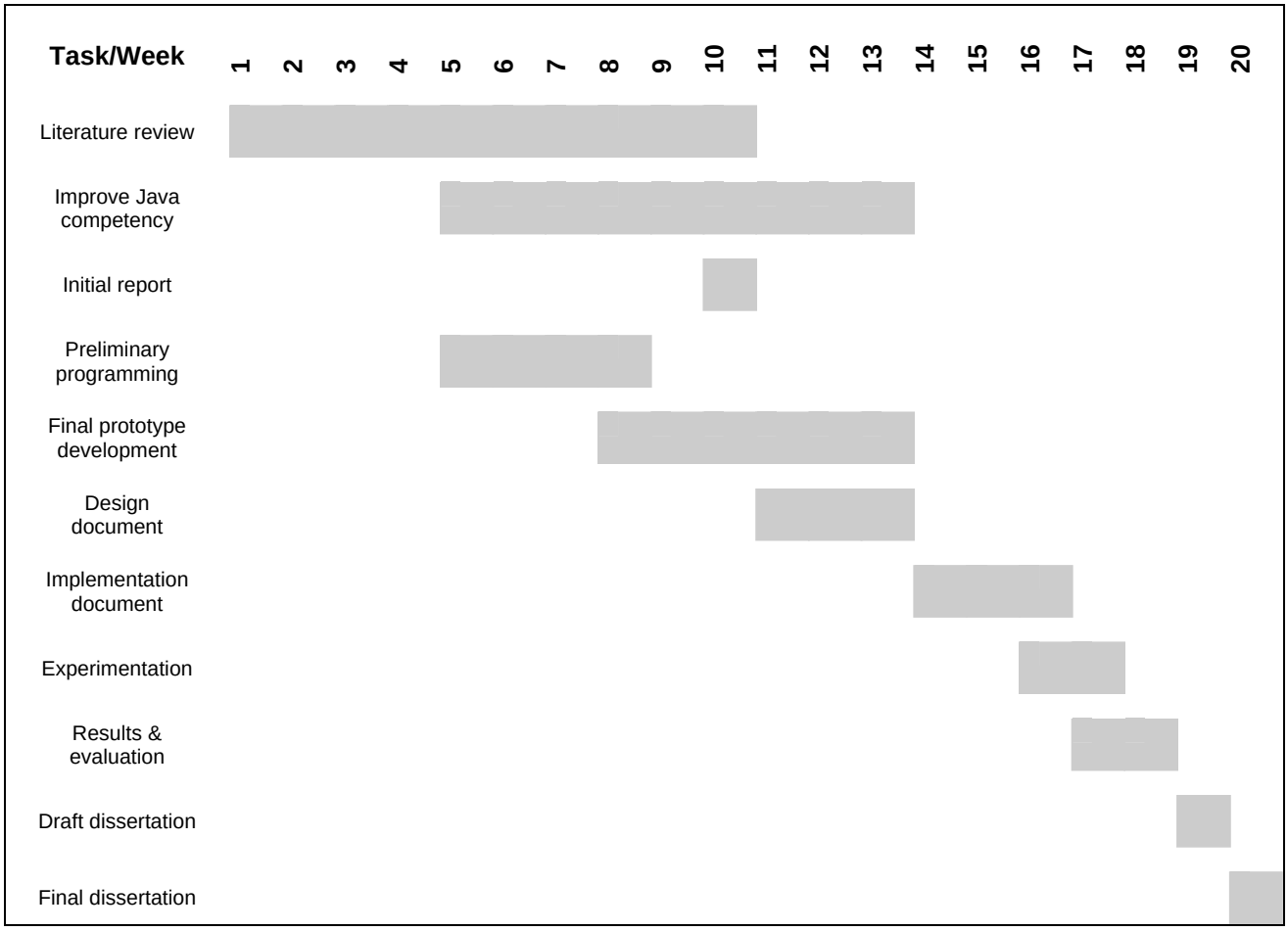
Secondly, mobile agents have a bigger chance of evading being sensed and attacked by intruders, since their host visits normally take fractions of a second. Third of all, mobile agents can be more intelligent than their static counterparts. Due to their mobility, they can pursue the information they require and obtain it on a local host basis, much quicker than static agents that may resort to client/server communications which can be eavesdropped upon and corrupted by spoofing attackers.

The future could bring to surface the prevalence of mobile agent based IDSs that collaborate with other security elements such as personal firewalls and antivirus software. The merging of mobile agent environments into today's operating systems would take it one step closer to fulfilling this vision. Worldwide, in different organizations and firms, mobile agents could be adopted to automate a variety of everyday policy-related tasks, such as ensuring Windows automated updates are enabled without having the individual user carry them out manually, minimizing the margin for error.

Migrating agents over the open Internet or private networks would be possible, once they are globally accepted in concept. A possibility would be designing lightweight agent skeletons that could have support for functionality plug-ins would be a major advantage. For example agents could be loaded with mal-ware detection modules or other packages that allow them to have multiple purposes.

8 Appendices

8.1 Appendix 1: Project Management (GANTT Chart)



8.2 Appendix 2: Project Management (Project Diaries)

NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Hussain Al Sebea **Supervisor: Bill Buchanan**

Date: 24 November 2004 **Last diary date: <none>**

Objectives:

Notes:
- First meeting with supervisor.

Objectives:
- Start familiarisation phase with Grasshopper mobile agent environment (read documentation)

Discussed main software aspects:
- Write a mobile agent that migrates from one host to another
- Write a static agent that performs local host functions and communicates with the mobile agent

Progress:

Supervisor's Comments:

NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Hussain Al Sebea **Supervisor: Bill Buchanan**

Date: 12 December 2004 **Last diary date: 24 November 2004**

Objectives:

- Gather more research papers and publications from online resources and library
- Start reviewing literature and following thematic approach with all taken notes
- Gain competency in Java programming (online tutorials and reference books)
- Create a prototype system consisting of mobile and static agents (basic framework)
- Read Grasshopper documentation (Basics, Programmer's and User's guides)

Progress:

- Started reading Grasshopper documentation including programmer's guide
- Experimented with Grasshopper sample agents (creating, moving and deleting agents)
- Installed NetBeans Java IDE and enabled Grasshopper library
- Supervisor explained static and mobile agent concepts (inter-agent communication suggestions)
- Discussed with supervisor a sample static agent scenario for IDS (SNMP example)
- Mentioned experimentation using agent timer method (for project evaluation)

Supervisor's Comments:

NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Hussain Al Sebea **Supervisor: Bill Buchanan**

Date: 25 February 2005 **Last diary date: 12 December 2004**

Objectives:

- Define experiments strategy for system (referring to supervisor's comments last meeting)
- System needs to be more IDS oriented, requires design improvements

Progress:

- Got familiar with Grasshopper internal workings (agencies and agent management)
- Drafted literature review and theory sections and sent to supervisor for review
- Developed a mobile agent that migrates to one host and communicates with a static agent that retrieves local file data (file's MD5 checksum). The mobile agent displays the retrieved checksum to the user

Supervisor's Comments:

NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Hussain Al Sebea **Supervisor: Bill Buchanan**

Date: 4 March 2005 **Last diary date: 25 February 2005**

Objectives:

- Create a second prototype that is more IDS oriented that targets mal-ware detection and elimination. Implement new design using improved communication protocol.
- Draft design section

Progress:

- Devised an improved system design and added to IDS features
- Discussed different experiment types (return trip timer, host performance, agent size, Internet migration) and each one's relevance
- Finalised experiment types to measure mobile agent performance and how they shall be conducted

Supervisor's Comments:

NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Hussain Al Sebea **Supervisor: Bill Buchanan**

Date: 11 March 2005 **Last diary date: 4 March 2005**

Objectives:

- Finalise prototype development
- Finish design document

Progress:

- Work well under way with development of mobile and static agent according to new specifications
- Mobile agent can now migrate to multiple hosts and return with a report of the infections it found (if any)

Supervisor's Comments:

NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Hussain Al Sebea **Supervisor: Bill Buchanan**

Date: 18 March 2005 **Last diary date: 11 March 2005**

Objectives:

- Draft implementation chapter
- Incorporate experiments into mobile agent code

Progress:

- Static agent can apply remedies and remove mal-ware according to the mobile agent's instructions
- Design document draft completed and sent to supervisor for review

Supervisor's Comments:

NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Hussain Al Sebea **Supervisor: Bill Buchanan**

Date: 6 May 2005 **Last diary date: 18 March 2005**

Objectives:

- Conduct designed experiments and record results

Progress:

- Coded mobile agent with experiment functions
- Drafted implementation chapter and sent to supervisor
- Paper accepted at ECIW

Supervisor's Comments:

NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Hussain Al Sebea **Supervisor: Bill Buchanan**

Date: 11 May 2005 **Last diary date: 6 May 2005**

Objectives:

- Draft results chapter

Progress:

- Completed experiments and recorded results
- Ethereal was used to measure the size of inter-host conversations to transport the mobile agent
- Attempted to test mobile agent migration across Internet but failed
- Internet migration experiment skipped (due to failure) because of lack of direct connectivity between residential workstation and Napier laboratory in Edinburgh

Supervisor's Comments:

NAPIER UNIVERSITY

SCHOOL OF COMPUTING

PROJECT DIARY

Student: Hussain Al Sebea **Supervisor: Bill Buchanan**

Date: 20 May 2005 **Last diary date: 11 May 2005**

Objectives:

- Proof read dissertation

Progress:

- Conclusion and evaluation draft completed and sent to supervisor for review
- Compiled first dissertation draft and sent to supervisor for review
- Revised literature and theory sections

Supervisor's Comments:

8.3 Appendix 3: Source Code (MalDes)

Maldes.java

```
/* By Hussain Al Sebea 2005*/

import javax.swing.JOptionPane;
import de.ikv.grasshopper.communication.GrasshopperAddress;
import java.io.*;
import java.net.*;
import java.awt.event.*;
import java.util.*;

public class MalDes extends de.ikv.grasshopper.agent.MobileAgent{
int state;
String md5;
boolean infected = false;

long time_departure;
long time_return;

int infectioncount;
// Hosts: List of destination grasshopper ip addresses, to be loaded from textfile
String[] hosts;

// reports array: will be populated with info on whether a host is infected of not, its ip address, its infections if any
// let the first field be infected (true or false, in text form), the second field is the ip address of the machine we reported, the third is the first malware name detected
// start off blank, create an array of 15 rows, which can therefore report on 15 different hosts
String reports[][] ={    {"","","",""} ,   {"","","",""} ,   {"","","",""} ,   {"","","",""} ,
{"","","",""} ,
{"","","",""} ,   {"","","",""} ,   {"","","",""} ,   {"","","",""} ,   {"","","",""} ,
{"","","",""} ,
{"","","",""} ,   {"","","",""} ,   {"","","",""} ,   {"","","",""} ,   {"","","",""} ,
{"","","",""} ,
{"","","",""} ,
};

int nexthop = 0; // Keeps index of how many hops we"ve done. Also used to see which line to use from hosts string aray, as it is incremented after each hop

String infections[][] = {   {"Blaster Worm","msblast.exe","HKEY_CURRENT_USER","software\\microsoft\\windows\\currentversion\\run\\process msblast"} ,
 {"Sasser Worm","msprc.exe","HKEY_LOCAL_MACHINE","software\\microsoft\\windows\\currentversion\\run\\process msprc"}  ,
 {"Null Worm","msnull32.exe","HKEY_CURRENT_USER","software\\microsoft\\windows\\currentversion\\run\\process msnull"}  };

////////////////////////////////////////////////////////////////////////////////
// the following classes/functions are for the IP address auto load into variable mechanism //
////////////////////////////////////////////////////////////////////////////////

public class FileToArray {

```

```
43 public FileToArray() {
44 }
45
46 public String[] doIt(){
47
48 String fileName = "c:\\hosts.txt";
49 ArrayList lineList = new ArrayList();
50 BufferedReader reader = null;
51
52 try{
53 reader = new BufferedReader( new FileReader(new File(fileName)));
54 }
55 catch(FileNotFoundException fnfe){
56 System.out.println("Unable to find the file: " + fileName);
57 }
58
59 try{
60 String currentLine = "";
61 while( (currentLine = reader.readLine()) != null)
62 lineList.add(currentLine);
63 }
64 catch(IOException ioe){
65 ioe.printStackTrace();
66 }
67 finally{
68 try{
69 if(reader != null)
70 reader.close();
71 }
72 catch(IOException ioe) { /* Do Nothing */ }
73 }
74
75 return (String[])lineList.toArray(new String[0]);
76 }
77
78 }
79
80 public void init(Object[] creationArgs)
81 {
82
83 state = 0;
84 log("Dynamic Detection and Immunisation of Mal-ware using Mobile
Agents Prototype");
85 log("Starting...");
86
87 // Get current time in milliseconds
88 // This is to keep track of what time the agent left.
89 // We need to do the same when the agent returns
90 time_departure = System.currentTimeMillis();
91
92 String strTime = (new Long(time_departure)).toString();
93
94
95 log("Agent loaded at Timestamp: " + strTime);
96
97 // Load IP addresses of destinations from text file:
98
99 log("Loading list of IP addresses");
100 FileToArray fileToArray = new FileToArray();
101 hosts = fileToArray.doIt();
```

```
102
103 }
104
105 private void mymethod()
106 {
107
108
109 // mymethod is executed at remote PC
110
111
112 String    ANY_SUSPECTS            = "ANY SUSPECTS?";
113 String    NUMBER_SUSPECTS         = "NUMBER SUSPECTS"; // Asked by mobile
agent
114 String    RESPONSE_NUMBER         = "RESPONSE NUMBER"; // Prefix to integer that
specifies how many files are suspicious
115 String    LIST_SUSPECTS           = "LIST SUSPECTS";
116
117 String    REMEDY_RECEIVED         = "REMEDY RECEIVED";
118 String    NEXT_SUSPECT            = "NEXT SUSPECT"; // Used as kind of ACK
when more than one file name is being reported.
119 String    REMEDY_PACKAGE          = "REMEDY"; // Used as kind of ACK when more
than one file name is being reported.
120 String    NEGATIVE                = "NEGATIVE";
121 String    POSITIVE                = "POSITIVE";
122 String    SUSPECT                 = "SUSPECTFILE"; // Prefix to filename. I.e.
SUSPECT msnull32.exe
123
124 String    NO_REMEDY_FOUND         = "NO REMEDY FOUND"; // This file is safe (sent
out to server)
125
126 String    NO_MORE_SUSPECTS        = "NO MORE SUSPECTS";
127 String    QUIT                    = "QUIT";
128
129 // declaration section:
130 // tcpClient: our client socket
131 // os: output stream
132 // is: input stream
133
134 Socket tcpSocket = null;
135 DataOutputStream os = null;
136 DataInputStream is = null;
137
138 // Initialization section:
139 // Try to open a socket on port 2005
140 // Try to open input and output streams
141
142 try {
143 // connect to localhost:2005
144 tcpSocket = new Socket("127.0.0.1", 2005);
145 os = new DataOutputStream(tcpSocket.getOutputStream());
146 is = new DataInputStream(tcpSocket.getInputStream());
147 } catch (UnknownHostException e) {
148 System.err.println("Don"t know about host: hostname");
149 } catch (IOException e) {
150 System.err.println("Couldn"t get I/O for the connection to: localhost
on this machine");
151 }
152
153 // If everything has been initialized then we want to write some data
```

```
154 // to the socket we have opened a connection to on port 2005
155
156 if (tcpSocket != null && os != null && is != null) {
157 try {
158
159 // 1. Ask if there are any suspects
160 os.writeBytes(  ANY_SUSPECTS + "\n");
161 // default to infected status of false
162
163 reports[nexthop][0] = "false"; // log that this host is NOT infected
164 reports[nexthop][1] = hosts[nexthop]; // log the ip address
165
166
167 // keep on reading from/to the socket till we receive CRLF from the static agent (server)
168 String responseLine;
169 while ((responseLine = is.readLine()) != null) {
170 System.out.println("Server: " + responseLine);
171
172 // We have received a response from the remote server.
173 // Analyse it and respond accordingly
174
175 // 2. Response will be either + or -
176 if (responseLine.indexOf(NEGATIVE) != -1) {
177
178 os.writeBytes("OK, I"m leaving now. Best of luck in the future\n");
179 infected = false;
180
181 // make a report
182
183 reports[nexthop][0] = "false"; // log that this host is infected
184 reports[nexthop][1] = hosts[nexthop]; // log the ip address
185
186 break;
187 }
188
189 if (responseLine.indexOf(POSITIVE) != -1) {
190
191
192 os.writeBytes(  NUMBER_SUSPECTS + "\n");
193 infected = true;
194 }
195
196 // 3. Next response will be number of suspects
197
198 if (responseLine.indexOf(RESPONSE_NUMBER) != -1) {
199
200 // ask for list of each file...
201 os.writeBytes(LIST_SUSPECTS + "\n");
202 }
203
204
205 if (responseLine.indexOf(SUSPECT) != -1) {
206
207 // 5. Search infections array for given exename
208
209 // BEWARE of using double \\ to represent one backslash within code...
210
211 String buffer = responseLine;
212 // Find the space
```

```
213 int theindex = buffer.indexOf(" ");
214 buffer    =   buffer.substring(theindex + 1); // Do not include space preceding filename
215
216
217
218 // this boolean is important. With it we can see if we found a remedy,
219 // if not the file is considered, in this case, harmless (safe) and can be removed from the suspects list on the server
220 boolean foundremedyforthisfile = false;
221
222
223 for (int counter = 0; counter < infections.length;  counter++)
224 {
225
226 if (infections[ counter ][1].equals(buffer)) // Compare strings, cannot compare with ==, see page 513, section 11.2 (Java How To Program)
227 {
228
229 // Use COUNTER variable to know index of remedy in infections array
230 os.writeBytes(REMEDY_PACKAGE + "|" + infections[ counter ][0] + "|" + infections[ counter ][1] + "|" + infections[ counter ][2] + "|" + infections[ counter ][3] + "\n");
231
232 // make log of this infection
233 // nexthop is the current hop
234
235 reports[nexthop][0] = "true"; // log that this host is infected
236 reports[nexthop][1] = hosts[nexthop]; // log the ip address
237 reports[nexthop][2] = infections[ counter ][0]; // human name of worm/infection
238
239 // increment filecount
240 infectioncount++;
241
242 // set this to true, because if false (by declaration default), we will tell the server (belo)w that this file is harmless
243 foundremedyforthisfile = true;
244 // Break or loop will continue
245 break;
246
247 }
248 else
249 {
250
251 //not equal - do nothing
252 }
253
254 }
255
256
257 // end of infection lookup. if execution reaches here (without breaking above)
258 // and foundremedyforthisfile = false, this means we didn"t find a remedy for this file
259 // we will consider it safe for this prototype agent
260 if (foundremedyforthisfile == false)
261 {
262
263 // send out code to indicate this file is safe
264 // send in form of a "noremedy", so server can process next file if any
265 os.writeBytes(NO_REMEDY_FOUND + "|" +   buffer  + "\n");
```

```
266
267 }
268
269 // Once server receives this, it will acknowledge by saying it has received the remedy
270 }
271
272 // Await remedy ACK, and see if there are any more suspects
273 if (responseLine.indexOf(REMEDY_RECEIVED) != -1) {
274
275 // ACK received, ask if there are any more suspects
276 os.writeBytes(NEXT_SUSPECT + "\n");
277 }
278
279 if (responseLine.indexOf(NO_MORE_SUSPECTS) != -1) {
280
281 // We may now leave, no more suspects
282 os.writeBytes(QUIT + "\n");
283 break;
284 }
285
286 if (responseLine.indexOf("quit") != -1) {
287
288 break;
289 }
290 }
291
292 // clean up:
293 // close the output stream
294 // close the input stream
295 // close the socket
296
297 os.close();
298 is.close();
299 tcpSocket.close();
300 } catch (UnknownHostException e) {
301 System.err.println("Trying to connect to unknown host: " + e);
302 } catch (IOException e) {
303 System.err.println("IOException:  " + e);
304 }
305 }
306
307 }
308 public String getName()
309 {
310 return "MalDes";
311
312 }
313
314 public void live() {
315
316
317 String location;
318 switch(state) {
319 case 0:
320
321 log("Processing list of destination hosts.");
322
323 location = hosts[nexthop];
324
325 if (location != null)
```

```
{
state   =   1;
log("Attempting to move to " + location);



try {
log("See you!");

move(new GrasshopperAddress(location));
}
catch (Exception e) {
log("Migration failed for some reason: ", e);
}

// Agent should have moved. If execution reaches here then this means migration failed!
// Reset state
state = 0;

}
break;

case 1:
{

log("Arrived successfully at hop number " + nexthop);

log("Move success: I"m here to gather some information and then leave
again. Working...");

mymethod();

log("Finished work. Going to next hop.");
log("Leaving!");



nexthop++;  // Increment for next hop
// See if we"ve reached all hops

if (nexthop == hosts.length)
{

// go home, we"ve reached all hops

state = 2;
try
{
move(getInfo().getHome());
}
catch (Exception e) {
log("Return trip failed: ", e);
}

}


// If execution reaches here, then we have more hops to do.
// Paste migration code here (from case 0)
```

```
385
386 // the state variable should stay at 1, so this case is repeated at destination host.
387
388 String nextlocation = hosts[nexthop];
389 try {
390 log("Over and out");
391
392 move(new GrasshopperAddress(nextlocation));
393 }
394 catch (Exception e) {
395 log("Migration failed for some reason: ", e);
396 }
397
398 } // case 1
399
400 case 2:
401 {
402 // Agent has returned. Calculate time now and subtract time_departure from it.
403 time_return= System.currentTimeMillis();
404 String strTimeReturn = (new Long(time_return)).toString();
405 log("I"m back. TimeStamp now: " + strTimeReturn);
406
407 long totaltime = time_return - time_departure;
408
409 String strTotal = (new Long(totaltime)).toString();
410 log("Work completed in " + strTotal + " milliseconds");
411
412 // Count how many remedies were sent out this mission
413 String strnuminfections = Integer.toString(infectioncount);
414
415
416 log("");
417 log("Generating report, please wait...");
418
419 boolean noInfectionsAtAll = true;
420
421 int flaglength;
422
423 for (int j = 0; j < reports.length;  j++)
424 {
425
426 flaglength = reports[j][0].length();
427 if (flaglength > 0)
428 {
429 if (reports[j][0].indexOf("true") != -1)
430 {
431 JOptionPane.showMessageDialog(null, "Machine: " + reports[j][1]
+"\nInfected: " + reports[j][0] + "\nFirst infection found: " +
reports[j][2]);
432 }
433
434 if (reports[j][0].indexOf("false") != -1)
435 {
436
437 JOptionPane.showMessageDialog(null, "Machine: " + reports[j][1] +"\nNot
Infected!");
438 }
439
440 }
441
```

```
442 }
443
444 state = 0;
445
446 }
447
448 }
449
450 }
451 }
```

8.4 Appendix 4: Code Extracts (FileMon – Remedy Application)

```
procedureTfrmMain.ParseRemedy(RemedyString: String);
Var
Temp: String;
iIndex, I, J, DelimterCount: Integer;
//InfectionName, InfectionFile, InfectionRegVal: String;

InfectionData: Array[1..4] ofString;
Begin// Sample String// REMEDY|SASSER WORM|MSPRC.EXE|SASSER_KEY

// Copy our remedy
Temp:=RemedyString;


// Count how many delimters we have
DelimterCount:= 0;

ForJ:=0 toLength(Temp) do

Begin

IfTemp[J]= '|' thenDelimterCount:= DelimterCount+1;

End;



// Re-use J variable
ForJ:= 0 toDelimterCount do

Begin

// See if this is first occurence.
// If so, delete first substring, which is not needed

IfJ=0 then

Begin

//////////////////////////////////////////////////////////////////////
////// Remove the word REMEDY which we don't need ////////////////////
//////////////////////////////////////////////////////////////////////

iIndex:= Pos('|', Temp);
// This returned index is our first, which we MUST delete because it
// represents the word REMEDY which was used by our communication protocol,
// and is of no use to this function


// Clear what we don't need from Temp to make it easier to see what's next

Temp:= Copy( Temp, iIndex + 1, Length(Temp) );

// This removes the word REMEDY and the trailing pipe character "|"

Continue;// with next iteration

End;
```

Appendix 4: continued

```
IfJ=DelimterCount then

Begin
InfectionData[J]:=Temp;

Break;

End;

////////////////////////////////////////////////////////////////////////////////
/// Extract Infection Data
////////////////////////////////////////////////////////////////////////////////

// This is repeated until we reach last iteration, where we are left// with
the last substring

// Now copy the next substring which is the actual infection name, file,// or reg
key, depending on which iteration we are at // Find the next pipe delimiter
iIndex:= Pos('|', Temp);

// Remember J starts at 1
InfectionData[J]:= Copy( Temp, 0, iIndex -1 );

// Clear what we have already extracted from Temp to make it easier// to see
what's next
Temp:= Copy( Temp, iIndex + 1, Length(Temp) );
End;
////////////////////////////////////////////////////////////////////////////////
// End of extraction
////////////////////////////////////////////////////////////////////////////////
With ListViewKill.Items.Add do
Begin
Caption:=InfectionData[2];// filename
End;

WithListViewDel.Items.Add do
Begin
Caption:=InfectionData[2];// filename // Find filename's corresponding folder

ForI:=0 toListViewFiles.Items.Count-1 do
Begin
IfLowercase(ListViewFiles.Items[I].SubItems[0]) = Lowercase(InfectionData[2]) then
Begin
// Found folder// Add to ListViewDel
SubItems.Add(ListViewFiles.Items[I].SubItems[1])
End;
End;
End;

WithListViewReg.Items.Add do
Begin
Caption:=InfectionData[2];
SubItems.Add(InfectionData[3]);// reg key
SubItems.Add(InfectionData[4]);// reg value
End;
End;
```

8.5 Appendix 5: ECIW Research Paper

Agent-based Forensic Investigations with an Integrated Framework

Buchanan WJ, Graves J, Saliou L, Al Sebea H, and Migas N
Disributed Systems and Mobile Agents Group,
School of Computing, Napier University, Edinburgh
e: w.buchanan@napier.ac.uk

Keywords: Agents, Mobile Agents, Intrusion Detection, SNMP, Integrated Framework

Abstract

Forensics investigations can be flawed for many reasons, such as that they can lack any real evidence of an incident. Also, it can be the case that the legal rights of an individual has been breached, or that the steps taken in the investigation cannot be verified. This paper outlines an integrated framework for both data gathering, using mobile and static agents, and also in the creation of a data gathering system which logs data in a verifiable and open way. Forensic information which is gathered over a network is often more verifiable over host-based data gathering. The framework for logging data for future investigations uses a formal approach where a forensics policy is defined, which is then compiled into an implementation which can run on agent systems, such as with SNMP agents, and IDS (Intrusion Detection System) agents. The paper also proposes a system which uses mobile and static agents to formalize the investigation process. This should produce investigations which can be verified, and which are programmed the expertise of an investigator, and also contain legal and moral programming to constrain the limits of a forensic investigation.

Introduction

The concept of a software agent which could investigate criminal activities is one which has intrigued modern society. It is unlikely that software will ever replace human abilities for discovering new patterns of activity, but it is possible for them to act as software tools which will intelligently filter information, and make reasoned judgements. They do, though, have many issues which would have to be overcome to make them possible. One of the main ones is to allow them to travel as programs across networks, and then intelligently gather information, and move away from their target and back to their source. This research proposes a network of intelligent crime agents which included to main types of mobile agent: a forensic investigation agent and a covert agent. These agents could migrate, investigate criminal data, and meet in a safe meeting place and exchange information, which would be passed onto the security services. The forensic investigation agent is programmed with the expertise of a forensic investigator, and securely gathers information on hosts around a network, and returns this to the forensic investigator in a form which allows a human to make judgements. In a similar way, the covert mobile agent is programmed with the expertise of surveillance activities, and its main objective is to gather information that could be used in a criminal activity, especially for large-scale criminal activities. There are, of course, many issues related to privacy, but these would be programmed into the agent in order that it did not breach the laws of the land.

At present, most programs require some sort of program running on the machine which provides a hook for a remote computer to make a connection to it. The problem for criminal investigators is thus to get a program to run on the criminal's computer, and then for it not to be detected, if it was investigating a possible crime, or for it to robustly deliver the information over a network, when involved in a forensic investigation.

A new concept has arisen, called the mobile agent, which allows this to happen, where an autonomous program has the ability to initiate its migration across a network, carrying with it its program code, its current state of execution, and also its data. These agents can freeze themselves and migrate to another host on the network. They are, in effect, the equivalent of security agents who go to an investigation and then gather the information based on the relevant data, and leave. The intelligence of the investigation is thus built into the agent and not within any programs which could run on the investigated host. It is thus simple to recall agents from hosts, as required, and to reissue them with new objectives.

There is thus the need for mobile agents who can migrate themselves onto remote hosts and gather information, and to intelligently filter it and return with useful conclusions. If the law allowed, the mobile agent could be programmed with its objectives, and migrate itself onto a remote host, in order to determine information on criminal activities. On the other hand, a forensic agent, raises fewer issues and could also be used to migrated itself onto a remote host, and sift through the information in a methodical way, and gather key information, without either sending information over a network. The agent can then return to its originator with the key elements of the investigation. After which it can return with new objectives. In order for these agents to be secure against users stealing their information, or, in the case of covert activities, detecting their presence, they must be hardened against any form of detection, thus their code must be encrypted, and they must have sensors which detect the presence of a monitor. In the case of a mobile agent, it is possible for the agent to actually kill its own presence, as required.

In order to optimize the proposed system, the data gathering must fit within an overall framework for data gathering, thus this research proposes an integrated framework for the gathering of data in an organised way.

Objectives of The Framework

This research focuses on the implementation of mobile agents which contain the expertise of forensic and covert investigators. The objectives are to:

- Provide a framework for the integrated collection of data which can be used for forensics investigations.
- Support the early detection, and possible resolution, of criminal activities.
- Provide a legal framework which controls the moral operation of the agents.
- Create models of a forensic agent and a covert investigation agent, with embedded investigation intelligence and filtering abilities.
- Implement mobile agents which contain their own execution environment, and would thus allow agents to migrate and execute on a remote computer without the need of a system hook.
- Integrate a secure communication facility to allow these mobile agents to intercommunicate in a secure way.
- Provide a robust mechanism to secure any code and data which the agents may carry.

Mobile Agents

The mobile agent paradigm is a relatively new technology that has its origins in intelligent agents, and is proposed as an alternative approach to client-server communications model. A mobile agent is a software entity that inherits some of the features of an intelligent agent and requires an agent environment to execute. A mobile agent can suspend its execution on a host computer, and then transfer its code, data state, and possibly its execution state (strong migration) to another host on the network that must provide an agent environment, and resume execution on the new host. The aim of an agent environment [9] is to provide the appropriate functionality to mobile agents to execute, communicate, migrate, and use system resources in a secure way. In general, a mobile agent comprises of an agent model, a life-cycle model, a computation model, a security model, a communication model, and finally a navigation model [1]. Mobile agent applications include information retrieval [2], e-commerce [3], network management [4], intrusion detection [5], and collaborative applications [6], and wireless computing.

Although each application can be run with the existing technologies [10], the use of mobile agents can contribute to build these distributed applications in a simpler and more effective [7]. Mobile, or wireless, computing is the most frequently proposed application area of mobile agent technology [8]. This is because mobile agents have two important features that make them particularly useful in such environments: task continuation and minimal connection.

Mobile Agent Development

Mobile agents run on specialized platforms such as Grasshopper. These allow agents to be invoked and allow for a preservation of state and context, including the migration of a mobile agent from one machine to another. Grasshopper takes care of inter-host communication that involves agent migration from between hosts.

In a typical situation, the execution of the Grasshopper platform would result in the declaration of an *agency* and a GUI would be displayed allowing the administrator to access the agency's specific tasks. Agencies must be named to differentiate between them. Agents can move around hosts by knowing the names of the different available agencies.

To allow agent identification, agents must be given distinct names, preferably human names. In addition, a unique agent identifier is automatically generated by the running agency during the creation of an agent. From a general perspective, as previously stated, a Grasshopper agent is implemented by means of a Java class or a set of classes. Each agent has one agent class which characterizes the agent and which must be derived from one of the predefined *super-classes*. Several super-classes exist, including mobile-agent and stationary-agent. Due to the nature of our project, with the involvement of mobile agents, the mobile-agent class shall be adopted. Through its super-class, each Grasshopper agent has access to the following methods [11]:

- **action**(): This method is automatically invoked by the agency if a user performs a double-click on the corresponding agent entry in the agency GUI.
- **getInfo**(): This method returns a set of information that is associated with the agent. Among others, this set of information comprises the agent's identifier, type, and name.
- **getName**(): This method returns the name of the agent. In contrast to the unique agent identifier which is automatically generated by an agency during the creation of an agent, the name can be specified by the agent programmer during the implementation phase or by the user when creating the agent, provided that this is supported by the agent implementation
- **init**(): This method is automatically called by the hosting agency when an agent is created. It offers the possibility to provide creation arguments to the agent.
- **live**(): This is the core method of each Grasshopper agent, since its implementation realises the agent's active, autonomous behaviour.
- **log**(): Allows an agent to print textual messages onto the text console of the local agency.
- **move**(): With this method, an agent is able to migrate to another agency

The basic framework of the mobile agent code which migrates to a single host (in this case to 10.0.0.1), is:

```
package examples.simple;

import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.type.*;
import de.ikv.grasshopper.communication.*;
import java.util.*;
import java.io.*;

public class ForensicAgent extends de.ikv.grasshopper.agent.MobileAgent {

  String[] files;
  int numfiles;
  GrasshopperAddress HomePlatform, suspectAddress;
  String state;
```

```
  public void init(Object[] creationArgs) {
    // Initialize data state
    files[] = new String[1000];
    numfiles = 0;
    HomePlatform = getAgentSystem().getInfo().getLocation();
    suspectAddress = new GrasshopperAddress("10.0.0.1");
    // Pass arguments to the Mobile Agent
    if (creationArgs.length < 2) {
      System.out.println("Creation arguments needed: <String> <String>");
      System.out.println("Exiting.");
      throw new RuntimeException();
    }
    else {
      setProperty(creationArgs[0].toString(),
            creationArgs[1].toString());
    }
  }

  public String getName() {
    return "ForensicAgent";
  }
  public void live() {
    if (getProperty("state").equals("INVESTIGATING")) {
      setProperty("state", "RETURN");
      move(suspectAddress);
    }
    else if (getProperty("state").equals("RETURN")) {
      try {
        Runtime rt = Runtime.getRuntime();
        Process proc = rt.exec("dir *.jpeg /s");
        InputStream stderr = proc.getErrorStream();
        InputStreamReader isr = new InputStreamReader(stderr);
        BufferedReader br = new BufferedReader(isr);
        String line = null;
        System.out.println("<ERROR>");
        while ( (line = br.readLine()) != null) {
          files [i] = line;
          numfiles++;
          if (numfiles > 998) {
            break;
          }
        }
        int exitVal = proc.waitFor();
      } catch (Throwable t) { t.printStackTrace(); }
      move(HomePlatform);
    }
    else
      remove();
  }
}
```

The migration path of the mobile agent is initially set within the init() method, and the main code to execute the forensic investigation is contained within the live() method. In the above example the mobile agent has retrieved a list of JPEG files, but in most cases the mobile agent would only communicate with a local static agent. This is achieved by opening a local TCP port on 127.0.0.1, such as to communication using port 2005:

```
tcpSocket = new Socket("127.0.0.1", 2005);
os = new DataOutputStream(tcpSocket.getOutputStream());
is = new DataInputStream(tcpSocket.getInputStream());
```

Thus, the local static agent simply creates a server socket, and the mobile agent connects to it using a local connection. The communicate is then achieved locally, and not over the network, as with a standard client-server program.

Mobile Agent framework

Figure 1 shows an outline of the usage of the mobile agent in a forensics system. Within this, the investigator has no direct interface to the host-under-investigation (HUI). This thus keeps the integrity of the system, where a static agent is immediately installed on the HUI, and guards against any changes to the information stored on the HUI. The investigator then gives the mobile agent requirement, such as the names of the JPEG images on the HUI, and the mobile agent migrates itself to the HUI, and authenticates itself to the static agent, and vice-versa. The mobile and static agents can then intercommunicate with each other and pass information about the required information. Both the mobile and the static agent are programmed with a framework which supports a legal and moral framework, thus any part of the investigation which breach the limits of the current investigation will be stopped.

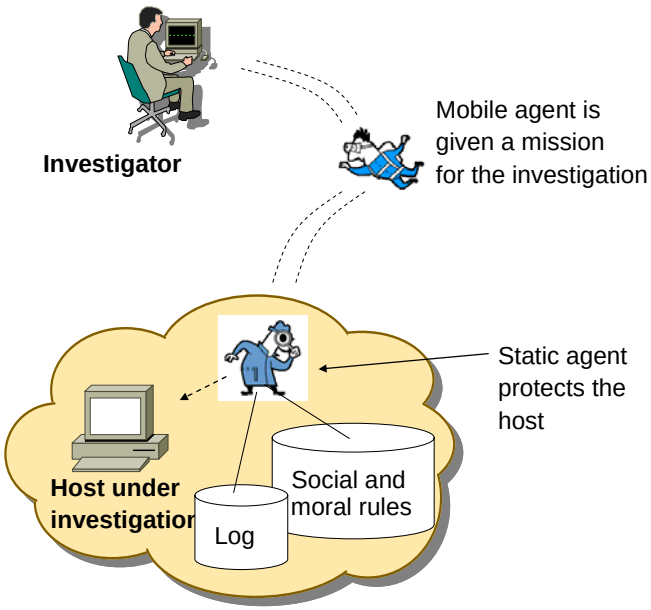


Figure 1: Forensics system using mobile and static agents

Data Gathering Framework Design

Some forensic investigations can be flawed when there is not enough care taken in the logging information that could be used in the investigation. Most involve data generation after an event has occurred, thus in modern systems the forensic logging should be carefully considered as part of the overall security policy and implementation of an organisation. Figure 2 show a standard model for forensic computing investigations, which care is taken before the incident to prepare the policies and procedures for the detection of an event. In modern day systems, organisations must be open about their forensic policies, and the data that will be logged, as a failure to do this may result in the organisation being held liable for not detecting events, or in not informing their staff on the data that is logged.

Typically forensic investigations and system security are seen as different entities, where they can be integrated into the same policy, where the security policy defines the basic services that can be operated on the system, and how threats can be dealt with. These threats are typically detected using alerts and alarms, which are based on network monitoring, and in the investigation of log files. This is where network security joins with forensic computing, as the monitoring, logs, alerts, and alarms are basic building blocks for any future forensics investigation. Security thus focuses on the active protection of systems, users and data, while forensic computing focuses on the collection,

preservation, analysis and reporting (CPAR) of events. The two come together in requiring reliable logging of data, as illustrated in Figure 3. It thus makes sense of integrate both the definition and implementation of the security policy with the definition and implementation of the forensic computing policy. Saliou [12] has developed a security framework which has a formal security policy as its input and this is used to create an implementation which maps onto the aims and objectives of the organisation, but also fits in with the legal and moral responsibilities of the organisation. This type of approach should also be applied to the generation of data for future forensic investigations, but creating a forensics policy, which integrates with the security policy, and is then used to generate the implementation of logging rules, which create formal data logs. Figure 4 shows an example of the steps taken from generating the forensic policy into implementation, and then using data agents to gather the required information into logs. The main elements:

- **Forensics Compiler**. This coverts the forensic policy, which defines the rules that define the logging of data on a network and on hosts, and coverts into a formal forensic language, which can be modelled to determine if there are any conflicts with the implementation.
- **Log implementer**. This converts the output from the forensics compiler into an implementation, such as for rules for IDSs, or to enable router logs.
- **Forensic verification traffic**. This is the network traffic which is used to verify that the system is working correctly, and includes a capture of normal traffic along with the required additional traffic which verifies that the system is working as required.
- **Live deployment**. This involves downloading the rules from the Log implementer to the devices. This includes agents such IDS agents, SNMP agents, and router/switch log agents. The agents will be responsible for filtering network traffic and logging it the required log file.
- **Data gathering**. This includes gathering the data which are stored by the agents, into a useable format.

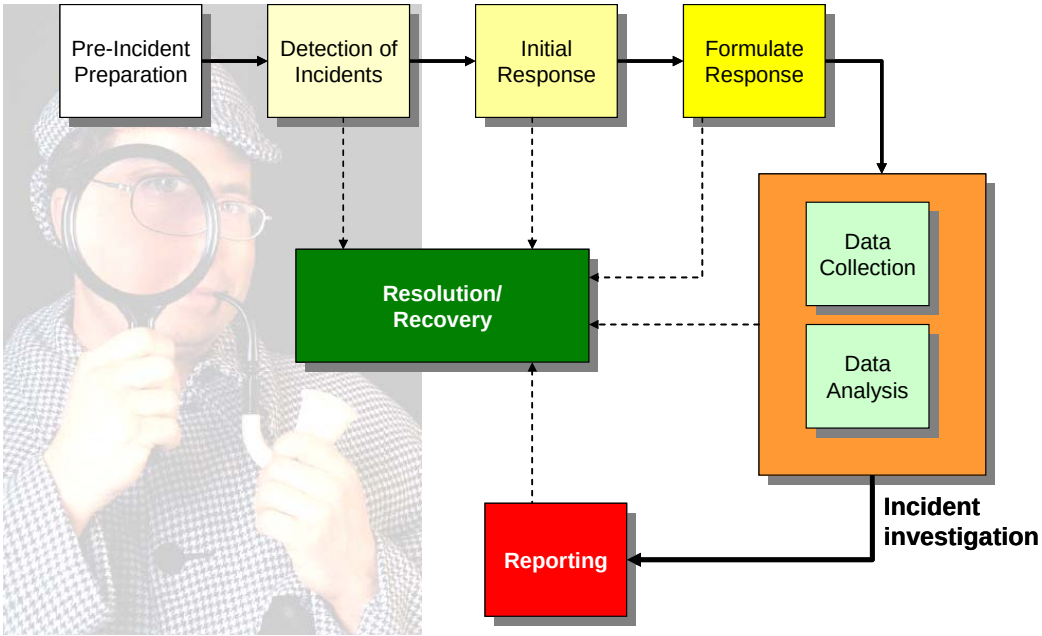


Figure 2: Forensic investigation stages

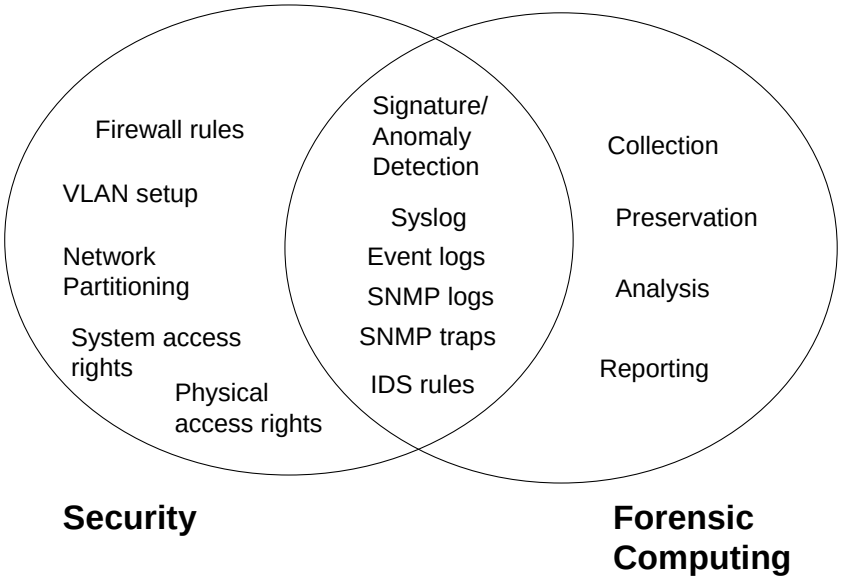


Figure 3: Intersection of forensic computing and security

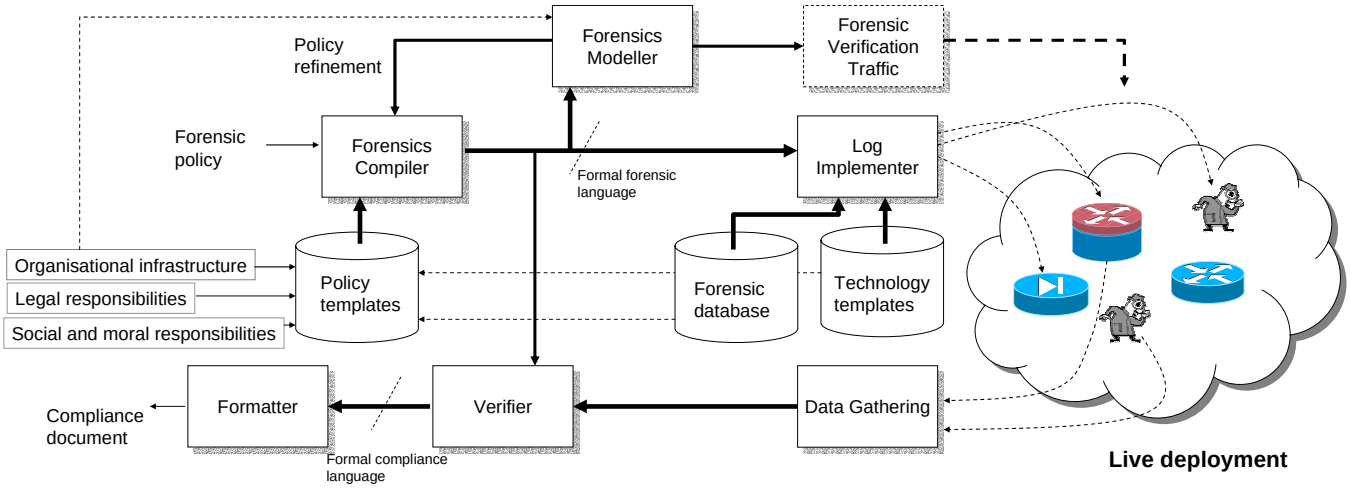


Figure 4: Formal definition and implementation of the forensic computing policy

Conclusions

Some forensic computing investigations can be flawed in that they can lack the required information. The framework proposed in this paper aids the collection of data for forensics investigation. This framework integrates with the general security definition for an organisation, as security policies also require data gathering. The cross-over between the security definition and the data logging enables the data gathering policy to be defined, so that everyone in an organisation understands the range and type of data that is logged. Along with this, alerts can be setup to provide an early detection of criminal activities before they can develop into more serious activities. These network-based logging of data for investigations are typically more reliable in its source as host-based, which can often be tampered with, or which may not leave a trace.

For host-based investigations, the combination of mobile and static agents approach proposed in the paper has advantages that it allows the verifiable logging of the investigations, and also provides the investigations to be constrained within the limits defined by moral and legal restrictions. Along with this, the mobile agent can be programmed with the intelligence of a criminal investigator, where it can be given high-level investigation command, which it then communicates with a static agent on the host under investigation. The mobile and static agents cooperate to make sure that the investigation is conducted within a legal framework.

References

[1] Harrison, C. G. et al, 1995. Mobile Agents: Are they a good idea. *Technical Report*. IBM T.J. Watson Research Centre. New York, USA.

[2] Cardi, G. et. al., 2000. Agents for information retrieval: Issues of mobility and coordination. *Journal of mobility and coordination*. Vol. 46, No. 15. pp. 1419-1433.

[3] Lee, T. O. et. al., 2001. An agent-based micropayment system for E-commerce. *E-commerce agents, Marketplace solutions, security issues, and supply and demand*. Berlin, Germany. pp.247-63.

[4] Marques, P. et. al., 2001. Providing applications with mobile agent technology. *IEEE Open Architectures and Network Programming Proceedings*. Piscataway, USA. pp.129-36.

[5] Spafford, E. H. and Zamboni, D., 2000. Intrusion detection using autonomous agents. Computer Networks. Vol. 34, No. 4. pp.547-70.

[6] Wong, D. et. al., 1997. Concordia: An Infrastructure for Collaboration Mobile Agents. *In Proceedings of the first International Workshop on Mobile Agents (MA97)*. Berlin, Germany. pp. 86-97.

[7] Puliafito, A. et. al., 2000. MAP: Design and implementation of a mobile agents' platform. *Journal of Systems Architecture*. Vol. 46, No. 2. pp.145-62.

[8] Kotz, D. et. al., 1997. Agent TCL: Targeting the needs of Mobile Computers. *IEEE Internet Computing*. Vol. 1, No. 4. pp. 58-67.

[9] Silva, A. R. et. al., 2001. Towards a reference model for surveying mobile agent systems. Autonomous Agents and Multi Agent Systems. Vol. 4, No. 3. pp.187-231.

[10] Harrison, C. G. et al, 1995. Mobile Agents: Are they a good idea. *Technical Report*. IBM T.J. Watson Research Centre. New York, USA.

[11] IKV++ GmbH (1999). Grasshopper Programmer's Guide (Release 2.2). Germany.

[12] Saliou L, Buchanan WJ, Graves J, and Munoz J, Novel Framework for Automated Security Abstraction, Modeling, Implementation, and Verification, EICW 2005.

9 References

Asaka, M., Okazawa, S., Taguchi, A. (1999). *A Method of Tracing Intruders by Use of Mobile Agents*. Japan: Waseda University [Electronic version]. Retrieved 20 February, 2005 from http://www.ipa.go.jp/STC/IDA/paper/inet99.pdf

Balasubramaniyan, J. S., Garcia-Fernandez, J. O., Isacoff, D., Spafford, E., & Zamboni, D. (1998, June). *An Architecture for Intrusion Detection using Autonomous Agents*. USA: COAST Laboratory, Purdue University. [Electronic version]. Retrieved October 18, 2004 from https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/archive/98-05.pdf

Buchanan W., Graves J., Saliou L., Al Sebea H., and Migas N. (2005). *Agent-based Forensic Investigations with an Integrated Framework*. Accepted for 4th European Conference on Information Warfare and Security (ECIW) Conference

Covaci, S. Dr. (1998). Grasshopper, The First Reference Implementation of the OMG Mobile Agent System Interoperability Facility. [Electronic document] Retrieved on May 3, 2005 from http://www.omg.org/docs/orbos/98-04-05.pdf

De Queiroz, J. D., Da Costa Carmo, L. F. R., Pirmez, L. (1999). *Micæl: An Autonomos Mobile Agent System to Protect New Generation Networked Applications*. [Electronic version]. Retrieved on November 15, 2004 from http://www.raid-symposium.org/raid99/PAPERS/Queiroz.pdf

Eid, M. (2004). *A New Mobile Agent-Based Intrusion Detection System Using Distributed Sensors*. Lebanon: Department of Electrical and Computer Engineering, American University of Beirut [Electronic version]. Retrieved November 15, 2004 from http://webfea.fea.aub.edu.lb/proceedings/2004/SRC-ECE-43.pdf

Evans, T. D. (2002). *NetBIOS, NetBEUI, NBF, SMB, CIFS Networking*. [Electronic document]. Retrieved May 1, 2005 from http://ourworld.compuserve.com/homepages/timothydevans/browse.htm

Fairhurst, G. (2001). *Ethernet Frame Calculations*. Scotland: University of Aberdeen. [Electronic document]. Retrieved May 1, 2005 from http://www.erg.abdn.ac.uk/users/gorry/course/lan-pages/enet-calc.html

Gajic, Z. (2005). *Exchanging Data Over The Network Using Delphi*. [Electronic document]. Retrieved December22, 2004 from http://delphi.about.com/od/networking/l/aa112602a.htm

Groth, D. (2001). *Network+ Study Guide*. Third Edition. San Francisco: Sybex.

Helmer, G., Wong, J. S. K., Honavar, V., Miller, L., Wang, W. (2002). *Lightweight Agents for Intrusion Detection*. The Journal of Systems and Software, 67 (2003), 109-122. [Electronic version]. Retrieved December 7, 2004 from http://www.recursionsw.com/Mobile_Agent_Papers/Light_Weight_Agent_for%20Intrusion_Detection.pdf

IKV++ GmbH (1999) [A]. *Grasshopper Basics and Concepts (Release 2.2)*. Germany

IKV++ GmbH (1999) [B]. *Grasshopper Programmer's Guide (Release 2.2)*. Germany.

Janakiraman, R., Waldvogel, M., Zhang, Q. (2003). *Indra: A Peer-to-peer Approach to Network Intrusion Detection and Prevention*. Proceedings of the annual international workshop on enabling technologies: Infrastructure for collaborative enterprises, USA, 12, p. 226. [Electronic version]. Retrieved November 15, 2004 from http://www.nisl.wustl.edu/~rama/papers/janakiraman03indra.PDF

Jansen, W. A. (2001). *Intrusion Detection with Mobile Agents*. Computer Communications 25(15), 1392-1401.

Kruegel, C., Toth, T. (2002). *Applying Mobile Agent Technology to Intrusion Detection*. Austria: Distributed Systems group, Technical University of Vienna. [Electronic version]. Retrieved November 11, 2004 from http://www.infosys.tuwien.ac.at/Staff/tt/publications/Applying_Mobile_agent_Technology_to_Intrusion_Detection.pdf

Lange, D. B., Oshima, M. (2003*). Programming and Deploying Java Mobile Agents with Aglets*. Massachusetts: Addison Wesley Longman

Mahmoud, Q. H. (1996, December). *Sockets Programming in Java: A Tutorial*. [Electronic document]. Retrieved December 17, 2004 from http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets-p2.html

Ramachandran, G., & Hart, D. (2004). *A P2P Intrusion Detection System Based on Mobile Agents*. Proceedings of The Annual ACM Southeast Regional Conference, NY, USA, 42, 185-190

Reily, D. (1998). *Mobile Agents - Process Migration and its Implications*. [Electronic Document]. Retrieved May 15, 2005 from http://www.davidreilly.com/topics/software_agents/mobile_agents

Shekhar, H. (2003.) *RPC Vs RMI*. [Electronic Document]. Retrieved May 15, 2005 from http://java.ittoolbox.com/documents/document.asp?i=2306

Shevin, E. (2000). *TSHChangeNotify Component*. [Electronic document]. Retrieved February 20, 2005 from http://delphi.about.com/library/code/ncaa030403b.htm

Sundsted, T. (1998, June). *An Introduction to Agents*. [Electronic document]. Retrieved October 5, 2004 from http://www.javaworld.com/javaworld/jw-06-1998/jw-06-howto.html

Tabor, D. Z. Jr. (1995, November). *Telnet and Rlogin*. USA: New Jersey Institute of Technology. [Electronic document]. Retrieved May 1, 2005 from http://www.cis.njit.edu/~cis456/protected/lesson24/single24.html

Tanenbaum, A. S., Van Steen, M. (2002). *Distributed Systems: Principles and Paradigms*. New Jersey: Prentice-Hall

Wier, S. K. (2002).*Designing and Managing Successful Projects*. [Electronic document] Retrieved May 1, 2005 from http://home.earthlink.net/%7Eswier/design3.html