Protecting Agents Against Malicious Host Attacks

KAMALRULNIZAM ABU BAKAR

Doctor of Philosophy

ASTON UNIVERSITY

October 2004

ASTON UNIVERSITY

Protecting Agents Against Malicious Host Attacks

KAMALRULNIZAM ABU BAKAR

Doctor of Philosophy, 2004

Thesis Summary

The introduction of agent technology raises several security issues that are beyond conventional security mechanisms capability and considerations, but research in protecting the agent from malicious host attack is evolving.

This research proposes two approaches to protecting an agent from being attacked by a malicious host. The first approach consists of an obfuscation algorithm that is able to protect the confidentiality of an agent and make it more difficult for a malicious host to spy on the agent. The algorithm uses multiple polynomial functions with multiple random inputs to convert an agent's critical data to a value that is meaningless to the malicious host. The effectiveness of the obfuscation algorithm is enhanced by addition of noise code. The second approach consists of a mechanism that is able to protect the integrity of the agent using state information, recorded during the agent execution process in a remote host environment, to detect a manipulation attack by a malicious host. Both approaches are implemented using a master-slave agent architecture that operates on a distributed migration pattern.

Two sets of experimental test were conducted. The first set of experiments measures the migration and migration+computation overheads of the itinerary and distributed migration patterns. The second set of experiments is used to measure the security overhead of the proposed approaches. The protection of the agent is assessed by analysis of its effectiveness under known attacks.

Finally, an agent-based application, known as Secure Flight Finder Agent-based System (SecureFAS) is developed, in order to prove the function of the proposed approaches.

Keywords: Agent Security, Malicious Host, Random Sequence 3-level obfuscation algorithm, Recorded State Mechanism

Acknowledgements

I wish to thank my supervisor Mr Bernard S. Doherty for his help, guidance and valuable advice throughout the research and writing of the thesis.

I also wish to thank the staff of the Department and to the folks in room MB306 for their support

Finally, I wish to thank my sponsor, the Universiti Teknologi Malaysia for providing the financial support for this research.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

Agent is a new technology that will become a major trend of distributed systems in the next decade. This technology is capable in managing the complexity of building complex applications because they provide a new way of describing a complex system or process compared to Client-Server and Peer-to-Peer Networking. However, with the advent of agent technology, the need for security of applications becomes a concern that is beyond classical data security capability and considerations. Without proper security protection, especially against attacks by malicious hosts, the widespread use of agent technology can be severely impeded.

This thesis presents the results of research carried out on the problem of protecting agents against malicious hosts attacks. It describes an investigation of the malicious host problem, the design of proposed security mechanisms for protecting agents against malicious host attacks and development of a prototype system to implement the proposed security mechanisms. The effectiveness of the protection offered by the proposed security mechanisms is evaluated by analysis of its ability to withstand known attacks. In addition, experimental results on the agent's migration pattern and the overhead of the proposed security mechanisms are also presented in this thesis.

1.1 Motivation

The motivation of this research is based on problems that arise when agents are used in an open and unsecured environment. For example, a customised agent application is sent out to visit several airline servers (in an open and unsecured environment) to find a suitable flight. In this example, the agent application is allowed to completely migrate to (the agent's originating host transfers the agent's code, data and state to the remote server) and execute in (the remote server executes the receiving agent application) the remote server environment, to take advantage of exploiting resources at the data source and thus reducing network traffic (*Wang et al.*, 2002). This opens a greater opportunity for the agent application to be abused by the executing host, because the agent application is fully under control of the executing host (*Hohl*, 2000; *Kotzanikolaou et al.*, 2000; *Vigna*, 1998).

An example of an attack by the executing host (which is known as a malicious host) is spying on the agent's data or state (*Hohl*, 1998a; *Sander and Tschudin*, 1998a). *Spying attack* by the malicious host may invade the agent's privacy, especially an agent's critical data such as user budget. The knowledge of an agent's critical data gives a malicious host an advantage in any competition over other hosts because the malicious host knows what is expected by the agent. For example, a customised agent is sent out (in an open and unsecured environment) to find a suitable flight with the fare price less than or equal to 500 pounds. An attack from a malicious host based on the spying attack is implemented by raising the offered price until it meets the maximum price that has been set by the agent's owner, even though the normal price is much lower. This is possible because the malicious host knows that the price that it offers still fulfils the requirement set by the agent's owner. A Spying attack by the malicious host on an agent's data or state is one of the most difficult attacks to detect, because the attack does not leave any trace that could be detected (*Hohl*, 1998a; *Sander and Tschudin*, 1998a; *Hohl*, 2000). In addition, the executing host has to read the agent's code, must have access to agent's data, and must be able to manipulate the agent's variable data in order to execute the agent (*Hohl*, 1998a, 2000; *Mandry et al.*, 2001).

Therefore, the executing host can see and access all of the agent's code including data and state, which makes any attempt to address spying attack difficult.

Another kind of malicious host attack is to tamper with the agent's code, data or state, so that the agent will forget all the previous visits and offers held by the agent, and thus force the agent (application) to accept an offer from the malicious host even though the malicious host's offer is not the best offer (*Hohl*, 2000; *Sander and Tschudin*, 1998a; *Vigna*, 1998). That kind of attack is known as a *manipulation attack* (*Hohl*, 1998a, 2000). In this attack, the owner of the agent may not know the attack has happened. This is because the malicious host may make subtle changes in the agent's code, data and state, which are difficult to detect, thus enabling the malicious host to achieve its objective. In addition, the agent (application) that returns from the malicious host does not show any different behaviour from an untampered agent, which makes the attack difficult to detect and prevent.

The problem of a malicious host attacking an agent executing inside the malicious host cannot be addressed fully by traditional security services such as confidentiality, integrity and authenticity (*Hohl*, 1998a, 2000). In addition, traditional security services were not devised to address attacks on the application by the executing host (*Hohl*, 2000; *Jansen*, 2000).

Although encrypting the entire agent's code, data and state could prevent attack by the malicious host, the inability of the malicious host to read, write or execute an encrypted agent application, prevents the host executing the agent, thus preventing the agent from executing its tasks (*Mandry et al.*, 2001).

This thesis proposes two security mechanisms for protecting agents against malicious hosts attacks. The first mechanism aims to protect the confidentiality of an agent against a malicious host's spying attack. This mechanism uses multiple polynomial functions for obfuscating the actual value of the agent's critical data to an obfuscated value that is meaningless to the malicious host, in order to prevent the malicious host from spying on the agent critical data. The obfuscation method used in the first mechanism enables the execution host to execute the comparing process in an obfuscated format without the execution host having any knowledge of the actual value of the agent's critical data. This comparing process can be done on obfuscated data without needing deobfuscation of the data, unlike cryptographic methods, which require decryption of the data, thus revealing its value, before a comparison can be made. The second mechanism is proposed to protect the integrity of the agent's data and state against the malicious host's manipulation attack. This mechanism uses the agent state information, which consists of the agent's data (located in its variables) and execution information. The state is recorded during the agent execution session inside the executing host environment, and used by the master agent to detect manipulation attacks from the malicious host.

In order to enhance the level of security protection provided by the proposed security mechanisms, a distributed migration pattern and master-slave agent architecture are combined with the proposed security mechanisms. This combination will overcome *extraction of information attack* [1] and *collaboration attack* [2] by consecutive executing hosts (the malicious hosts). This is because, instead of using one agent to migrate through a sequence of n remote hosts, which allows the extraction of information and collaboration attack to occur, the distributed migration pattern allows n agents (slave agent) to migrate to n remote hosts. Here, one specific agent serves one specific remote host without any relationship with other agents. This prevents extraction of information attacks because each of the agents has knowledge only of a specific remote host visit. Furthermore, the isolated relationship between the agents makes collaboration attack by the malicious hosts impossible. This thesis offers the hypothesis that the security of agents can be improved by the proposed security mechanisms.

This thesis also investigates two performance hypotheses:

- the performance of an agent-based application that uses master-slave agent architecture with a distributed migration pattern is faster than an agent-based application using single agent architecture with an itinerary migration pattern,

[1] an attack from the malicious host by extracting information from agent's previous visit to win any competition against other remote hosts

[2] an attack from the malicious host by collaborating with other malicious hosts to remove any attacks' traces recorded by the agent from the previous agent visit

even though the first agent-based application is required to generate and dispatch more than one agent to execute a transaction, which does increase the network overhead at the originating host, and

- the performance of an agent-based application equipped with security mechanisms is slower than the agent-based application without security mechanism.

In order to investigate the hypotheses, this thesis analysed agent security under known attacks and developed an evaluation method, proposing two sets of experimental measurements:

- an experimental test on the migration and migration+computation overhead of the itinerary and distributed migration patterns, and
- an experimental test on the implementation overhead of the proposed security mechanisms.

1.2 Aims of Research

The aims of this thesis are:

1. to propose security mechanisms for protecting agents against malicious host attacks,
2. to assess the effectiveness of the security mechanisms in protecting the agent,
3. to develop and carry out tests for evaluating the security mechanisms in terms of their overheads, and
4. to develop a prototype system to implement the security mechanisms.

1.3 Research Methodology

1.3.1 Literature Review

The literature review covers two main areas:

- **Agent Technology.** The area of agent technology is investigated due to the use of agent technology as a foundation technology for this research. Agent technology has been found in the literature as an alternative technology for the traditional client-server and message-based architecture in designing distributed systems application. Agent technology also provides powerful and effective mechanisms to develop applications in distributed systems. The investigation in this area provides:

  - an introduction to agent technology, which includes the description of its components, migration process and execution environment,
  - a brief discussion of agent evolution, which emerges from the Remote Procedure Call (RPC) and Remote Evaluation (REV) technology (*Minar*, 1998; *Stamos and Gifford*, 1990; *Wong et al.*, 1999),
  - a general discussion of agent technology that presents a definition of an agent, description of agent attributes and their types, the advantages and disadvantages of using agents and a few examples of agent applications,
  - an introduction to the Aglets Software Development Kit (ASDK), which is used as an agent system for this research work.

- **Agent Security.** In the area of agent security, emphasis is given to the problem of the malicious host attacking the agent executing on the remote host. The lack of security protection against a malicious host has been a crucial aspect that has severely impeded widespread use of agent technology. In addition, existing security mechanisms were not devised to address attacks on an application by the executing host. This investigation has identified four areas of agent security that need to be addressed, which are:

  - the security between agents,
  - the security between hosts,
  - the security between a host and unauthorised third parties, and

- the security between an agent and a host.

1.3.2 Analysis and Investigation

Analysis and investigation are carried out in the agents security area, which is mainly focused on the problem of protecting agents against malicious hosts attacks. Two main kinds of the malicious host attacks on the agents, spying attack and manipulation attack (These attacks will be discussed further in Chapter 3, Section 3.4.2), were studied (*Hohl*, 1997, 1998a; *Guan et al.*, 1999). Conclusions found were used to develop the proposed security mechanisms.

1.3.3 Prototyping

A prototype was developed to implement the proposed security mechanisms in order to examine the feasibility of the proposed security mechanisms. The prototype was developed using the Java language. This is due to the following reasons:

- the Aglets Software Development Kit (ASDK) used for developing an agent-based application in this research uses the Java language as its programming language,
- suitable cryptographic functions and APIs for the prototype system security implementation are available in Java,
- to make the prototype system able to operate in heterogeneous environments and support platform independence, and
- to benefit from the Java language that works well with the agent technology.

1.3.4 Tests Used

In this research, two sets of experimental tests were used to investigate the research hypotheses:

1. An experimental test on the itinerary and distributed migration patterns in order to evaluate the migration and migration+computation overheads,

2. The experimental test to evaluate the overhead of the proposed security mechanisms.

The first experimental test was conducted by timing the interval taken starting from sending the agents to the remote hosts and ending by receiving the agents from the remote hosts. For evaluating migration overhead, no processing activities were required at the remote host, the agents were simply returned. However, for evaluating migration+computation overhead, the agents have to execute a numerical calculation algorithm inside the remote host environment.

The second experimental test was conducted by measuring the time taken starting from sending the agents to the remote hosts, where the proposed security algorithm or mechanism was executed inside the remote host environment, and ending by receiving the agents back from the remote hosts.

1.4 Novel features of the thesis

The novel aspects of this thesis are:

- the Random Sequence 3-level obfuscation algorithm, which was designed from the idea of *Hohl* (1998a), is used to overcome the problem of a malicious host spying attack on an agent's critical data. The Random Sequence 3-level obfuscation algorithm is able to prevent the malicious host from spying on the initial conversion process of the actual value of agent's critical data to the obfuscated value and, combining the algorithm with noise codes (*Ng and Cheung*, 1999a,b), makes it more difficult for the malicious host to guess the agent's critical data. The implementation overhead of this algorithm is also measured.

- the Recorded State Mechanism, which is built on the work of Hohl (*Hohl*, 2000), Vigna (*Vigna*, 1998) and Farmer (*Farmer et al.*, 1996a), to address the problem of a malicious host's manipulation attack on an agent's data and state. We extended that work by using a master slave agent architecture with a distributed migration pattern that is able to prevent collaboration attacks by more than one consecutive remote host and also prevent the extraction of information by the malicious host. Also three Recorded State Mechanism's containers (i.e. RecordedReadOnly, RecordedExecuteOnly and RecordedCollectOnly) were used for detecting a manipulation attack from the malicious host. In addition, the evaluation of the agent's migration pattern overheads are conducted to measure:

- the migration overhead (which includes communication and serialization overhead) that occurs while dispatching the agent to or receiving the agent from the remote host, and
- the total execution time (including processing and migration time) of the agent plus the migration overhead.

Furthermore, an evaluation of the Recorded State Mechanism is also conducted to measure the time overhead for executing the mechanism.

1.5 Outline of thesis

The thesis is organised into seven chapters. This first chapter gives an overall introduction to the research and areas related to it. This includes discussion on the research problem, aims, methodology and novelty of the thesis.

Chapter 2 presents an overview of the agent technology, which is a foundation technology for this research.

Chapter 3 mainly describes on the problem of protecting the agents against the malicious hosts attacks.

Chapter 4 describes the design of the proposed security mechanisms for protecting the agents against the malicious hosts attacks.

Chapter 5 describes the prototype developed to implement the proposed security mechanisms in order to examine the feasibility of the proposed security mechanism.

Chapter 6 summarises the experimental results, which followed by the interpretation of the results.

Chapter 7 presents the evaluation of the research as a whole, proposals for further research and a conclusion.

Appendix A describes a step-by-step installation and configuration procedures of the Aglets Software Development Kit (ASDK).

Appendix B presents the SecureFAS user manual.

Appendix C presents the design of the SecureFAS using Use Case and Sequence Diagram.

Appendix D presents program listings.

Part II

Literature Review and Research Formulation

Chapter 2

Agent Technology

2.1 Introduction

Agent technology is an emerging technology that is gaining momentum in the field of distributed systems (*Schoder and Eymann*, 2000; *Wong et al.*, 1999; *Silva et al.*, 2000). It provides a new paradigm in designing and implementing applications that operate in a distributed environment (*Perraju*, 1999). It supports asynchronous and autonomous execution, and robustness in complex distributed system application. In addition, agent technology has gained much attention from industry and the academic community, and many commercial implementations of agents have been presented in the market, such as ARCHON (*Cockburn and Jennings*, 1996), Kasbah (*Chavez and Maes*, 1996) and BargainFinder (*Krulwich*, 1996).

2.2 Evolution of Agents

The rapid growth of network technology and the demand for resource sharing have motivated computing to evolve from centralised systems into distributed systems (*Coulouris et al.*, 2001; *Bacon*, 1997). In distributed systems, four major enabling technologies are in place. In historical order, they are the Message Passing, Remote Procedure Call (RPC), Remote Evaluation (REV) and Agent technology (*Minar*, 1998; *Stamos and Gifford*, 1990; *Wong et al.*, 1999), with the later one building on top of the previous one.

Message Passing involves with an architecture in which client and servers communicate using communication lines. This technology uses communication media such as sockets, ports and server sockets for two or more computers to communication with each other's (*Ince*, 2002).

As an alternative to Message Passing technology, Remote Procedure Call (RPC) and Distributed Objects are a client-server infrastructure that allows programs (applications) to be distributed over multiple heterogeneous platforms. It reduces the complexity of developing applications by insulating the application developer from the details of the various operating systems and network interface function calls (*Coulouris et al.*, 2001; *Peterson and Davie*, 2003). In RPC, a program (client program) that is executing in a client host, communicates with another program (server program) that is executing in a server host by calling functions provided by the other program (server program)(*Minar*, 1998; *Coulouris et al.*, 2001; *Peterson and Davie*, 2003; *Bacon*, 1997) (see figure 2.1). The client program that issues the communication request will then wait for a response to be returned from the server program before the client program can continue to execute other processes.

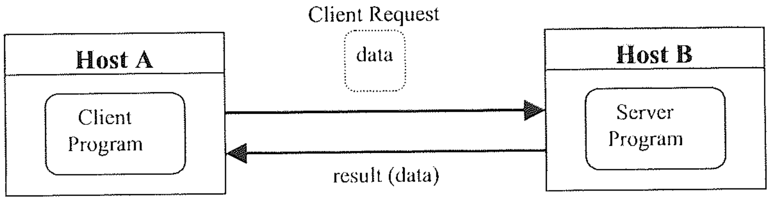


Figure 2.1: Remote Procedure Calls (RPC)

On top of all that, Remote Evaluation (REV) was proposed by *Stamos and Gifford* (1990). In REV (see figure 2.2), the client, instead of using a remote procedure calls to communicate with a server, sends its own program to a server and requests the server to execute the program. Upon receiving the client request, the server executes the program supplied to it by the client. Once the execution is completed, a result is sent to the client. In this technology, the client that sends the Remote Evaluation request can wait until the result is received from the server, or continue executing some other processes whilst the server is processing the client program (*Stamos and Gifford*, 1990; *Hughes*, 2001).

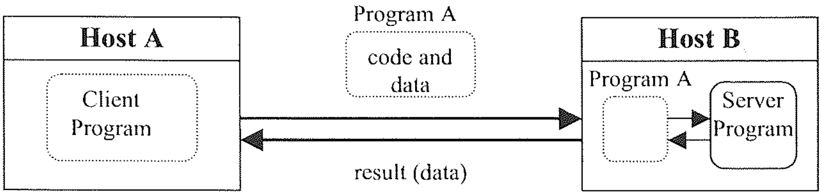


Figure 2.2: Remote Evaluation (REV)

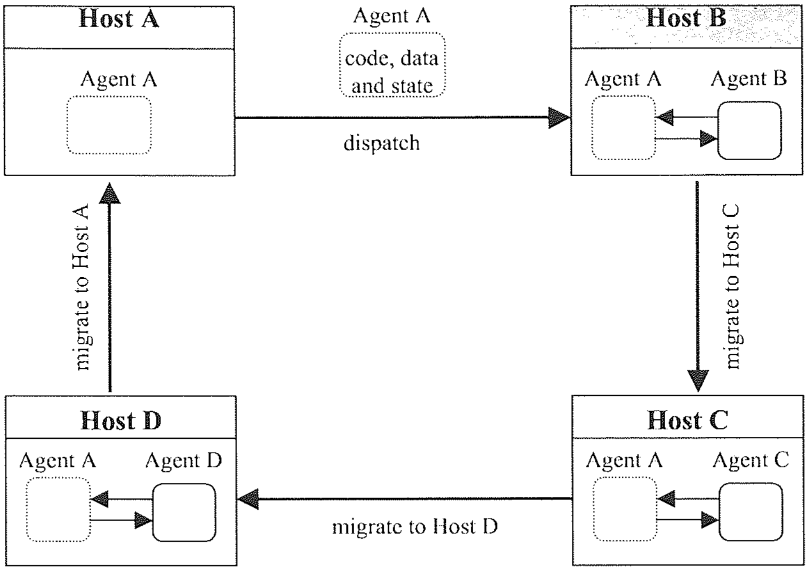
Agent technology (see figure 2.3) on the other hand, is an extension of the Remote Evaluation (REV). An agent is an executable program that consists of code, data and state sent by a client to a server (*Lange and Oshima*, 1998, 1999; *Hohl*, 1997; *Biehl et al.*, 1998). Unlike a Remote Evaluation (REV) which requires a client to send a program to a server to get a result (if any) (*Stamos and Gifford*, 1990), an agent could migrate to wherever there are computing resources, execute its tasks on a remote server and return to its origin as needed. An agent is able to control its own migration[1], thus it has more autonomy than a Remote Evaluation (REV).

2.3 Agent Definition

Various definitions for agent have been produced, but so far, the research community has accepted none of them as a standard definition (*Nwana*, 1996; *Sundsted*, 1998). This is because many definitions are given from different perspectives based on the specific work of the researchers. For example, Nwana (*Nwana*, 1996) defines agents as,

> *"a component of software and/or hardware which is capable of acting independently in order to accomplish tasks on behalf of its user".*

[1] the agent ability to move from one machine to another



Note: Agent's data and state can change at each Host.

Figure 2.3: Agent Technology

While Maes (*Maes*, 1995) defines agents as,

> *"a computational system that inhabits a complex, dynamic environment. The agent can sense, and act on, its environment, and has a set of goals or motivations that it tries to achieve through these actions",*

and Hayzelden and Bigham (*Hayzelden and Bigham*, 1999) say an agent is

> *"an independently executing program able to handle autonomously (i.e., without direct input at run time from a human) the selections of actions when expected or limited unexpected events occur".*

Coulouris (*Coulouris et al.*, 2001) on the other hand defines an agent as,

> *"a running program (including both code and data) that travels from one computer to another in a network carrying out a task on someone's behalf, such as collecting information, eventually returning with the results ".*

The definition of an agent given by *Coulouris et al.* (2001) will be used in this research.

2.4 Agent and Agent System

An agent has three main components (*Hohl*, 1997; *Biehl et al.*, 1998):

- the code, which consists of the instructions that define the behaviour of the agent,
- the data, which is the value of the instance variables in agent code, and
- the current state of execution of the agent (including agent program counter and frame stack).

An agent has the ability to travel or migrate from one execution host to another execution host that contains services with which the agent wants to interact. This ability enables the agent to take advantage of being in the same execution host as the services (*Lange and Oshima*, 1998, 1999; *Wang et al.*, 2002) (see figure 2.3), which can reduce network communication, thus reducing network traffic (*Wang et al.*, 2002; *Tripathi et al.*, 2002).

In order to execute inside the execution host environment, an agent needs an execution environment provided by the execution host. This is known as an agent system. The agent system can be regarded as the operating system for agents (see figure 2.4) (*Lange and Oshima*, 1998; *Chess et al.*, Mar., 1995). This agent system has the capability to create, execute, transfer and terminate the agent processes. It is responsible for providing services for the agent to do its work and can support more than one executing agent at the same time. The agent system consists of four elements (*Lange and Oshima*, 1998):

- An engine which serves as virtual machine for executing the agent, and provides links to the underlying network and other resources provided by the execution host,
- Resources such as databases, processors, and other services provided by the execution host,

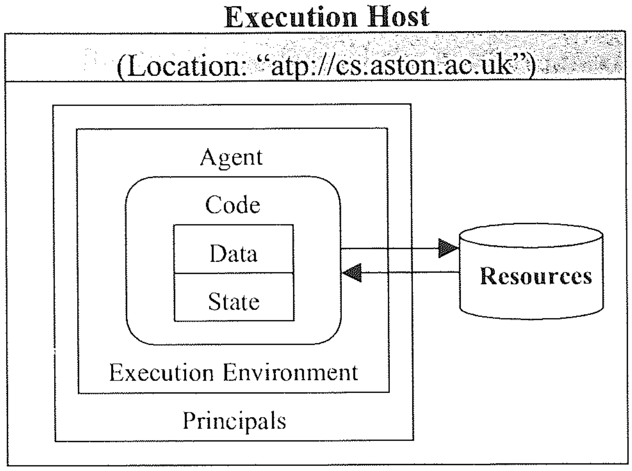


Figure 2.4: Agent and Agent System

- A location which is an address of an executing agent, produced from the combination of the name of the execution environment and the network address of the engine in which the execution environment resides, and
- Principals which control the operation of the execution environment.

There are many agent systems but some examples of agent system are Aglets Software Development Kit (ASDK) from IBM (*Lange and Oshima*, 1999; *Tai*, 1999; *Silva et al.*, 2000), Odyssey and D'Agents from General Magic (*Noble and Satyanarayanan*, 1999; *Ousterhout*, 1994; *Silva et al.*, 2000), and Voyager from ObjectSpace (*ObjectSpace*, 1997; *Silva et al.*, 2000).

2.5 Agent Terminology

2.5.1 Agent Attributes

Some of the commonly identified agent attributes that are useful for the thesis are described below:

- **Mobility.** An agent can travel or migrate to other environments, interact with remote hosts, gather information on behalf of its owner and come back to the originating host after having performed the duties (*Jansen*, 2000).

- **Autonomy.** An agent can operate without any intervention or guidance from humans or others (*Nwana*, 1996; *Chauhan*, 1997). This means, agent should have a degree of autonomy from its owner[2] to do its jobs. Otherwise, it is just doing the tasks as instructed by the owner of the agent, and it will be locked-step and fixed, i.e. waiting to be instructed by its owner (*Foner*, 1993).
- **Social Ability/Co-operation.** An agent can interact with other agents and/or humans (*Chauhan*, 1997). Owner-agent cooperation can be described as a form of collaboration in constructing a contract. The owner specifies what actions should be performed on their behalf, and the agent specifies what it can do and provides results (*Foner*, 1993). In inter-agent cooperation, both agents can co-operate with each other to solve large problems that are beyond their individual capabilities. The agents can exchange their knowledge and plans in order to work together (*Chauhan*, 1997).
- **Temporal Continuity.** An agent is a continuously running process, not a "one-shot" computation that terminates itself when the processes had been completed (*Chauhan*, 1997).

2.5.2 Types of Agents

The combination of the agent's attributes creates different types of agent and each of them serves specific type of tasks. Different types of agent are described as below:

- **Mobile Agent.** An agent that is capable of moving from one computer to another, interacting with remote hosts, gathering information and returning to its originating host after having performed a task (*Chauhan*, 1997).
- **Autonomous Agent.** An agent that can sense and act autonomously within the environments where it is situated (*Franklin and Graesser*, 1997).

[2] a creator of an agent

- **Information Agent.** An agent that has access to potentially many information sources and is able to collate and manipulate information obtained from these sources to answer queries posed by users and/or agents (*Chauhan*, 1997). This agent is sometimes referred to as an Internet Agent as such an agent roams the Internet in order to collect information.

- **Intelligent Agent.** An agent that can carry out some sets of operations on behalf of a user or another program with some degree of independence (*Chauhan*, 1997). This agent is encoded with artificial intelligence and has the ability to perform a learning operation during its execution in order to find the best way to complete a task.

2.6 Advantages of Agents

There are many advantages claimed for agents (*Lange and Oshima*, 1999; *Chess et al.*, Mar., 1995). Some of the more frequently quoted and accepted claims are:

- **Reduce the network load.** In distributed systems, a communication protocol involves multiple interactions in accomplishing a given task. As a result, it causes a lot of network traffic (see figure 2.5 a). However, using an agent, user's conversation will be packaged and dispatched to a destination host where interactions take place at the destination host. This reduces network traffic. On the other hand, agents can also reduce the flow of raw data in the network by processing the raw data where it is found rather than transferring it over the network. This can be done by moving the computation to the data rather than the data to the computation (see figure 2.5 b).

- **Overcome network latency.** Network latency[3] is a very critical problem to real-time systems especially in distributed environments. One of the solutions to this problem is to use agents. The agents can reduce network latency problems

[3] the delay on a network that occurs while a data is being stored and forwarded

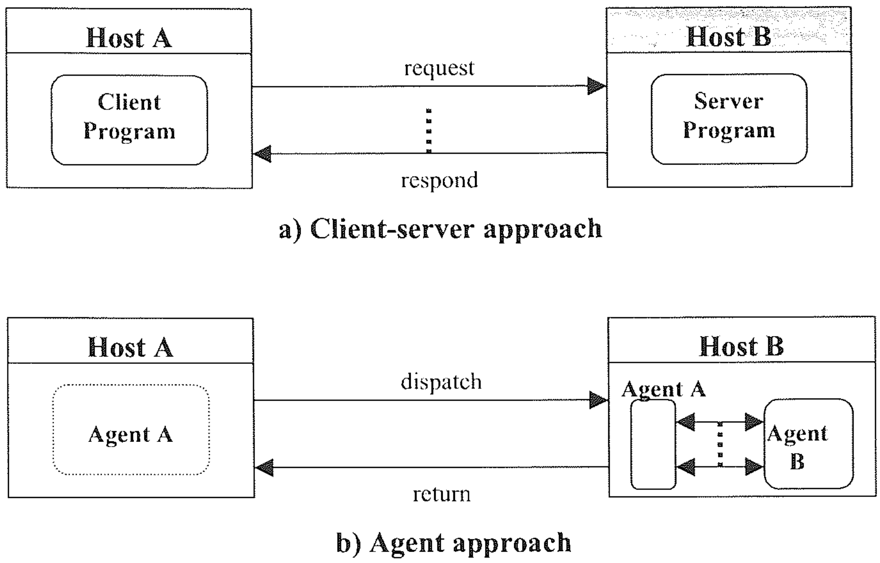


Figure 2.5: Agents Reduce the Network Load

because it can be dispatched from a central system to act locally and execute the system's directions directly.

- **Execute asynchronously and autonomously.** An agent has the capability to operate asynchronously and autonomously in order to execute the given tasks. After being dispatched, the agent's owner can disconnect his network connection. After that, the agent becomes independent and can operate asynchronously and autonomously. The agent's owner can reconnect at a later time to collect the agent (see figure 2.6).

- **Adapt dynamically.** An agent can sense its execution environment and react autonomously to changes.

These advantages make agents a suitable and beneficial technology for various application domains. The author elaborates on this in section 2.8.

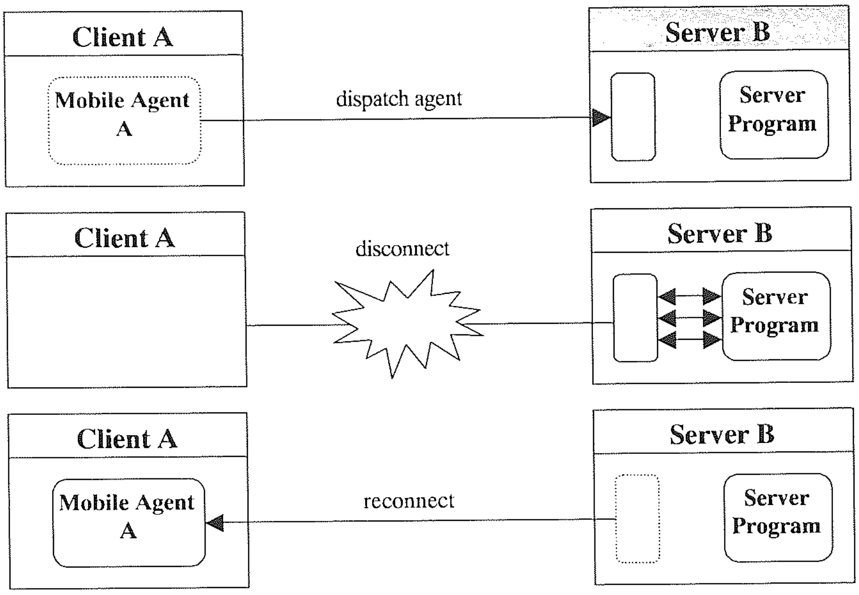


Figure 2.6: Disconnected Operation

2.7 Disadvantages of Agents

Although many advantages of agents have been described, there are a few disadvantages of agents that need to be considered, such as (*Chess et al.*, Mar., 1995; *Schoder and Eymann*, 2000):

- a need for highly secure agent execution environment to protect the remote host,
- performance limitations resulting from security to protect the agents,

2.8 Application Domains for Agents

There are wide ranges of application domains that make use of agents. Agent applications are being developed for fields as varied as electronic commerce, information gathering and parallel processing. *Lange and Oshima* (1998, 1999) have compiled a list of agent application domains:

- **E-Commerce.** This application consists of a commercial transaction that requires real-time access to remote resources such as customer, shopper and banking databases. Agent technology is well suited for this kind of application because it has the ability to support real time application. In addition, agents can act and negotiate on behalf of their owners in order to accomplish transactions in electronic commerce.

- **Information Gathering.** Agent owners can dispatch their agent to remote information sources to perform searching and create search indexes locally. Agents can also perform extended searches that are not constrained by the hours during which an owner's computer is operational.

- **Information Dissemination.** Information in the computer network can easily be disseminated using agent technology. Agents can disseminate news, software updates from vendor and installation procedures, direct to the customers' computers. Agent can also autonomously update and manage that information without user interference.

- **Monitoring and Notification.** In this application, an agent is used to monitor remote hosts without being dependent on the systems from which the agent originated. The agent will notify the originator hosts if certain kinds of information become available.

- **Parallel Processing.** Agents have an advantage in this application. Agents can create a cascade of clones in the network and administer parallel processing tasks. Agents can also distribute tasks among multiple processors if a computation requires more processor power.

2.9 The Enabling Technologies

Two enabling technologies used to support the implementation of the proposed security mechanisms are:

- the master-slave agent architecture,

- the distributed migration pattern.

2.9.1 The master-slave agent architecture

This architecture was first introduced by *Buschmann et al.* (1996) and later extended by *Lange and Oshima* (1999) to fit into agent technology. Generally there is not much difference between these two architecture (*Buschmann et al.*, 1996; *Lange and Oshima*, 1999), nevertheless their usage is completely different. Whereas Buschmann's intent was to support fault tolerance, parallel computation and computational accuracy, Lange and Oshima used it for one more purpose, which is to support tasks at remote destinations.

The master-slave agent architecture has been defined as a scheme whereby the master agent can delegate tasks among slave agents (*Lange and Oshima*, 1999) (see Figure 2.7). This scheme allows the master agent to continue its tasks after despatching the slave agent to other destination hosts to perform the assigned task. For instance, the master agent creates a slave agent for each subtask and dispatches it to a remote host. While the slave agent computes the partial result to the task it has been assigned, the master agent can continue its work. When the slave agents have all finished their work, the master agent compiles the final result and returns it to the user.

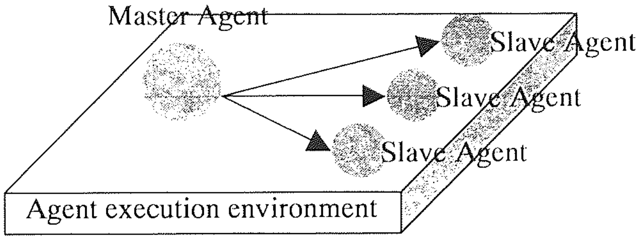


Figure 2.7: Master-slave agent architecture

The main objective behind this architecture is to achieve a better performance in terms of the processing speed by delegating tasks to other agents. However, the drawback of this architecture is that the behaviour of a slave agent is fixed at design time. In addition, simple problems may not benefit from partitioning and delegating tasks, because it increases the overall computation effort of the global task.

2.9.2 The distributed migration pattern

The distributed migration pattern is the agent migration pattern that allows agent migration to n remote servers, but instead of using one agent to migrate through a sequence of n remote servers, the distributed migration pattern allows n agents to migrate to n remote servers in parallel, with one agent for one remote server (see Figure 2.8).

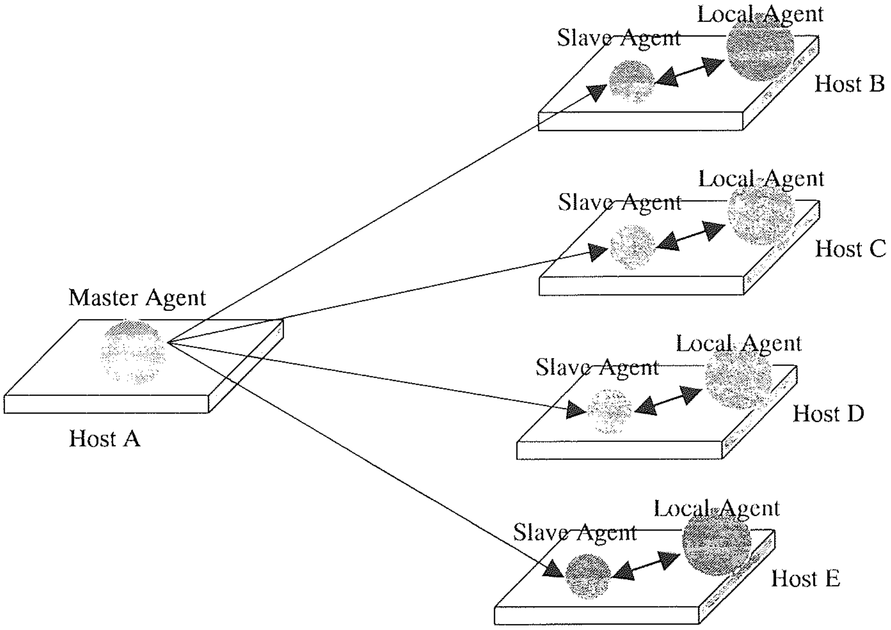


Figure 2.8: Distributed Migration Method

The distributed migration pattern can be used together with the master-slave agent architecture to support operation of the proposed security mechanisms.

2.10 Concluding remarks

This chapter presents an overview of agents, which is a foundation technology for the research work. It started by giving agent definitions, then presented the evolution of the agent, which emerges from Remote Procedure Call (RPC) and Remote Evaluation (REV) technology. This was followed by a brief discussion on agents to give a description of its components and the execution environment.

The attributes and the types of agent form the terminology of agents in the next section. However, only those related to the thesis were described.

The advantages and disadvantages of agents were presented in this chapter for a basic guideline on the benefits and problems in using the technology. A few examples on agent application domains were presented to show the impact of the technology in real world applications. Finally, two agent's enabling technologies were introduced to support the implementation of the proposed security mechanisms that will be discussed in the later chapter.

This chapter has been focussing on agent technology in general and the next chapter will look specifically into the literature of the security in agent technology.

Chapter 3

Security in Agent Technology

3.1 Introduction

In the literature, many application areas such as electronic commerce, mobile computing, network management and information retrieval can benefit from agent technology because the exploitation of agent technology offer several advantages such as reduction of network load, overcoming network latency and allowing asynchronous execution (*Lange and Oshima*, 1998, 1999; *Corradi et al.*, 1999b; *Hohl*, 1997). In addition, agent technology is often described as a promising technology for developing applications in an open, distributed and heterogeneous environment, such as the Internet (*Coulouris et al.*, 2001; *Corradi et al.*, 1999b; *Hohl*, 1997). However the lack of security protection has severely impeded the widespread use of the agent technology (*Coulouris et al.*, 2001; *Jansen*, 2000; *Oppliger*, 1999; *Corradi et al.*, 1999b).

3.2 Attack Definition

The term "attack" in this thesis is defined as the act of extracting, spying on or changing the executing agent's code, data or state by the execution host, resulting in an unintended change in behaviour or unauthorised access to data (*Hohl*, 2000).

From this definition, an attempt to extract or spy on agent's code, data or state requires a security mechanism where sensitive information need to be protected, while an attempt to change the agent's code, data or state requires a security mechanism that can at least detect the difference between the attacked agent and non-attacked agent.

This definition of attack will be used as the foundation for the whole discussion in this thesis.

3.3 Agent Security Areas

An agent can migrate among hosts, seeking to fulfill tasks on behalf of its owner and eventually returning to its originating host with a result (*Coulouris et al.*, 2001; *Luckham*, 2002; *Rus et al.*, 1997).

In a closed network environment, for example one contained entirely within a single organisation where all hosts are controlled by one administration and the same administration also employs the agents, it may be possible to trust both the agents and the hosts (*Tripathi and Karnik*, 1998). On the other hand, in an open and unsecured network environment where different hosts are controlled by different administrations and a different administration employs the agents (*Farmer et al.*, 1996b; *Hohl*, 1998b; *Jansen*, 2000), neither the agents nor the hosts are necessarily trustworthy.

The agent, if it is a malicious agent might try to access or destroy privileged information, which the agent is not authorized to access, or may consume more resources than it can should. The execution host, if it is a malicious host, might try to extract confidential information from the visiting agent, tamper with the agent's code, data and state or spy on agent's code, data or state (*Hohl*, 1998a, 2000).

As agent technology is expected to become a possible base platform for an electronic services framework, especially in the area of Electronic Commerce (*Hohl*, 1998a; *Wang et al.*, 2002; *Corradi et al.*, 1999a), reliable security protection is a crucial aspect, since some transactions in this area might involve confidential information, such as credit card number, bank account information or some form of digital cash, that has value and might therefore be attacked. In addition, without proper and reliable security protection, the widespread use of agent technology in real world applications could be impeded.

In the literature, there are four main security areas in agent technology that need to be addressed (*Jansen*, 2000; *Hohl*, 1997; *Gray et al.*, 1998):

- security between agents, which is concerned with the problem of an agent attacking another agent that is resident in the same execution host,
- security between hosts, which is concerned with the problem of a host attacking another host,
- security between host and unauthorised third parties. This is concerned with the problem of other entities (agent or host) from outside or inside the execution host attacking the execution host, and
- security between agent and host, which is concerned with the problem of an agent attacking an execution host, and an execution host attacking an agent.

3.3.1 Security Between Agents

An agent has the capability to communicate, exchange information and services, and collaborate with other agents in the same execution host in executing its tasks. This capability could expose the agent to the problem of malicious agent attacks (*Jansen*, 2000; *Hohl*, 1997, 2000). For example, consider a situation where an agent is used to execute payment transactions on behalf of its owner in a host owned by a bank. In this situation, the malicious agent, which is also executing inside the bank host execution environment, can pretend (masquerade) to be an authorised agent that works on behalf of the bank host in responding to any transactions from other executing agents (*Hohl*, 1998a,b). The malicious agent could convince the attacked agent to provide it with, for example, a payment in some form of digital cash, a credit card number, bank account information, or other private information during the attacks. This attack could harm the agent that is being deceived and the agent whose identity has been assumed, especially in agent societies where reputation is valued and used as a means to establish trust.

The malicious agent can also attack other agents by repudiating transactions or communications from other agents (*Vigna*, 1998). This could lead to serious disputes that may not be easily resolved without proper countermeasures in place. For example, in the situation where the malicious agent buys goods from a trader agent and then refuses to pay after receiving the goods, because the malicious agent denies receiving the goods. Without sufficiently strong evidence the trader agent cannot prove that the malicious agent was involved in the transaction and has received the goods.

A denial of service attack is another kind of malicious agent attack that can be launched against another agent (*Hohl*, 1997). The malicious agent can attack other agents by repeatedly sending messages to the attacked agent. The attack will cause the attacked agent to find a high burden on its message handling routines, which could affect the attacked agent's performance and eventually could prevent it executing its tasks. In addition, if the attacked agent is to be charged for payment by the number of CPU cycles it consumes on the agent's host, the attacked agent could be charged with a high amount of money when the agent ends its execution process, even though the agent's own task requires only a small number of CPU cycles to processes it. This could be due to an extra processing overhead consumed by the attacked agent to handle a denial of service attacks from the malicious agent.

However, attacks such as masquerade, repudiation and denial-of-service from a malicious agent do not raise any new security issues that do not arise in current computer systems (*Hohl*, 1997, 2000). Many existing security mearures such as digital signature, public key infrastructure and other cryptographic algorithms, such as DES and IDEA could be use to overcome these attacks (*Stallings*, 1999; *Pfleeger*, 1997; *Schneier*, 1996).

3.3.2 Security Between Hosts

As in an ordinary computer host, an execution host has the capability to communicate with other execution hosts to exchange information or set up transactions. This capability could lead to an internetwork security problem (*Stallings*, 1999; *Pfleeger*, 1997; *Ford and Baum*, 2001). To illustrate the problem, consider the following cases:

- Host A transmits a file to Host B. The transmitted file contains sensitive information, such as payroll records that need to be protected from disclosure. Host C (the malicious host), which is not authorised to read the transmitted file, is able to monitor the transmission and capture a copy of the file during its transmission.

- The Certificate Authority (CA) server transmits a message to Host A. The message instructs Host A to update its authorisation file to include the identities of a number of new trusted hosts, which are to be given access to that host. Host C (the malicious host) intercepts the message, alters its contents to add or delete entries, and then forwards the message to Host A, which accepts the message as coming from the CA server and updates its authorisation file accordingly.

- Rather than intercept a message, Host C (the malicious host) constructs its own message with the desired entries and transmits the message to Host A as if it had come from the CA server. Host A accepts the message as coming from the CA server and updates its authorisation file accordingly.

- A message is sent from Host B to a Stock Broker Host with instruction for various transactions. Subsequently, the investments lose value and the Host B denies sending the message.

- Host C (the malicious host) *floods* Host B with dummy messages to prevent Host B providing services or obtaining resources needed to perform its own tasks.

- Host C (the malicious host) pretends to be trusted host to access other host confidential data or damage the system.

None of the above attacks are new to current computer systems (*Hohl*, 1997, 2000). Common security protection such as authentication and authorisation can be applied to overcome the problems without any modifications specific to agent technology (*Hohl*, 1997, 2000; *Stallings*, 1999).

3.3.3 Security Between Host and Unauthorised Third Parties

A host can interact with other entities both from outside and inside the host environment, which may include agents and other hosts. In an open and unsecured network environment, these entities may attempt to disrupt, harm, or subvert the host, which interacts with it (*Hohl*, 1998a; *Mandry et al.*, 2001). For example, the entity could masquerade as a trusted agent, and request access to services or resources for which it is not authorised (*Hohl*, 1998a; *Jansen*, 2000). The entity could also intercept agents or messages in transit to manipulate their contents, or simply replay the transmission dialogue at a later time in an attempt to disrupt the synchronisation or integrity of the agent, or the entity could intercept a shopping agent and replay it several times to make the shopping agent buy more than the original shopping agent had intended (*Jansen*, 2000).

This area does not introduce new security issues that differ from the security issues in current computer system. Existing security mechanisms can be applied to address the problems that exist in this area (*Hohl*, 1997; *Mandry et al.*, 2001).

3.3.4 Security Between Agent and Execution Host

In this area, there are two security issues that need to be addressed (*Hohl*, 1997; *Jansen*, 2000):

- security of a host against a malicious agent, and
- security of an agent against a malicious host.

Security of a Host against a Malicious Agent

An execution host can easily be exposed to various security threats, such as a malicious agent attacks, because the host has to provide an execution environment to support the execution of incoming agents that originate sometimes from generally unknown and untrusted host (*Corradi et al.*, 1999b; *Kun et al.*, 2000). For example, consider a situation where a malicious agent masquerades as an authorised agent. In this example, the malicious agent can easily gain access to services or resources of the execution host, and use them in an unexpected and disruptive fashion. The malicious agent can also shift the blame for any actions for which it does not want to be held responsible to the agent it is pretending to be. This could damage the trust the legitimate agent has established in an agent community and it associated reputation.

In the literature, there are many security protection methods have been developed to protect the execution host against the malicious agent attacks, such as Sandboxing, Digital "shrink-wraps", and Proof-Carrying Code (*West and Gloudon*, 2003; *Oppliger*, 1999; *Necula and Lee*, 1998).

- **Sandboxing.** A "Sandbox" is a technique that protects the execution environment of the host from malicious agent attacks (*West and Gloudon*, 2003). The technique imposes security restrictions on privileges and access rights of the agents that execute inside the execution host environment to prevent the agent from damaging the execution host. Java is one of the well-known programming languages that use this technique.

- **Digital "shrink-wraps".** Digital "shrink-wrap" is a technique that has been pioneered by Microsoft in its Authenticode technology (*Oppliger*, 1999). This technique is able to protect the execution host from the malicious agent attacks by authenticating the arriving agent before the agent is executed. This done by having the execution host inspect the arriving agent to identify the producer of signed code and verify that the code has not been tampered with.

- **Proof-Carrying Code.** Proof-Carrying Code (PCC) is a technique that has been proposed to enable the execution host to determine whether the arriving code (agent), which is provided by other hosts is safe to install and execute (*Necula and Lee*, 1998). This technique uses an encoding of a proof provided by the code producer, which is encoded in a form that can be transmitted digitally to the consumer (the execution host) and then quickly validated using a specific proof-checking process.

The solutions described above can be used to address the problem of a malicious agent attacking a host without requiring any changes specific to agent technology. Therefore, the security problem in this area is considered solved.

Security of an Agent against a Malicious Host

The problem of the execution host attacking the application (the agent) executing inside the execution host environment is difficult to solve (*Hohl*, 1998a; *Oppliger*, 1999). That is because existing security mechanisms were not devised to address attacks on the application by the execution host (*Hohl*, 1998a; *Oppliger*, 1999; *Jansen*, 2000). Normally in the current computer system, the party that maintains the execution environment generally also employs the application. However, in agent technology, the agent and the execution host are operated in most cases by different parties. This situation leads to the problem of the malicious host (*Mandry et al.*, 2001; *Chan et al.*, 2000; *Schelderup and Olnes*, 1999).

A malicious host can be defined as a party that is able to execute an agent that belongs to another party and tries to attack that agent in some way (*Hohl*, 1998a; *Oppliger*, 1999). The malicious host can exist anywhere along the agent route, pretending to be a trusted host to deceive agents as a true and trusted destination to launch attacks. For example, the malicious host pretends to be a trusted shopper host, attracting shopping agents to its execution environment and attacks the executing agents by extracting sensitive information, such as a credit card number, bank account information, some form of digital cash, or other private information that is valuable to the attacker. The malicious host could also deny the execution of an agent or terminate the agent execution without any notification.

During the execution process of an agent in a remote host execution environment, the entire agent code, data and state is exposed to the execution host. This offers a greater opportunity for the malicious host to attack the agent that executing inside the malicious host execution environment (*Mandry et al.*, 2001; *Schelderup and Olnes*, 1999). For example, the malicious host could spy on an agent's secret keys or electronic cash, where the simple knowledge of the data results in loss of privacy or money (*Hohl*, 1998a). The malicious host could also manipulate the executing agent code with the effect that the agent prefers the offer of a certain airline provider, regardless of the price, or modify the shop list after setting the offer of the local airline provider as the best offer (*Hohl*, 1998a).

Since the aspect of securing an application (an agent) against a malicious host is a new security issue in computer science, and the existing security mechanisms are not applicable to address the issue, there are very few propose solutions to the problem of protecting agents against a malicious host is estimated in the literature (*Oppliger*, 1999; *Corradi et al.*, 1999b; *Hohl*, 1998a). For that reason, this thesis will investigate how to prevent the problem of a malicious host attacking an agent that is executing inside the malicious host environment.

3.4 The Malicious Host Problem

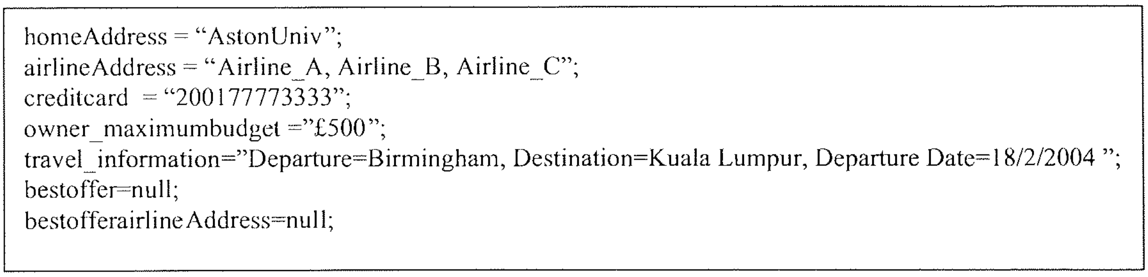
The term attack has been defined in section 3.2 as extracting, spying on or changing agent's code, data or state.

In the literature, the problem of a malicious host attacking agents that are executing under the malicious host execution environment has been classified as a difficult problem to be solved (*Corradi et al.*, 1999a; *Vigna*, 1998; *Chess et al.*, Mar., 1995). This is due to the fact that different agents have been managed and executed by various parties, the execution host has full access to all parts of the agents (i.e. code, data and state), and also has total control on the agents execution (*Diaz et al.*, 2000; *Hohl*, 1998b; *Vigna*, 1998). In addition, some researchers have declared that the problem of malicious host is not solvable (*Farmer et al.*, 1996a; *Chess*, 1998) and currently only a few approaches exist that try to solve the problem entirely. These include Mobile Cryptography (*Sander and Tschudin*, 1998a) and Time Limited Blackbox Protection (*Hohl*, 1998a), which are considered not mature enough to be used in real world applications (*Hohl*, 2000).

Security protection of an agent can be divided into two categories: security protection for an agent that is in transit and security protection for an agent that is at its destination host (*Vigna*, 1998; *Reisner and Donkor*, 2000; *Kun et al.*, 2000; *Chess*, 1998). Security protection for the agent in transit is relatively simple. The sending host could encrypt and digitally sign the entire agent, including its code, data and state, such that only a true destination host will be able to read and execute the agent. In addition, a few attacks such as traffic analysis, stolen keys and attacks on the key-distribution infrastructure that could attack the agent in transit are mostly well understood and could be addressed by current security mechanisms (*Chess*, 1998; *Schneier*, 1996; *Ford and Baum*, 2001). However, once the agent arrives at its destination host, the agent is exposed to a severe problem of malicious host attack that has rarely occurred in current computer systems (*Hohl*, 1997, 2000; *Vigna*, 1998).

3.4.1 The analysis of the Malicious host problem

To analyse the problem of malicious host, an example of a simple purchasing agent is presented. In this example, the purchasing agent is required to buy an airline ticket from virtual airline companies on the Internet on behalf of its owner. The purchasing agent is equipped with a list of virtual airline companies, a credit card number, the owner's maximum budget and the owner's travel information, as shown in figure 3.1.



```
homeAddress = “AstonUniv”;
airlineAddress = “Airline_A, Airline_B, Airline_C”;
creditcard  = “200177773333”;
owner_maximumbudget =”£500”;
travel_information=”Departure=Birmingham, Destination=Kuala Lumpur, Departure Date=18/2/2004 ”;
bestoffer=null;
bestofferairlineAddress=null;
```

Figure 3.1: A Purchasing Agent (data block) - adapted from *Hohl* (1997)

The journey of the purchasing agent starts when the owner of the purchasing agent dispatches it on the Internet (see figure 3.2). The purchasing agent then migrates to every virtual airline company in the owner list to ask for the price of the requested ticket. If the price is lower than the maximum budget and lower than the lowest price so far, the purchasing agent stores the new lowest price and the address of the virtual airline company in its data variable (see figure 3.3). After visiting all the virtual airline companies in the owner list, the purchasing agent then migrates back to the virtual airline company that offers the lowest price and buys the ticket using its owner credit card number, which the purchasing agent carries.

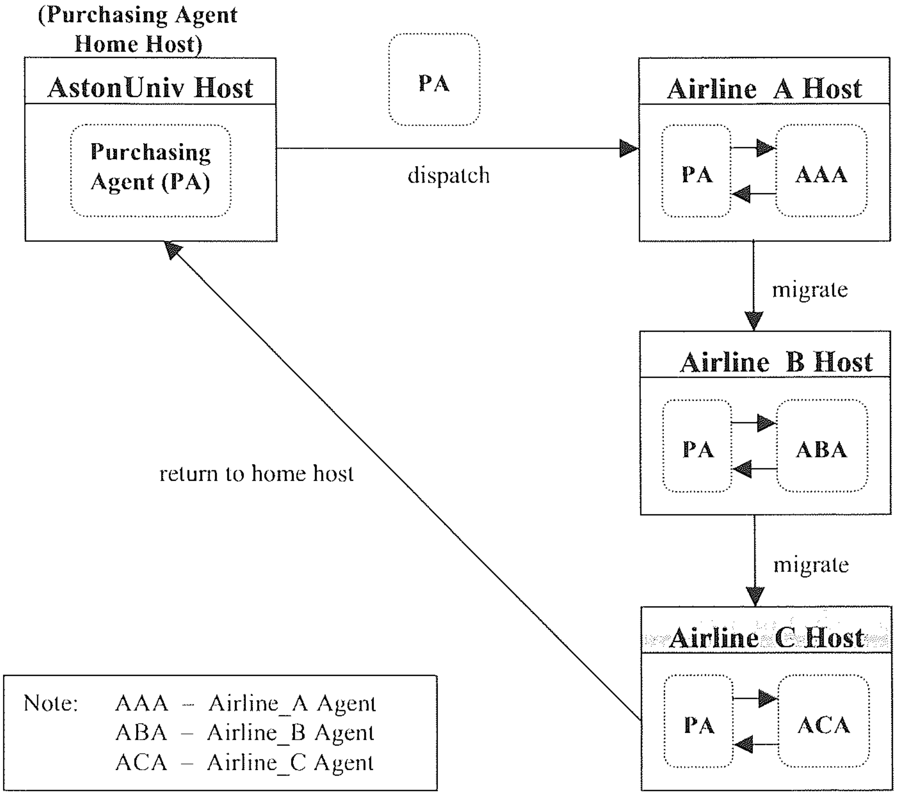
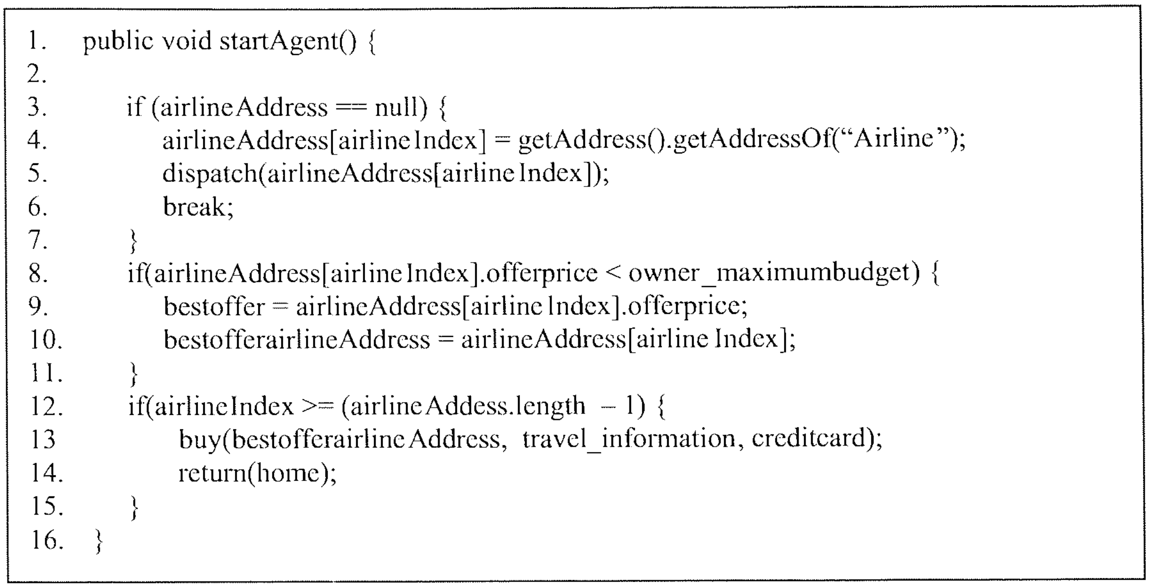


Figure 3.2: A Purchasing Agent Route

In this example, the purchasing agent carries items that are sensitive and confidential such as the credit card number, the owner's maximum budget, the best offer and best offer airline address. Those data items are vulnerable to attack.

To illustrate a malicious host attack, assume that Airline_C host, which is one of the virtual airline companies in the example (figure 3.2), is a malicious host. During the purchasing agent's journey, the purchasing agent will visit all virtual airline companies listed in the owner list including the Airline_C host, without knowing that the Airline_C host is a malicious host. As mentioned in section 3.4, the execution host has full access



```
public void startAgent() {

    if (airlineAddress == null) {
        airlineAddress[airlineIndex] = getAddress().getAddressOf("Airline");
        dispatch(airlineAddress[airlineIndex]);
        break;
    }
    if(airlineAddress[airlineIndex].offerprice < owner_maximumbudget) {
        bestoffer = airlineAddress[airlineIndex].offerprice;
        bestofferairlineAddress = airlineAddress[airlineIndex];
    }
    if(airlineIndex >= (airlineAddess.length - 1) {
         buy(bestofferairlineAddress, travel_information, creditcard);
         return(home);
    }
 }
```

Figure 3.3: A Purchasing Agent (code block) - adapted from *Hohl* (1997)

to all parts of the agent (i.e. code, data and state), and also has total control of the agents execution. In this example, when the purchasing agent migrates to the Airline_C host, the purchasing agent's code, data and state will be accessible to the Airline_C host. This gives the Airline_C host the opportunity to manipulate the existing best offer, the best offer airline company and copy the purchasing agent's credit card number. Without security protection, the purchasing agent owner can detect none of these attacks.

3.4.2 Malicious Host Attacks

This section discusses the different types of attack that could be theoretically launched on an agent executing inside a malicious host execution environment.

Spying attack

A spying attack is used to gather information about the agent, especially the agent's sensitive information such as a credit card number, bank information and user purchase requirement, which could be used by a malicious host in future attacks (*Hohl*, 1997; *Guan et al.*, 1999; *Hohl*, 1998a).

Although a spying attack does not physically harm the agent or tamper with its code, data or state, the attack could cause the problem of a leak of privacy from the agent (*Hohl*, 1998a).

Spying attack by a malicious host is difficult to detect or prevent. This is due to the fact that this attack does not leave any trace on the attacked agent (*Hohl*, 1998a, 1997).

Manipulation attack

Due to the fact that the execution host (malicious host) must be able to read and access the entire agent in order to execute the agent, the malicious host can easily attack the executing agent by manipulating its code, data and state (*Hohl*, 1997, 1998a). In this attack, the owner of the agent may not know that the attack has happened. That is because the malicious host may make subtle changes in the agent's code, data and state, which are difficult to detect but enable the malicious host to achieve its objective. In addition, the agent that returns from a malicious host does not show any different behaviour to distinguish it from an untampered agent, thus this attack is difficult to detect and prevent.

To illustrate a manipulation attack, consider an example where a purchasing agent is sent out to visit several airline servers in an open and unsecured environment to find a suitable flight fare before booking the flight. One kind of attack by the malicious host is to manipulate the executing agent with the effect that the agent prefers the offer of a certain airline server, regardless of the price, or to modify the airline server list after setting the offer of the local airline server as the best offer, forcing the purchasing agent to purchase the required flight ticket from the local airline server and forcing the purchasing agent to return to its home host without visiting all of the airline servers on its original list.

This kind of manipulation attack is difficult to prevent. However if we can detect it, we can avoid being tricked by the malicious host (*Hohl*, 1998a, 1997).

Incorrect execution of code

Normally, once the agent arrives at its destination host, the host will be responsible for providing the agent with an execution environment in which to execute the agent. However, in a malicious environment, the malicious host can execute the agent in many different way without changing the code or the control flow of the agent (*Hohl*, 1997, 1998a). For example, the malicious host could delay the execution of the agent, the malicious host could jump within the executable code to execute the agent selectively, or the malicious host could terminate the execution of the agent prematurely. However if we able detect it, the possibility to be tricked by the malicious host can be avoided (*Hohl*, 1998a, 1997).

Masquerading

The malicious host pretends to be a trusted host to deceive the agent to execute inside its execution environment (*Hohl*, 1997, 1998a). In this attack, once the agent starts to execute inside the malicious host execution environment, the malicious host can launch other attacks, such as spying on or manipulation.

Denial Of Execution

Since the execution host has full control of the agent that is resident inside its execution environment, a malicious host could deny the execution of the agent (*Hohl*, 1997, 1998a). This is known as a denial of execution attack. For example, in the situation where the malicious host knows about a time limited offer of another host, the malicious host could delay the execution of the agent that is resident inside its execution environment until the offer expires. In addition, the malicious host could totally refuse to execute the agent to prevent the agent continuing its work.

3.5 Existing approaches for the malicious hosts problem

This section presents existing approaches that try to solve the problem of malicious hosts. The approaches can be divided into three categories (*Hohl*, 1998b):

- protect agents using trusted hosts,
- protect agents against single attacks, and
- protect agents against all attacks.

3.5.1 Protecting agents using trusted hosts

This category comprises approaches that either employ a host infrastructure that is operated by a single party (Organisational Solution), or approaches that allow agents to migrate only to trusted hosts (Trusted Hosts) (*Farmer et al.*, 1996b; *Hohl*, 1998b; *Sander and Tschudin*, 1998a).

Organisational Solution

The organisational solution is an approach that uses one trustworthy party to maintain the execution hosts and also to execute the agents (*Hohl*, 2000, 1998a). Normally, this approach is implemented within the same organisation or company where only one trustworthy party is required to maintain the execution hosts. Unfortunately, the use of this approach will restrict the agent's autonomy and require a critical mass of infrastructure in order for it to be used (*Hohl*, 2000).

- **Trusted Hosts**

  Trusted hosts is an approach that uses a separate trustworthy host to evaluate the data that has been collected by the agent (*Farmer et al.*, 1996b; *Marques et al.*, 1998). In this approach, both code and data of the agent are encrypted and digitally signed and only the trusted host can read it. This guarantees the confidentiality and integrity of the code and data. However, according to *Mandry et al.* (2001), the problem of this approach is to find trustworthy hosts. Furthermore, all of the collected data needs to be sent over the network, but only a part may be actually used.

- **Protect agents against single attacks**

  This category contains approaches that try to prevent a single malicious host attack (*Vigna*, 1998; *Yee*, 1997; *Meadows*, 1997), and these approaches are set out in the following subsection.

Reference State

Reference State is a mechanism that is able to detect most manipulation attacks from the malicious host (*Hohl*, 2000). The mechanism consists of the variable parts (i.e. the state) of an agent executed by a host, showing reference behaviour (i.e. the information about agent activities during its execution session). It uses the next host in the agent's path to measure the difference in the variable parts of an agent computed from the untrusted host on one hand and a reference host on the other hand. The input to the reference states mechanism includes all of the data from outside the agent, such as both communication with partners residing on other hosts and data received directly by or via the current host. This includes the results from system calls such as random numbers or the current system time (*Hohl*, 2000, 1999).

The Reference State mechanism was developed based on the "Cryptographic Traces" approach (*Vigna*, 1998), but used different ways to check the resulting states. In the cryptographic traces approach, the suspicions of the owner are used to start the checking. However, in reference states mechanism, the checking is done in every case. In addition, the reference states mechanism uses the next host in the agent's path to check the resulting state regardless of whether the next host is a trusted one or an untrusted one. The decision has the disadvantage that a collaboration attack involving two or more consecutive hosts cannot be detected (*Hohl*, 2000). The mechanism is able to detect the attack if the resulting state is different from a reference state, arising from

20	homeHost=HomeAddress	20	homeHost=www.HomeSweetHome.co.uk
21	targetHost=targetAddress	21	targetHost=www.trader.co.uk
22	Time=systemTime	22	Time=14:15 PM
23	read(Price)	23	Price=100
24.1	if (userBudget > price)	24.1	
24.2	Status = Accept	24.2	
24.3	else	24.3	
24.4	Status = notAccept	24.4	
24.5	endif	24.5	Status = Accept
	(a)		(b)

Figure 3.4: (a): Code Fragment and (b): Trace of the Code

manipulation, write and incorrect execution attack. However, not every manipulation, write and incorrect execution attack can be detected, only those which indeed result in an incorrect state of the agent can be detected (*Hohl*, 2000, 1999). In addition, attacks such as a read attack cannot be detected by the mechanism because these attacks do not result in a different agent state.

Cryptographic Traces

Cryptographic traces is an approach for tracing the execution of a migrating agent (*Vigna*, 1998). It allows the owner of the agent to check its agent's execution traces at each hosts in the route followed by the agent when an attempt to tamper with the agent is suspected. In addition, the owner of the agent can prove, in case of tampering that the agent could never have performed the claimed operations. This approach requires each of the hosts visited by the agent to record the execution trace of the visiting agent when it executes the agent (see figure 3.4b). The execution trace includes a sequence of statements executed by the host and any related information obtained.

When the execution process of the agent has completed, the host creates a hash of the trace and a hash of the resulting agent state. These hashes are signed by the host and sent to the next host, together with the code and state of the agent. The trace itself has to be stored by the host. The agent then continues to fulfil its task and returns to its home host afterwards.

Now, the agent owner can decide whether he wants to check the agent or not. In case of suspicion, the owner of the agent can request the trace from the corresponding hosts starting with the first host. After receiving the trace, the owner of the agent can compare it with the one stored at the next host. If these traces are identical, the host commits to this trace and the agent with its initial state is re-executed. In the case that the statements used are input from the outside, the values recorded in the trace are used. If a hash of the resulting state of the agent on the host is equal to the one signed by this host (which can be provided also by the next host), this means that the host did not cheat, and the checking process continues (*Hohl*, 1999). However, the large size of the agent execution trace, even if the trace is compressed is a disadvantage of this approach (*Vigna*, 1998).

Partial Result Authentication Codes

Partial Result Authentication Codes (PRAC) is a mechanism that provides forward integrity of an intermediate agent state or a partial result that resulted from agent execution process on a host (*Yee*, 1997). In this mechanism, instead of authenticating the origins of a message, the mechanism authenticates the intermediate agent state or partial results by using a cryptographic checksum.

The PRAC mechanism requires an agent and its owner to maintain, or incrementally generate, a list of secret keys used in the PRAC computation. Once a key is applied to encapsulate the information collected, the agent destroys it before moving onto the next host, guaranteeing forward integrity. The forward integrity property ensures that if one of the hosts visited is malicious, the previous set of partial results remains valid. However, only the owner can verify the results since no other copies of the secret key remains. As an alternative, public key cryptography and digital signatures can be used in lieu of secret key techniques. One of the benefits of this mechanism is the authentication of the results can be made a publicly verifiable process at any platform along the way, while maintaining forward integrity.

However, the PRAC mechanism has a number of limitations (*Jansen*, 2000). The most serious occurs when a malicious host retains copies of the original keys or key generating functions of an agent. If the agent revisits the host or visits another host conspiring with it, a previous partial result entry or series of entries could be modified without the possibility of detection. Since the PRAC is oriented toward integrity and not confidentiality, any host visited can view the accumulated set of partial results, but this can be easily resolved by applying sliding key or other forms of encryption.

Double Integrity Verification

Double Integrity Verification is an approach that uses the combination of a one-way collision-resistant hash function and a digital signature (*Wang et al.*, 2002). As stated by *Wang et al.* (2002), for each code module fabricated by the agent factory, a digest using the hash function is computed, denoted as IMD (Intermediate Message Digest). Another digest of the overall agent code, including the original agent body, together with all the currently added code modules contributed to the value of OMD (Overall Message Digest). Each time the agent roams to a trusted host (TTP) in the SAFER[1] community, both the IMD and OMD digests are updated and digitally signed by the trusted host (see figure 3.5). For clarity, SAFER community is an infrastructure for intelligent agent-mediated electronic commerce. It comprises of various components and entities, such as the agent butler, agent factory and community administration center (*Zhu et al.*, 2000).

The task of integrity verification involves double verification of the digital signature and hash value of both IMD and OMD. The verification of digital signatures indicates whether or not the hash value is valid. IMD verification indicates whether the individual code part has been compromised, while OMD indicates the integrity of agent code as a whole.

However, this scheme currently has been applied only within the SAFER community, which is a closed network community and all of the hosts inside the SAFER community are a certified trusted host. Therefore, the probability of the existence of

[1] Secure Agent Fabrication, Evolution and Roaming

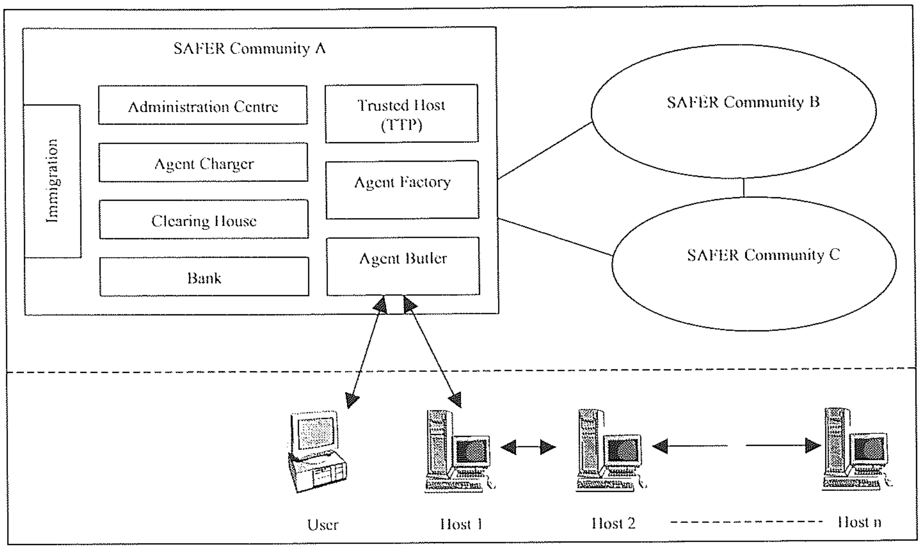


Figure 3.5: A SAFER Agent Community - adapted from *Wang et al.* (2002)

a malicious host is very low. In addition, if this approach is implemented in an open and unsecured network environment, collaboration attack by consecutive hosts cannot be detected.

Environmental Key Generation

Environmental key generation is an approach that allows an agent to take predefined action when some environmental condition is occurs (*Riordan and Schneier*, 1998). For example, an agent has a cipher-text message (a data set and a series of instructions) and a method for searching through the environment for the data needed to generate the decryption key. When the proper environment information is located (environmental condition is triggered), the key is generated, the cipher-text is decrypted, and the resulting plain text is acted upon. Unfortunately, if the agent is unable to find the environment input to decrypt the cipher-text, the agent itself cannot decrypt its own cipher-text.

In this approach, the environmental condition is hidden through either a one-way hash or public key encryption of the environmental trigger. This mean that a host of the agent cannot uncover the triggering message or response action by directly reading the agent's code (*Jansen*, 2000). However, the host could simply modify the agent code to force the agent to print out the unlocked executable code upon receipt of the trigger.

3.5.2 Protect agents against all attacks

This category contains approaches that try to protect an agent from any attack by a malicious host (*Sander and Tschudin*, 1998a,b).

Time Limited Blackbox

Time limited blackbox is an approach that is able to protect an agent from most malicious host attacks (*Hohl*, 1998a). This approach is based on an obfuscation algorithm such as variable re-composition, conversion of control flow elements into value-dependent jumps and deposited keys, where the strength of these algorithms does not necessarily imply encryption mechanisms, but relies mainly on obfuscation algorithms. The main strategy behind this approach is to scramble the agent's code in such a way that no one is easily able to gain a complete understanding of the agent's code function.

After being input into the conversion mechanism (see figure 3.6), the agent appears to be a blackbox to the intruders, where it becomes difficult to decode and analyse. In addition, when a time factor is applied to this blackbox agent, the computations carried by the agent are only valid within a period of time. If the intruders cannot understand the blackbox within the time interval, the attack is claimed void. Since an agent code may be invalid after certain period, this approach is suitable only for applications that do not convey information intended for long-lived concealment.

Unfortunately, no known algorithm to fully provide blackbox protection exists so far (*Hohl*, 1998a; *Jansen*, 2000). In addition, a serious drawback for this technique is the lack of an approach for quantifying the protection interval provided by the obfuscation algorithm, thus making it difficult to apply in practice (*Jansen*, 2000).

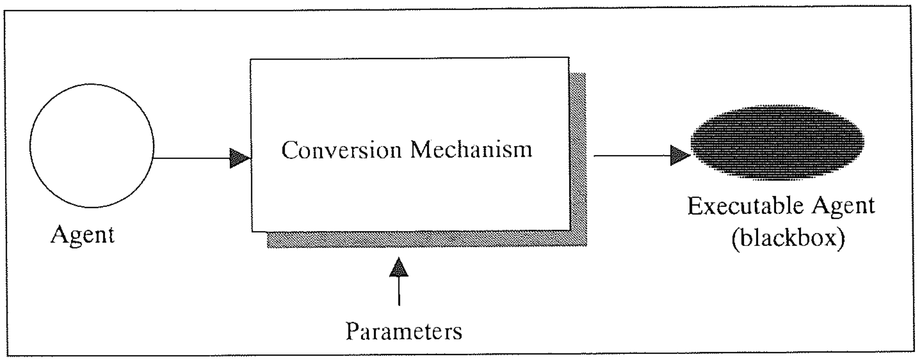


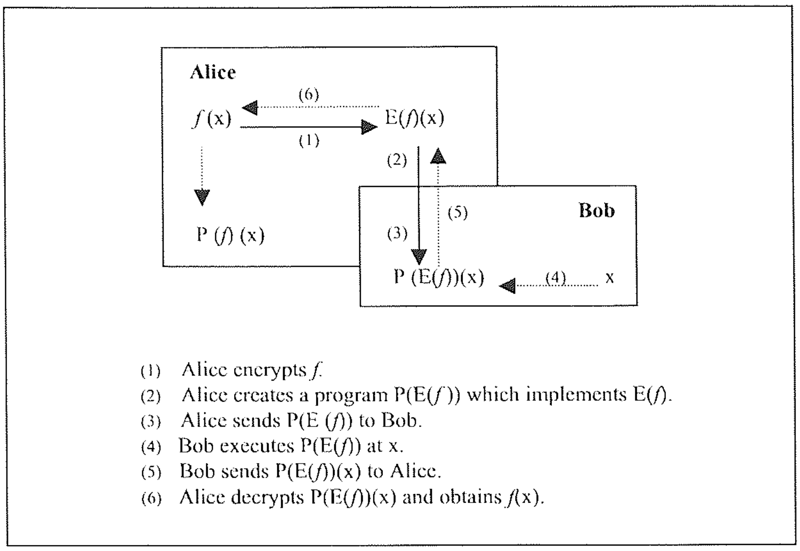
Figure 3.6: A Blackbox Approach - adapted from *Hohl* (1998a)

Mobile Cryptography

Mobile Cryptography is an approach that uses encrypted programs as a method to protect agents against malicious host attacks (*Sander and Tschudin*, 1998b,c,a). This approach provides computation privacy for the agent to safely compute any computation and operate autonomously in untrusted computing environment. The approach requires the execution host to execute an agent program embodying an enciphered function (*Jansen*, 2000).

An example of an approach that uses encrypted programs as in the Mobile Cryptography approach is the Computing with Encrypted Functions approach that has been introduced in (*Sander and Tschudin*, 1998a). Figure 3.7 shows how the computing with encrypted function approach works and the explanation of the approach is described as below:

> Alice has an algorithm to compute a function f. Bob has input x and he willing to compute f(x) for Alice. However, Alice does not want Bob to learn anything about function f. To address this problem, Alice transforms the original function f to the encrypted function E(f) and creates a program P(E(f)), that implements E(f). Since Bob receives an encrypted version of function f, he does not have any knowledge about function f. Alice then sends the program to Bob, the program is embedded within her agent. Bob then runs the agent, which executes P(E(f)) on x, and returns the result to



(1) Alice encrypts f.
(2) Alice creates a program P(E(f)) which implements E(f).
(3) Alice sends P(E (f)) to Bob.
(4) Bob executes P(E(f)) at x.
(5) Bob sends P(E(f))(x) to Alice.
(6) Alice decrypts P(E(f))(x) and obtains f(x).

Figure 3.7: Computing With Encrypted Functions - adapted from *Sander and Tschudin* (1998a)

Alice who decrypts it to obtain f(x). If f is a signature algorithm with an embedded key, the agent has an effective means to sign information without the platform discovering the key. Similarly, if it is an encryption algorithm containing an embedded key, the agent has an effective means to encrypt information at the platform (*Jansen*, 2000).

However, this approach currently supports polynomials and rational functions only and it is hard to find any appropriate encryption schemes to perform the approach (*Sander and Tschudin*, 1998a; *Jansen*, 2000). In addition, the approach also does not prevent attacks such as denial of service, replay, and extraction attack against the agent (*Sander and Tschudin*, 1998a).

Code Obfuscation

Code obfuscation is a collection of approaches to obfuscate the code and the flow of the program (*Collberg and Thomborson*, 2002; *Collberg et al.*, 1997). These approaches can protect an agent code from being reversed engineered by malicious host in order to extract a valuable piece of code or information. The general idea is to make the program look much more complicated than it really is by changing its structure, and complicate the way data is represented.

In order to obfuscate the program, a number of transformations method can be applied, which can result in a new program with the same functionality as the original one but is much harder to analyse. In the literature, there are three types of transformations:

- lexical transformations exchange names with variables, or replace them with names without semantic value.
- control flow transformations are created by inserting special predicates in order to make the flow of the program more complex, while obtaining its original functionality.
- data flow transformations act on data structures by changing storage, encoding, aggregation and order of data.

This code obfuscation approach might provide a medium of protection for an agent against attacks by malicious hosts.

3.6 The Enabling Technology

This section presents the enabling technology that is used to support the implementation of the proposed security mechanisms.

3.6.1 The cryptographic protocol

Cryptographic protocols (also known as security protocols) are essential to protect the applications. They are used to ensure identity (i.e. authentication), guarantee privacy, exchange keys, and for many other purposes.

There are two widely used cryptographic protocols used to support the proposed security mechanism:

- Digital Signatures, and
- The RSA Crypotsystem

A Digital Signatures

A digital signature is a digital code that can be attached to an electronically transmitted message that uniquely identifies the sender. Like a written signature, the purpose of a digital signature is to guarantee that the individual sending the message really is who he or she claims to be.

In this thesis, a combination algorithm between the Secure Hash Algorithm (SHA-1) (*NIST*, 1993, 1994) and the RSA algorithm (*Schneier*, 1996; *Russell and Gangemi*, 1991; *Devargas*, 1993) is used as a signature algorithm.

The SHA-1 algorithm takes a message of less than 2^{64} bits in length and produces a 160-bit message digest. The message digest can then be input to the RSA Algorithm, which generates or verifies the signature for the message (see Figure 3.8 and 3.9 respectively).

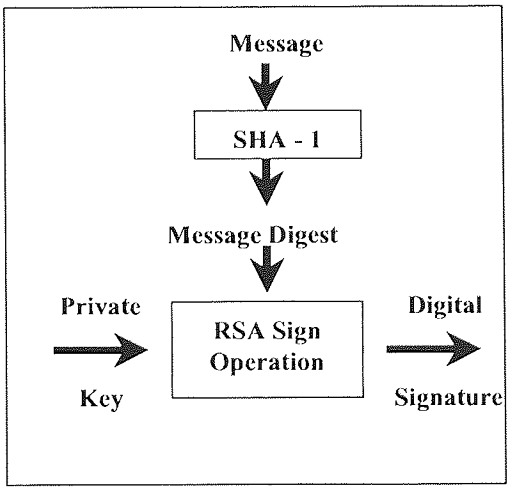


Figure 3.8: Signature Generation using the SHA-1 with the RSA algorithm

The digital signature is generated using the digital signature function in Figure 3.10. This function implements the IAIK-JCE digital signature generator to generate

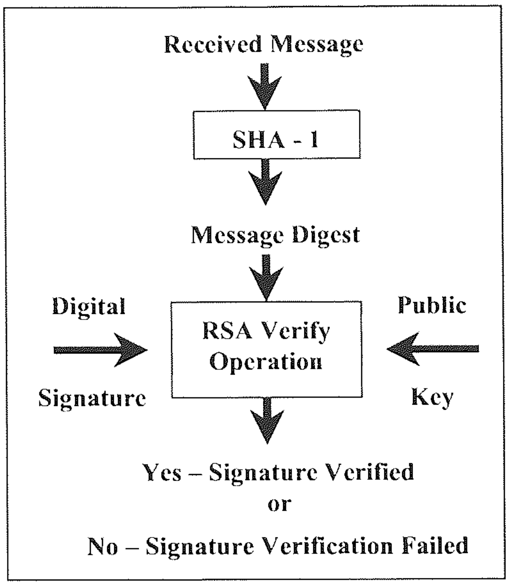


Figure 3.9: Signature Verification using the SHA-1 with the RSA algorithm

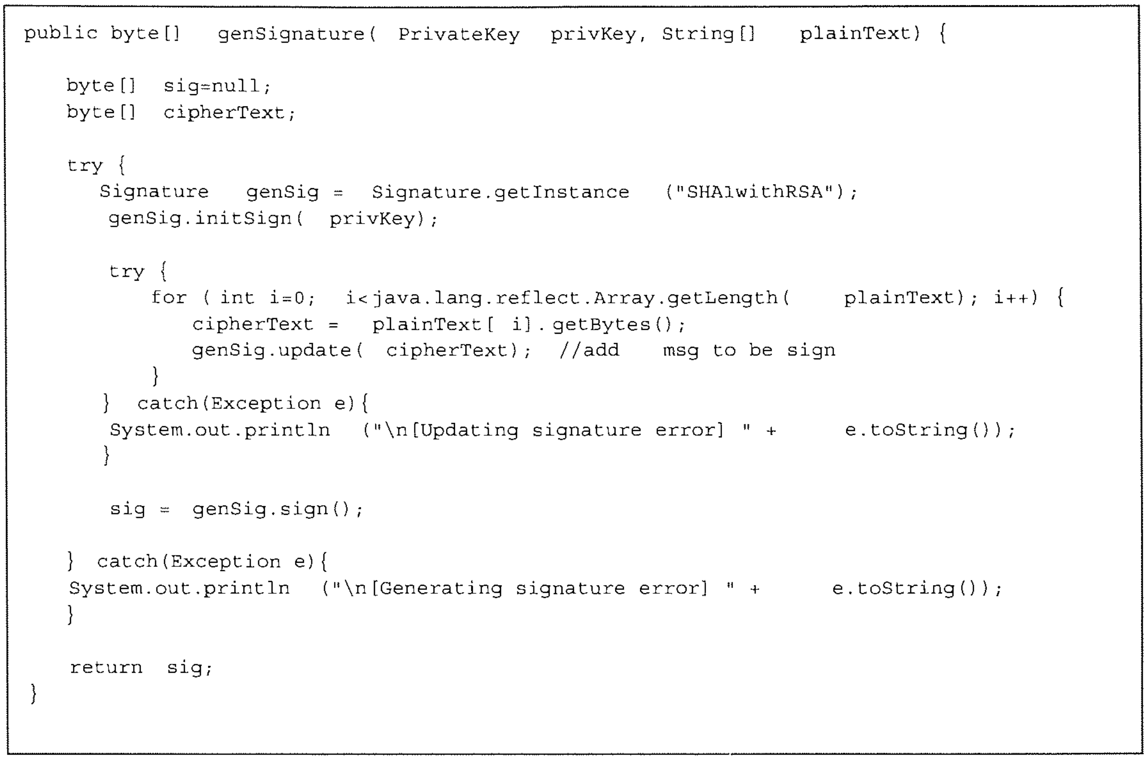
the digital signature using

Signature genSig = Signature.getInstance ("SHA1withRSA");

To identify any unauthorised access from unauthorised parties that try to violate the integrity of the data, all data have to be verified using a digital signature verification function shown in figure 3.11. This function implements the IAIK-JCE digital signature verification to verify the digital signature using

Signature verifySig = Signature.getInstance ("SHA1withRSA");

Both functions are implemented in this thesis to protect the integrity of the data and to prevent a repudiation attack from the malicious host.



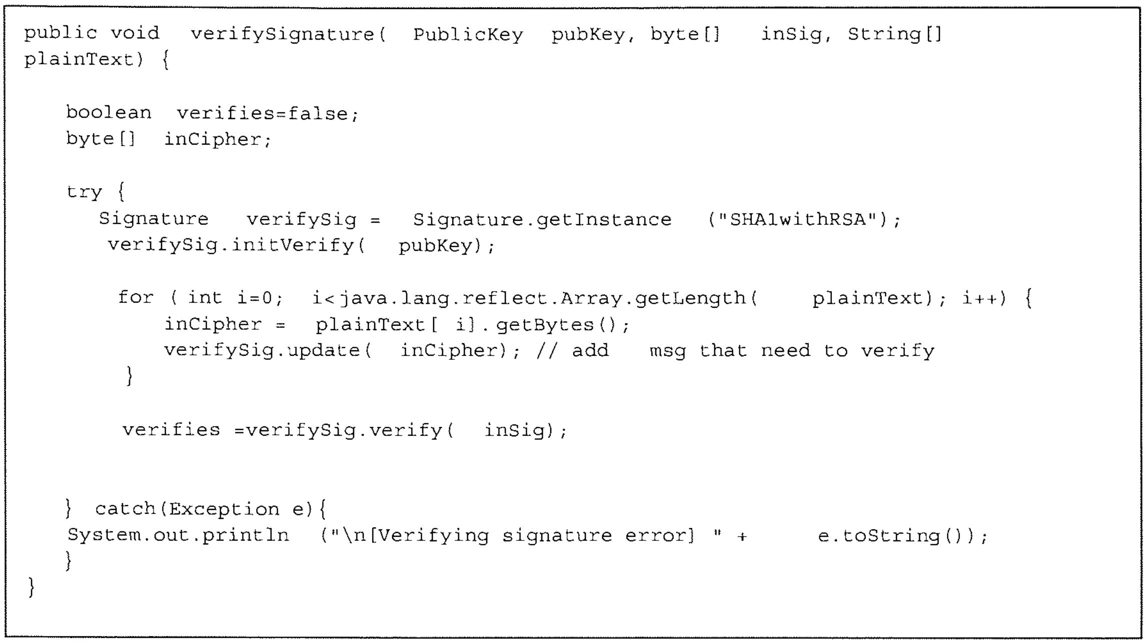
```
public byte[]   genSignature( PrivateKey   privKey, String[]   plainText) {

    byte[]  sig=null;
    byte[]  cipherText;

    try {
       Signature   genSig =  Signature.getInstance   ("SHA1withRSA");
        genSig.initSign(  privKey);

        try {
            for ( int i=0;  i<java.lang.reflect.Array.getLength(    plainText); i++) {
                cipherText =   plainText[ i]. getBytes();
                 genSig.update(  cipherText);  //add    msg to be sign
            }
       }  catch(Exception e){
        System.out.println   ("\n[Updating signature error] " +     e.toString());
       }

        sig =  genSig.sign();

    }  catch(Exception e){
    System.out.println   ("\n[Generating signature error] " +     e.toString());
    }

    return  sig;
}
```

Figure 3.10: A function that generates a digital signature written in Java

The RSA Cryptosystem

The RSA cryptosystem is a public-key cryptosystem that offers both encryption and digital signature (authentication) schemes (*Rivest et al.*, 1998). This cryptosystem enables users of an unsecured public network such as the Internet to securely and privately exchange data through the use of a public and a private cryptographic key pair that is obtained and shared through a trusted authority, e.g. Certificate Authority (CA) host.

The RSA cryptosystem use public key cryptography for authenticating a message sender or encrypting a message. In public key cryptography, a public and a private key are created simultaneously using the same algorithm. The private key is given only to the requesting party and the public key is made publicly available in a certificate authority (CA) host that all parties can access. The private key is never shared with



```
public void   verifySignature(  PublicKey   pubKey, byte[]    inSig, String[]
plainText) {

   boolean  verifies=false;
   byte[]  inCipher;

   try {
     Signature    verifySig =   Signature.getInstance    ("SHA1withRSA");
      verifySig.initVerify(   pubKey);

       for ( int i=0;  i<java.lang.reflect.Array.getLength(     plainText); i++) {
          inCipher =   plainText[ i].getBytes();
          verifySig.update(  inCipher); // add   msg that need to verify
        }

       verifies =verifySig.verify(   inSig);


   } catch(Exception e){
   System.out.println  ("\n[Verifying signature error] " +      e.toString());
   }
}
```

Figure 3.11: A function that verifies a digital signature written in Java

anyone or sent across the Internet.

In this thesis, the RSA cryptosystem is used for the authentication and encryption scheme. For instance, if the Sender (S) wants to send confidential data to the Receiver(R) (see Figure 3.12), using a public key cryptosystem, both the Sender (S_p and S_s) and the Receiver (R_p and R_s) have a pair of keys associated with them, one of which is publicly known (S_p and R_p), the other one only known to the Sender (S_s) or the Receiver (R_s) respectively. In order to be able to encrypt and send the data to the Receiver, the Sender must retrieve the Receiver's public key (R_p) from the Certificate Authority host. The Receiver's public key is used to encrypt data meant to be read by the Receiver; the Receiver can decrypt the result using its private key: $D = R_s(R_p(D))$.

The RSA cryptosystem is implemented using the RSA encryption function shown in figure 3.13. This function using the IAIK-JCE RSA encryption generator as follows:

Cipher cipher = Cipher.getInstance ("RSA", "IAIK");

cipher.init(Cipher.ENCRYPT_MODE,privKey);

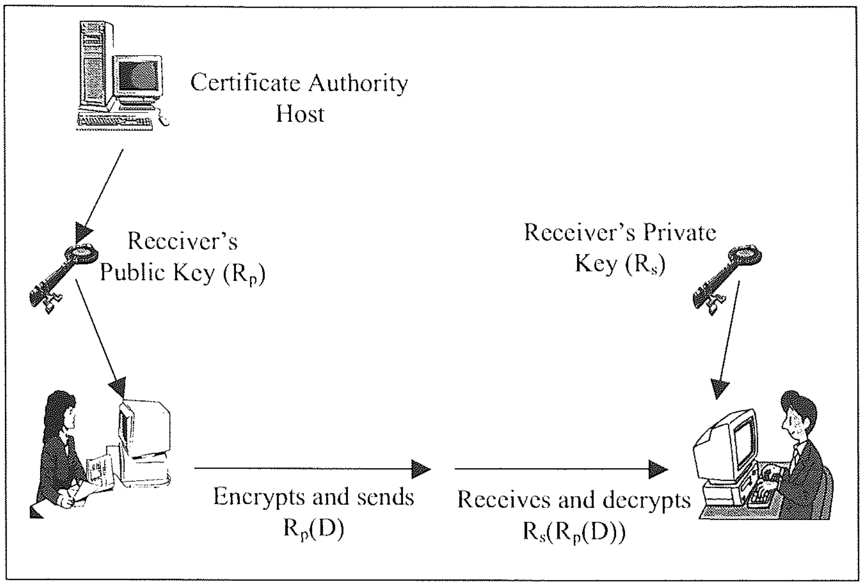

Certificate Authority Host
Receiver's Public Key (R_p)
Receiver's Private Key (R_s)
Encrypts and sends $R_p(D)$
Receives and decrypts $R_s(R_p(D))$

Figure 3.12: Public Key Infrastructure (RSA Encryption and Decryption Method)

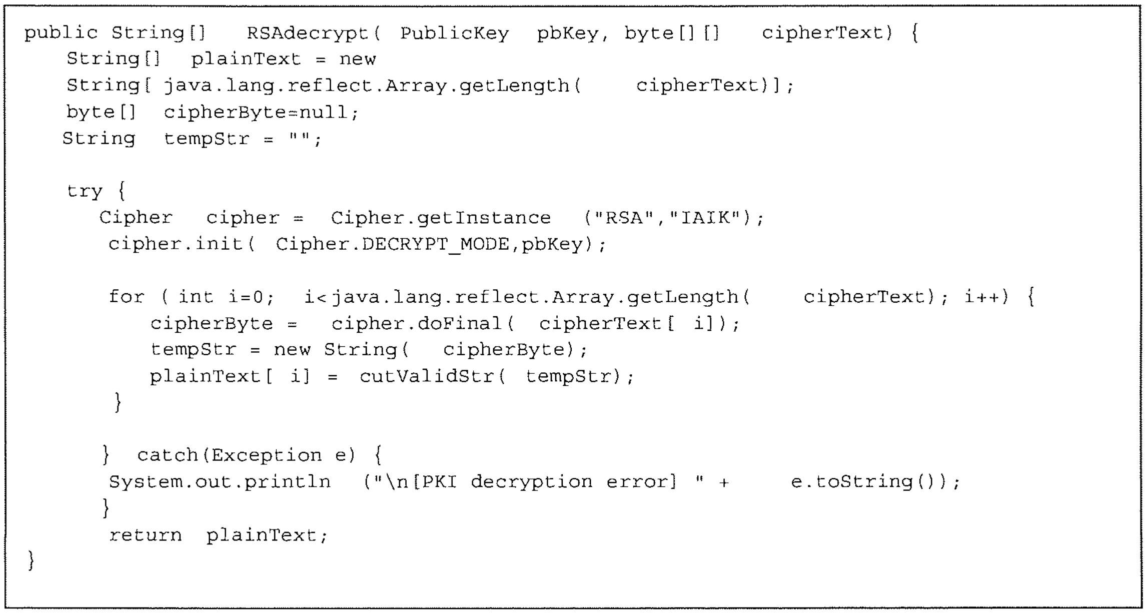
```
public byte[][]   RSAencrypt( PrivateKey  pvKey, String[] text) {
   byte[][]  cipherText=new byte[  java.lang.reflect.Array.getLength(text)][];
   byte[]  textByte=null;
   String  tempStr = "";

   try {
      Cipher  cipher =  Cipher.getInstance  ("RSA","IAIK");
      cipher.init( Cipher.ENCRYPT_MODE,pvKey);

      for( int i=0;  i<java.lang.reflect.Array.getLength(text); i++) {
         tempStr =  addValidStr(text[  i]);
         textByte =  tempStr.getBytes();
         cipherText[ i]=cipher.doFinal(  textByte);
      }

     }  catch(Exception e) {
      System.out.println  ("\n[PKI encryption error] " +    e.toString());
   }
   return  cipherText;
}
```

Figure 3.13: The RSA encryption function written in Java

On the other hand, in order to decrypt the encrypted data, the function in figure 3.14 is implemented. This function implements the IAIK-JCE decryption command as follows:

Cipher cipher = Cipher.getInstance (“RSA”, “IAIK”);

cipher.init(Cipher.DECRYPT_MODE,pubKey);



```
public String[]   RSAdecrypt( PublicKey  pbKey, byte[][]   cipherText) {
   String[]  plainText = new
   String[ java.lang.reflect.Array.getLength(    cipherText)];
   byte[]  cipherByte=null;
   String  tempStr = "";

   try {
      Cipher   cipher =  Cipher.getInstance   ("RSA","IAIK");
      cipher.init( Cipher.DECRYPT_MODE,pbKey);

      for ( int i=0;  i<java.lang.reflect.Array.getLength(    cipherText); i++) {
         cipherByte =   cipher.doFinal( cipherText[ i]);
         tempStr = new String(   cipherByte);
         plainText[ i] =  cutValidStr( tempStr);
      }

      } catch(Exception e) {
      System.out.println   ("\n[PKI decryption error] " +     e.toString());
      }
      return  plainText;
}
```

Figure 3.14: The RSA decryption function written in Java

3.7 Research Formulation, Design and Procedures

3.7.1 Formulation of the research problem

This section formulates the research problem based on the review of literature covered.

Summary of the literature review findings

In this thesis, the literature review on the malicious host problem has led to the recognition that:

- the lack of security protection has severely impeded the widespread use of agent technology (*Coulouris et al.*, 2001; *Jansen*, 2000; *Oppliger*, 1999; *Corradi et al.*, 1999b),

- existing security mechanisms were not devised to address attacks on the application (the agent) by the execution host (the malicious host) (*Hohl*, 2000, 1998a; *Jansen*, 2000), and

- the solubility of the problem of protecting agents against a malicious host attacks is estimated in the literature to be very low (*Oppliger*, 1999; *Corradi et al.*, 1999b; *Hohl*, 1998a).

Based on the investigation and analysis conducted, new security mechanisms are required to protect the agent against malicious host attacks.

In the literature, there are two main requirements that need to be considered in protecting agents against malicious host attacks (*Biehl et al.*, 1998; *Schelderup and Olnes*, 1999; *Sander and Tschudin*, 1998a; *Tripathi and Karnik*, 1998; *Abu Bakar and Doherty*, 2002):

1. **The agent is able to protect the confidentiality of its code, data and state.** The agent is able to prevent the malicious host from learning information about its code, data and state (*Biehl et al.*, 1998; *Schelderup and Olnes*, 1999). This is to guarantee the privacy of the agent against access by unauthorised parties.

2. **The agent is able to protect the integrity of its code, data and state.** The agent is able to guarantee the correctness of its code, data and state at all time (*Biehl et al.*, 1998; *Schelderup and Olnes*, 1999). This is to ensure that the agent code, data and state are not being manipulated by the malicious host during the agent execution process inside the malicious host execution environment and prevent the agents from being "brainwashed" by the malicious host (*Sander and Tschudin*, 1998a).

3.7.2 Research problem

This section outlines the problem, the purpose and the importance of this research.

Statement of the problem

The research problem is to provide security protection for agents that execute inside an untrusted execution host against attacks by any malicious host encountered by the agent. More specifically, to design and develop confidentiality and integrity protection mechanisms for the agent in order to protect the agent's confidentiality and integrity against malicious host attack.

Purpose of study

The purpose of this study is to develop confidentiality and integrity protection mechanisms, which includes:

- the design and evaluation of a confidentiality protection mechanism for protecting the agent against spying attack by the malicious host,
- the design and evaluation of an integrity protection mechanism for protecting the agent against manipulation attack by the malicious host,
- the evaluation of agent architectures and migration patterns, and
- the development of a secured agent-based application that implements the proposed confidentiality and integrity protection mechanism.

3.7.3 Research design and procedure

Research design

This research is conducted in three stages. These stages involved:

- the development of security protection mechanisms to protect the agent against malicious host attacks,

- the evaluation of the proposed agent architecture, migration pattern, and the performance of the proposed security protection mechanisms, and
- the development of an agent-based prototype to implement the proposed security mechanisms.

Focus of the study

This study focuses on the problem of protecting agents against a malicious host attack, specifically, spying and manipulation attacks by the malicious host. The reason for focusing on that is to the inability of the existing security mechanisms to address those kinds of attacks.

Outcomes from the study

The outcomes from this research provide the following:

- a confidentiality protection mechanism that is able to prevent spying attack from the malicious host,
- an integrity protection mechanism that can protect the agents against manipulation attack by the malicious host, and
- an agent-based application that is equipped with security protection that can protect the agents against spying and manipulation attack from the malicious host.

3.8 Concluding remarks

This chapter presents security issues of the agent technology. Four main security areas of the agent technology that impede the wide spread use of agent technology in real world applications have been identified and investigated. However, the work is focused only on the security area involving malicious host because the aspect of securing an application (an agent) against a malicious hosts is a new security issue in current computer system, and the existing security mechanisms cannot be applied to address the issue. Details of the malicious host problem have been discussed including analysis of the malicious host problem, malicious host attacks and existing approaches that address the problem. The formulation of the research problem based on the literature review done, is also presented in this chapter. The statement of the problem, the purpose of study and the importance of the study highlight the need for this research. Finally, the research methodology, research design, limitations and outcome of the study are presented.

Part III

Proposed Security Mechanisms and Prototype

Chapter 4

The Description of The Proposed Security Mechanisms

4.1 Introduction

This chapter describes the proposed security mechanisms for protecting agents against malicious host attack. The proposed security mechanisms aim to protect the confidentiality and the integrity of the agent executing inside an unsecured remote host execution environment, specifically against spying and manipulation attack by the malicious host.

4.2 The proposed security mechanisms

There are two security mechanisms proposed as follows:

- the Random Sequence 3-level obfuscation algorithm, and
- the Recorded State Mechanism.

4.2.1 The Random Sequence 3-level obfuscation algorithm

The spying attack by the malicious host on an agent's code, data and state is one of the attacks that is difficult to solve, because the attack does not leave any trace that could be detected (*Hohl*, 1998a). In addition, a host has to read the agent's code, must have access to agent's data, and must be able to manipulate the agent's variable data in order to execute the agent. Therefore, the host can see all of the agent's code including data and state, thus making difficult any attempt to address malicious spying attack.

In order to overcome spying attack from the malicious host, this thesis proposes an extension to the Conversion of Control Flow Elements into Value-dependent Jumps algorithm (*Hohl*, 1998a). The proposed algorithm is named the Random Sequence 3-level(RS3) obfuscation algorithm (*Abu Bakar and Doherty*, 2003a).

The RS3 obfuscation algorithm consists of multiple polynomial equations for obfuscating the actual value of the agent's critical data to an obfuscated value that is meaningless to the attacker. The polynomial function is selected because the function is a monotonic function, hence suitable for values comparison purposes. A function is said to be monotonic if is either always decreases or always increases. In this work, it is strictly increasing, that is $f(x^{'}) > f(x)$ for $x^{'} > x$. In addition, a function can also be proven monotonic if its first derivative does not change sign. The polynomial function that is used in the work can be proven to be a monotonic function as below:

Given $f(x) = ax^2 + bx + c$, where $a, b, c, x > 0$. Let say a=2, b=500 and c=10.

i. for $x^{'} > x$, then $f(x') > f(x)$; If $x = 5$ and $x^{'} = 10$, $f(5) = 2(5)^2 + 500(5) + 10 = 2560$ and $f(10) = 2(10)^2 + 500(10) + 10 = 5210$. Therefore, for $x^{'} > x$, $f(x^{'}) > f(x)$.

ii. $f^{'}(x)$ does not change sign.

$f^{'}(x) = 2ax + b$ and for $a, b, c, x > 0$, $f^{'}(x) > 0$.

Since the polynomial function used has been proven to satisfy both the conditions for a monotonic function, therefore it is a monotonic function.

This algorithm can only obfuscate numbers and not characters, therefore, the algorithm is only applicable to an agent-based application that carries numbers, such as a shopper agent that buys goods based on the user budget. The RS3 obfuscation algorithm can be divided into three levels, and each level consists of a sequence of three polynomial equations (see Figure 4.1).

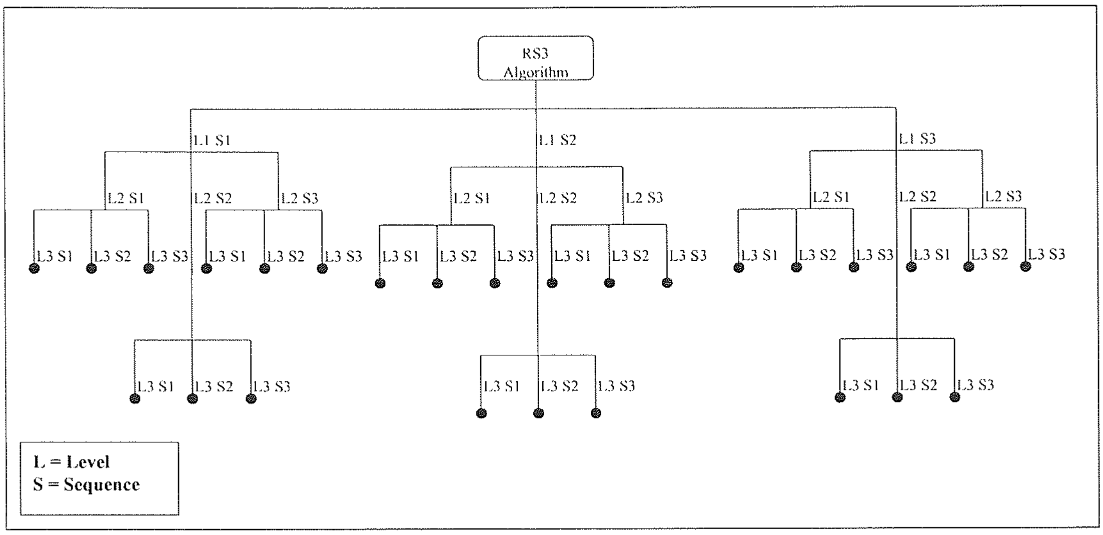
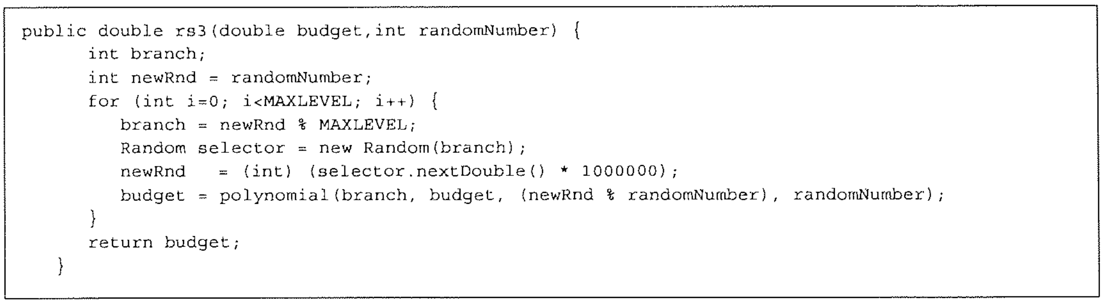


Figure 4.1: The Structure of Random Sequence 3-level obfuscation Algorithm

Only one polynomial function will be selected randomly in each level by executing the *mod* operation[1]. The selected function in each level together with multiple random inputs will produce an obfuscated value that obfuscates the actual value of the agent's critical data (see Figure 4.2).



```
public double rs3(double budget,int randomNumber) {
      int branch;
      int newRnd = randomNumber;
      for (int i=0; i<MAXLEVEL; i++) {
         branch = newRnd % MAXLEVEL;
         Random selector = new Random(branch);
         newRnd   = (int) (selector.nextDouble() * 1000000);
         budget = polynomial(branch, budget, (newRnd % randomNumber), randomNumber);
      }
      return budget;
   }
```

Figure 4.2: A Random Sequence 3-level obfuscation algorithm written in Java (shown partially)

[1] returns the remainder after division of two integer numbers

The RS3 obfuscation algorithm operation

Since RS3 is implemented using master-slave agent architecture and operates on the distributed migration pattern, the operation of the RS3 obfuscation algorithm can be divided into two:

- home operation, and
- remote operation.

The home operation.

The home operation of the RS3 obfuscation algorithm is executed only inside the agent originating host environment. This operation is controlled by the master agent, which executes the conversion process of the RS3 obfuscation algorithm and dispatches the slave agents, which carry an obfuscated value from the conversion process, to the remote hosts. Only the master agent knows the actual value of agent's critical data and no information about the actual value of the agent's critical data leaves the master agent environment. The detailed operation of the RS3 obfuscation algorithm's home operation is described below:

> The home operation of the obfuscation algorithm starts when the user sets up the master agent. The master agent asks the user to provide it with the user maximum budget, e.g. £500. After receiving the user maximum budget, the master agent starts generating its first random number, e.g. 59. The master agent then executes the *mod* operation of the random number with the RS3 obfuscation algorithm maximum level number, which is 3, that is 59 mod 3 = 2. The result is used to determine the selected sequence in the first level of the RS3 obfuscation algorithm. In this example, the remainder of 2 which is the selected sequence number means the third sequence in level 1 is selected because the first sequence in each level starts with zero.

The polynomial function in the selected sequence is given by $f(x) = ax^2 + bx + c$ where the value of the constant a is the selected sequence number, b is the user maximum budget, c is the result of mod operation between a new random number generated inside the current level using the previous selected sequence number (which is 2) as a seed number and a fixed value which is taken to be the first master agent random number obtained that is 59 and x is a fixed value which is the first master agent random number obtained. This polynomial function is used to produce a polynomial result (see figure 4.3) to substitute the value of the user maximum budget as one of the inputs for the next level. For this example, at this stage, $a = 2$, $b = 500$, $c = 731146 \ mod\ 59$ and $x = 59$.

The new random number generated in the current level is used in determining a new sequence in the next level of the RS3 algorithm and the process will continue until all levels of the RS3 algorithm have been executed. The result gathered from the third level of the RS3 algorithm is an obfuscated value of the user maximum budget that is use in the remote operation of the RS3 obfuscation algorithm. Note that the Java random number is guarantee to generate identical sequences of random numbers on different Java Virtual Machine (*Sun Microsystems*, 2004)

After the process of obfuscating user maximum budget is complete, the obfuscated value is transfered to the slave agent, together with the first random number generated by the master agent (which is 59) and the RS3 obfuscation algorithm. The slave agent then is dispatched to the remote host to execute the given tasks.

The remote operation.

The main objective of the RS3 obfuscation algorithm is to enable the operation of comparing confidential values within an unsecured remote host environment without exposing the value to an unauthorised party. An example of a slave agent searching for

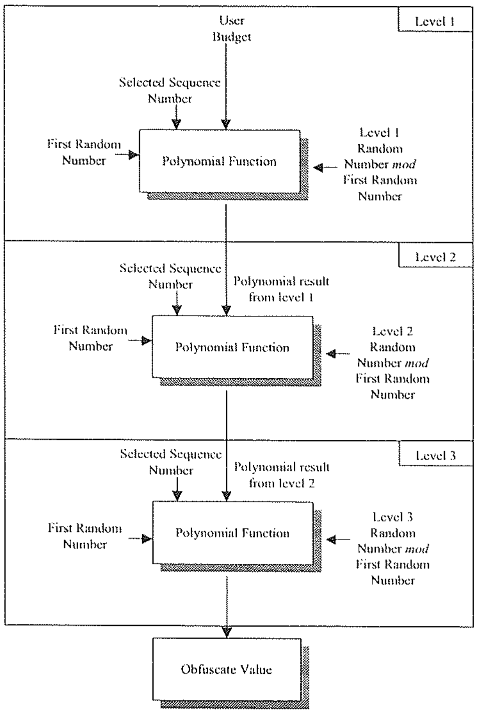


Figure 4.3: The RS3 Obfuscation Algorithm Obfuscation Process

a flight offer is used to show the remote operation of the RS3 obfuscation algorithm. The remote operation of the RS3 obfuscation algorithm starts when the slave agent arrives at the remote host environment. The detailed operation of the RS3 obfuscation algorithm's remote operation is described as below:

> Once the slave agent arrives at the remote host environment, the execution of the slave agent is started by the host. In the execution process, any offer that was gathered from the remote host is converted by the slave agent into an obfuscated value to be used in the comparing operation. The only difference between home operation and remote operation of the RS3 obfuscation algorithm is in generating the first random number that is used to start both obfuscation processes and the value of the user's maximum budget. In the remote operation, the first random number is supplied by the master agent during the initial execution of the home operation which is 59 to ensure the same sequence of polynomial function is selected for

obfuscation operation whereas, in the home operation the random number is generated by the master agent. The RS3 obfuscation algorithm uses the remote host offer in place of the user budget in the obfuscation process shown in Figure 4.3.

The vulnerabilities of RS3 obfuscation algorithm

Although the RS3 obfuscation algorithm is able to obfuscate the actual value of an agent's critical data to make it more difficult for the malicious host to spy, the malicious host can execute multiple copies of the obfuscation algorithm in parallel in order to analyse the algorithm quickly, making this obfuscation algorithm protection vulnerable. This can be addressed by limiting the processing time available to the host before the agent is discarded (*Hohl*, 1998a). However, the problem in determining an effective protection interval that can prevent the malicious host from having enough time to execute multiple copies of the obfuscation algorithm also makes it difficult for this obfuscation algorithm to be implemented in real applications. In order to overcome the problem of malicious host executing multiple copies of the RS3 obfuscation algorithm and of determining an effective protection interval to protect the algorithm, noise code (*Ng and Cheung*, 1999a,b) is introduced for the agent that executes in the remote host environment. The RS3 obfuscation algorithm with noise code is described in the next section.

The Random Sequence 3-level obfuscation Algorithm with Noise Code

The objective for implementing the noise code in the agent application is to hide the actual obfuscated value among a numbers of fake obfuscated values in order to make it more difficult for the malicious host to discover the true value of the agent's critical data (*Ng and Cheung*, 1999a,b). This is due to the fact that in order to discover the true value of the agent's critical data, the malicious host must first discover the actual obfuscated value among a number of fake obfuscated values. Any wrong decision in choosing the obfuscated value will result in determining a wrong value for the agent's critical data. For instance, if the agent is equipped only with the actual obfuscated value, X without adding any noise code, the probability that the malicious host could discover the actual obfuscated value by searching a range of values is one, i.e. $P(X) = 1$. However, if noise codes E_i (fake obfuscated values) are added to the agent, where $i = 1, 2, \ldots, 100 - 1$, the probability of discovering the actual obfuscated value is $\frac{1}{100}$. Hence, the probability of guessing the actual obfuscated value gets smaller as more noise codes are added, i.e. $P(X) \rightarrow 100$. This will make the actual obfuscated value difficult to guess because the malicious host never knows the right obfuscated value. Figure 4.4 illustrates the effect of introducing noise codes into the agent application. In addition, the time needed to guess the actual obfuscated value will delay the malicious host in analysing the obfuscation algorithm. Therefore, the use of an effective protection interval in enhancing the obfuscation algorithm protection could be less important.

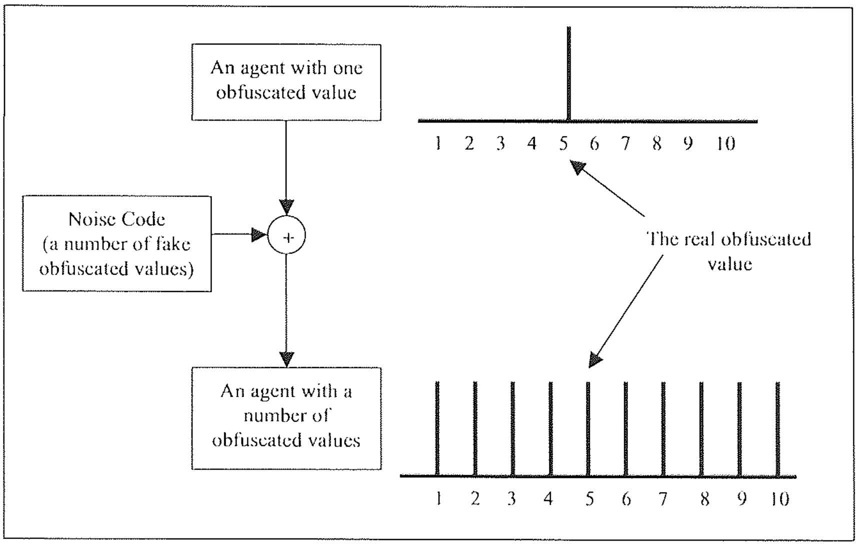
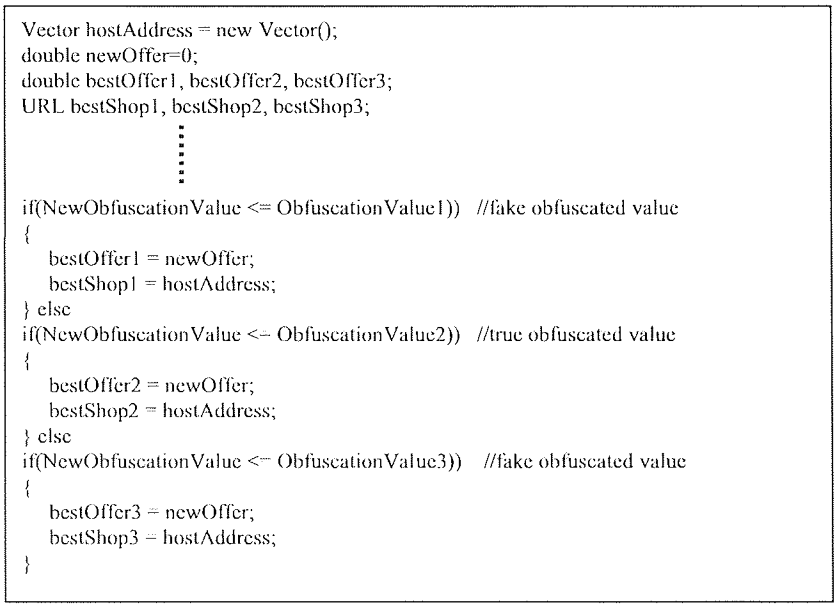

An agent with one obfuscated value
1 2 3 4 5 6 7 8 9 10
Noise Code (a number of fake obfuscated values)
+
The real obfuscated value
An agent with a number of obfuscated values
1 2 3 4 5 6 7 8 9 10

Figure 4.4: The Effect of Adding Noise Codes Into The Agent Application - adapted from *Ng and Cheung* (1999a,b)

Implementing RS3 obfuscation Algorithm with Noise Code

The operation of the RS3 obfuscation algorithm with noise code is almost the same as the operation of the RS3 obfuscation algorithm without noise code. The only difference between these two obfuscation algorithms is in the number of obfuscated values generated and added by the master agent into the slave agent application before the slave agent is dispatched to the remote host execution environment to execute its tasks.

In the operation of the RS3 obfuscation algorithm without noise code, the master agent only has to obfuscate the value of the user's budget and add the obfuscated value into the RecordedReadOnly container before dispatching the slave agent to execute its tasks in the remote host execution environment. However, in the operation of RS3 obfuscation algorithm with noise code, the master agent has to generate more than one obfuscated values, which serve as noise codes and add these obfuscated values into the RecordedReadOnly container before dispatching the slave agent to execute in the remote host execution environment (see figure 4.5).



```
Vector hostAddress = new Vector();
double newOffer=0;
double bestOffer1, bestOffer2, bestOffer3;
URL bestShop1, bestShop2, bestShop3;
                      ⋮

if(NewObfuscationValue <= ObfuscationValue1))   //fake obfuscated value
{
   bestOffer1 = newOffer;
   bestShop1 = hostAddress;
} else
if(NewObfuscationValue <= ObfuscationValue2))   //true obfuscated value
{
   bestOffer2 = newOffer;
   bestShop2 = hostAddress;
} else
if(NewObfuscationValue <= ObfuscationValue3))    //fake obfuscated value
{
   bestOffer3 = newOffer;
   bestShop3 = hostAddress;
}
```

Figure 4.5: A Slave Agent Program added with Noise Code (data block)

To illustrate, the noise code is an obfuscated value that is generated by the master agent from a fake user budget value. This fake user budget value is created by adding or subtracting a random number from the actual user budget value. For example, let say the actual user budget value is 500. To create a fake user budget value, the master agent needs to generate a random number, e.g. 176, and add or subtract the random number from the real user budget value. If the master agent chooses to add the random number to the real user budget value, the fake user budget value becomes 676. This value is then obfuscated using the RS3 obfuscation algorithm. On the other hand, if the master agent chooses to subtract the random number from the real user budget value, the fake user budget value becomes 324. The RS3 obfuscation algorithm will be applied to obfuscated that value. To generate more spurious obfuscated values, the master agent has to generate more random numbers and repeat the process described above.

Once the obfuscation process in the home host is completed, the master agent dispatches the slave agent together with all the generated obfuscated values to the remote host to execute its given tasks. In the remote host execution environment, the slave agent starts its execution process by converting any offer that was gathered from the remote host into an obfuscated value to be used in the comparing process. For example, if the slave agent has 4 obfuscated values (one real and three fake values), the slave agent has to execute 4 comparisons comparing the offer with each budget value.

4.2.2 The Recorded State Mechanism

To address manipulation attacks from an execution host, this thesis proposed the Recorded State Mechanism (RSM) (*Abu Bakar and Doherty*, 2002, 2003b). The RSM is developed from features of Reference States Mechanism (*Hohl*, 2000), Cryptographic Traces (*Vigna*, 1998) and State Appraisal (*Farmer et al.*, 1996a). However, instead of using the next host in the agent travel sequence to check the recorded state, where good execution performance is achieved but the agent is exposed to collaboration attacks, the RSM mechanism uses the master agent, resident in the home host, to do the checking, and multiple slave agents to do the travel, thus preventing a collaboration attack. Furthermore, the checking process is done automatically in every case without waiting for a suspicious owner to start it, giving prompt warning that an attack has occurred.

The mechanism uses Java object serialisation to capture the state information of the agent. This is due to the fact that Java object serialisation offers an easy way to capture the state of Java objects that exist in the agent application (*Funfrocken*, 1998).

The Recorded State Mechanism Containers

The RSM consists of three different kinds of container that are provided to an agent, executing inside a remote host execution environment. These different kinds of container are used to separate out three different type of data: read-only, execute-only and collect-only data, which are carried and collected by the agent inside a remote host execution environments. The categorisation of different types of data using these containers is useful for the RSM evaluation process because it helps the evaluation process to easily identify and analyse specific data in a short time. The three containers are:

- RecordedReadOnly container,
- RecordedExecuteOnly container,
- RecordedCollectOnly container.

The RecordedReadOnly container

An agent's program often contains some read-only data as part of its state. For instance, the user's purchase requirement data in a buyer agent should not be modifiable by anyone other than its owner, and thus are read-only during the agent's travels. Without any security protection, this read-only data could easily be tampered with by the malicious host. This is due to the fact that the execution host is able to access all parts of the agent during the agent execution session and has knowledge on the physical location of the agent data in its memory (*Hohl*, 1997). Thus even though the agent data is known to the execution host, it is essential to have a security mechanism that can protect this data.

In the RSM, the agent's read-only data is contained in the RecordedReadOnly container. In order to protect its integrity, the read-only data inside the container is digitally signed by the owner of the data and then encrypted with a particular host's public key before it leaves the owner host. This is to guarantee the integrity of the data and to avoid any unauthorised parties such as the malicious host easily accessing or tampering with the data without being detected by the agent owner.

The RecordedExecuteOnly container

Once the agent arrives at a remote host execution environment, the agent is fully under control of the remote host. Thus it opens greater opportunities for the remote host to abuse the agent. To overcome this problem, the execution activities of the agent inside the remote host execution environment need to be recorded. This is to enable the owner of the agent to detect any malicious activities that might occurred during the agent execution process inside the remote host execution environment and to provide proof in case of tampering that an alleged operation of the agent could have never been performed.

In the RSM, the execution activities of the agent are recorded in the RecordedExecuteOnly container. The owner of the agent set the agent to store specific agent state information such as the agent's execution results, i.e. Flight not available or Flight found, executing agent's ID, and executing agent's data. The agent execution activities continue to be recorded into the RecordedExecuteOnly container until the agent has completed its tasks in the remote host environment. Before the agent returns to its originating host, the RecordedExecuteOnly container is digitally signed by the remote host.

The RecordedCollectOnly container

The RecordedReadOnly container has a limited utility in that it only serves the kind of data that remains constant throughout the agent's travels. In some situations, the agent needs to collect data such as an offer or input data from the remote host that it visits.

In the RSM, collectable data is stored into the RecordedCollectOnly container. This container will store any input received from the host during the agent execution process. In order to prevent any subsequent modification from unauthorised parties or the malicious host, and to guarantee the integrity of the collected data, the data inside the container needs to be digitally signed by the remote host before the agent returns to its originating host.

The Recorded State Mechanism operation

Since the Recorded State Mechanism is implemented using master-slave agent architecture and operates on the distributed migration pattern, the operation of the Recorded State Mechanism can be divided into three:

- home operation,
- remote operation, and
- evaluation operation.

Home operation

The home operation of the RSM (Recorded State Mechanism) starts when the master agent that is executing inside the home host execution environment executes the mechanism. During this operation (home operation), the master agent will store the read-only data such as the user's purchase requirement in the RecordedReadOnly container, digitally sign the container and encrypt it using the targeted remote host's public key that was retrieved by the master agent from the Certificate Authority (CA) host. The master agent then equips each of the slave agents with a RecordedReadOnly container, a RecordedExecuteOnly container, a RecordedCollectOnly container, master agent's public key, and targeted remote host address before dispatching the slave agents to the targeted remote host.

Remote operation

The remote operation of the Recorded State Mechanism starts when the slave agent arrives at the remote host environment. Once the slave agent has been started by the remote host, the slave agent starts its execution process to execute its given tasks.

During the slave agent execution process, the execution data of the slave agent is recorded by the slave agent in a RecordedExecuteOnly container. In this process, specific execution data (e.g. executing agent's ID, data and originating host information), which have been determine previously by the master agent for the slave agent to record, will be recorded by the slave agent. On the other hand, any input data from the remote host including the result is recorded in a RecordedCollectOnly container.

Upon returning to the home host, the remote host signs the RecordedExecuteOnly and the RecordedCollectOnly containers.

Evaluation operation

The final operation of the Recorded State Mechanism, which is evaluating the RecordedReadOnly, RecordedExecuteOnly and RecordedCollectOnly containers, starts when the slave agent returns to the home host. This operation is executed by the master agent to detect manipulation attack by the malicious host that might attack the slave agent during the slave agent execution process inside the remote host execution environment. There are two-sub evaluation processes that will be executed:

1. digital signatures verification, and

2. data verification.

The digital signatures verification is a process that is executed to check the integrity of the RecordedReadOnly, RecordedExecuteOnly and RecordedCollectOnly data. In this process the master agent uses the visiting remote host public key to verify the RecordedReadOnly, RecordedExecuteOnly and RecordedCollectOnly containers digital signatures. An unverified digital signature will indicate that the containers have been tampered. This will cause the slave agent including its data and state to be discarded, and the visited remote host address is added to the blacklist address database. Otherwise, the containers are forwarded to the data verification process.

The data verification is a process to detect attacks by the malicious host. In this process, the Recorded State Mechanism containers entries will be analysed by the master agent. The entries in the RecordedReadOnly, RecordedExecuteOnly and RecordedCollectOnly will be compared. Any mismatch entries indicates that an attack had happened.

One possibility is to check for manipulation attack, where the offer value in the RecordedExecuteOnly and RecordedCollectOnly containers are not the same. For example, lets say the user budget in the RecordedReadOnly container is set at 150 pounds, and the offer given by the remote host in the RecordedCollectOnly is 100 pounds which is within the user budget. However, the lowest offer recorded by the RecordedExecuteOnly container is 170 pounds. As mentioned earlier, different offer values in the RecordedCollectOnly and RecordedExecuteOnly containers show that an attack had happened.

Another possibility is to check for incorrect execution attack by re-executing the Random Sequence 3-level obfuscation algorithm on the offer value given in the RecordedCollectOnly container. If the obfuscated value obtained is not the same as the obfuscated value recorded in the RecordedExecuteOnly container that means that incorrect execution attack has occurred.

Checking for manipulation attack on the read only data is another possibility. One example is an attack on the user budget value. The user budget value in the RecordedReadOnly container is compared with the user budget value in the RecordedExecuteOnly container. Any mismatch in the value means that manipulation attack had occurred since the read only data should be the same.

Cloning attack is another possible attack that can be traced by the Recorded State Mechanism. This can be done by comparing the agent identification number recorded in the RecordedReadOnly container, which is recorded by the agent before it is dispatched to the remote host with the agent identification number in the RecordedExecuteOnly container, which is recorded during the agent execution process in the remote host environment. If the number is not the same, it means that cloning attack has happened.

If any of the mentioned attacks above are detected, the slave agent's data and state will be discarded and the visited remote host address will be added to the blacklist address database.

4.3 Concluding remarks

This chapter presented the description of the Random Sequence 3-level obfuscation algorithm and the Recorded State Mechanism. The RS3 obfuscation algorithm provides us with a mechanism to hide information and prevent a spying attack. The Recorded State Mechanism provides us with the potential to detect manipulation attacks.

Chapter 5

The Secure Flight Finder Agent-Based System

5.1 Introduction

This chapter describes the design and development of a Secure Flight Finder Agent-Based System (SecureFAS), which serves as a test bed system for implementing and testing the proposed security mechanisms.

The objective of the SecureFAS is to search for the best flight offer (lowest flight offer) among unsecured virtual airline hosts in the Internet that is managed by untrusted parties. The scenario used is slightly more complex than the simple lowest flight offer scenario mentioned earlier. This is due to the SecureFAS design requirements that will be explained in the next section.

5.2 The SecureFAS design requirements

One of the aims of this research is to design and develop a prototype system to implement the proposed security mechanisms (Aim 3). To meet this aim, the design requirements for this prototype system are:

1. the ability to simulate a real world application, and

2. the ability to test the functionality of the Random Sequence 3-level obfuscation algorithm and the Recorded State Mechanism.

The first SecureFAS design requirement is fulfilled by selecting a flight finder scenario for a real world application simulation due to the following reasons:

- the selected scenario has been used in many real world applications (*FlightFound*, 2004; *Travelocity*, 2004; *Couriertravel*, 2004),
- the selected scenario has been used as an example in the literature in order to explain the malicious host problem (*Hohl*, 1997; *Berkovits et al.*, 1998; *Farmer et al.*, 1996a; *Yee*, 1997),
- the selected scenario requirement for an agent together with its sensitive information such as user maximum budget and user purchase requirement to be transferred to the remote host in order to find a flight information and fare, enables a malicious host attack scenario to be illustrated.

The second SecureFAS design requirement is fulfilled by implementing both security mechanisms in the SecureFAS prototype system to find the best flight offer among many other virtual airline hosts in the Internet. The Random Sequence 3-level obfuscation algorithm is used to prevent the data from being spied upon, and the Recorded State Mechanism is used to detect manipulation attacks.

5.3 The SecureFAS design

The SecureFAS is designed from the integration of two sub systems (components):

- the SecureFAS Master Agent system, and
- the SecureFAS Slave Agent system.

5.3.1 The SecureFAS Master Agent system

This section describes the design of the SecureFAS Master Agent system that is shown in Figure 5.1. The SecureFAS Master Agent system process consists of:

- Initialisation,
- Cryptographic key registration,
- Filtering bad addresses,
- Cryptographic key retrieval,
- Obfuscating data,
- Storing read-only data,
- Generating and dispatching slave agent,
- Evaluation, and
- Compare offers collected by the slave agents.

Initialisation

The initialisation process of the SecureFAS Master Agent system begins when the SecureFAS user starts up the SecureFAS Prototype application. Once the SecureFAS Prototype application has been started up, the SecureFAS Master Agent starts generating a public and a private cryptographic key by executing the RSA cryptosystem algorithm. This algorithm produces 1024 bit private and public keys that are useful for implementing digital signature and public key infrastructure (PKI) used throughout the SecureFAS operation. After generating a pair of cryptographic keys, the SecureFAS Master Agent prompts the user to input the user's specific purchase requirement and the list of virtual airline hosts addresses. The user's specific purchase requirement contains the user's flight information, such as the flight destination and its origin, the departure date, the passenger information including the number of passengers and the

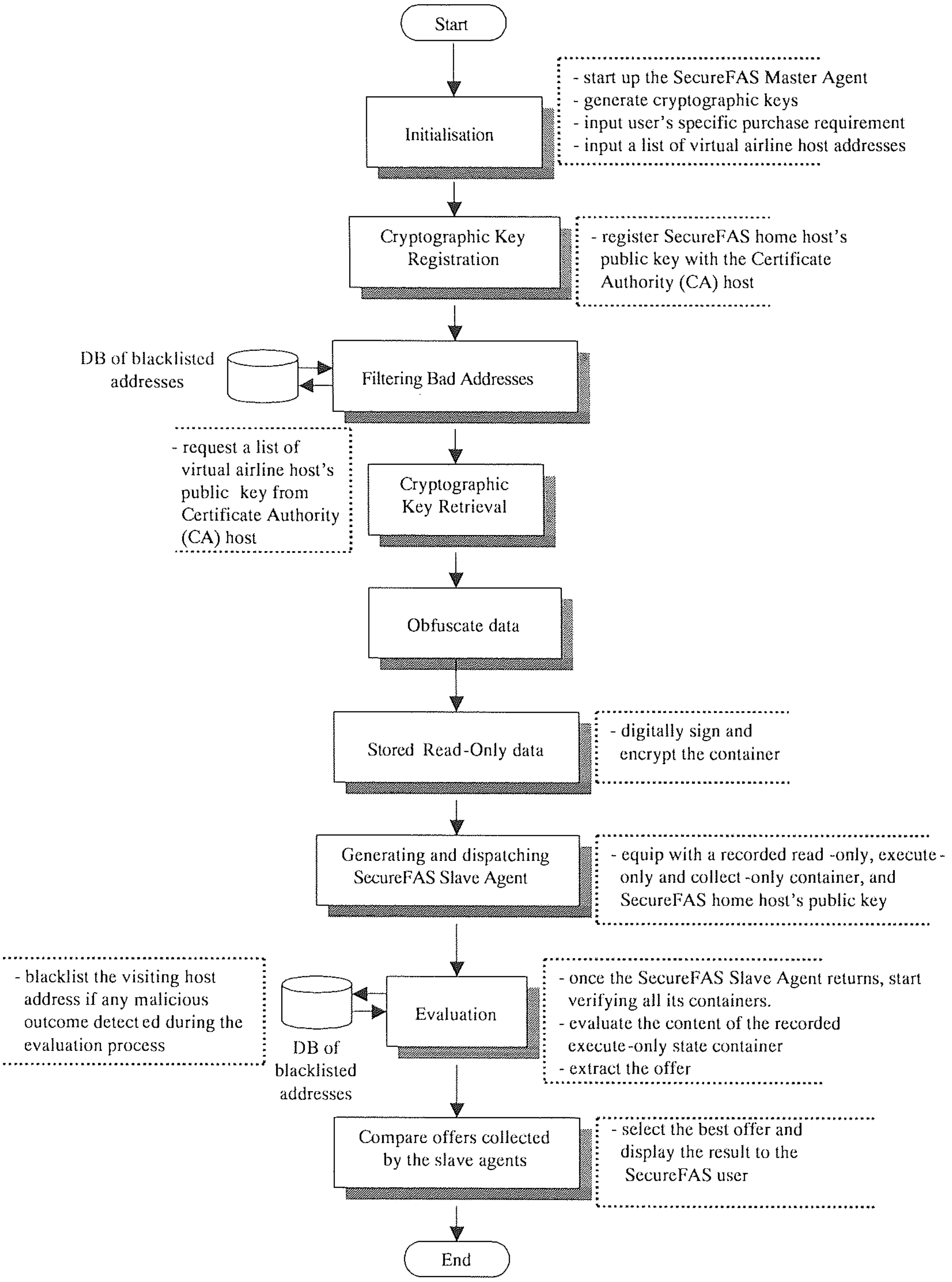

Start
Initialisation
- start up the SecureFAS Master Agent
- generate cryptographic keys
- input user's specific purchase requirement
- input a list of virtual airline host addresses
Cryptographic Key Registration
- register SecureFAS home host's public key with the Certificate Authority (CA) host
DB of blacklisted addresses
Filtering Bad Addresses
- request a list of virtual airline host's public key from Certificate Authority (CA) host
Cryptographic Key Retrieval
Obfuscate data
Stored Read-Only data
- digitally sign and encrypt the container
Generating and dispatching SecureFAS Slave Agent
- equip with a recorded read -only, execute-only and collect -only container, and SecureFAS home host's public key
- blacklist the visiting host address if any malicious outcome detect ed during the evaluation process
DB of blacklisted addresses
Evaluation
- once the SecureFAS Slave Agent returns, start verifying all its containers.
- evaluate the content of the recorded execute-only state container
- extract the offer
Compare offers collected by the slave agents
- select the best offer and display the result to the SecureFAS user
End

Figure 5.1: The SecureFAS Master Agent System

passenger type (such as adult, child or infant), and the user's maximum budget. The list of virtual airline hosts addresses contains the suggested virtual airline hosts addresses to be visited by the agent in order to find the best flight offer that fulfils the user's specific purchase requirement.

Cryptographic key registration

To enable each of the participant hosts in the SecureFAS implementation environment to securely and privately exchange data in the Internet, each of them needs to register their public key with the Certificate Authority (CA) host. In the SecureFAS Prototype application system, the SecureFAS Master Agent will register the SecureFAS public key to the CA host to enable other hosts to share or exchange data with it. For example, in order to verify the SecureFAS's data, which has been digitally signed by the SecureFAS Master Agent using the SecureFAS private key, the other hosts need to obtain the SecureFAS's public key from the CA host. Only a valid SecureFAS's public key from the CA host is able to verify the SecureFAS's digital signature.

Filtering bad addresses

The aim of this process is to prevent the SecureFAS Slave Agent from migrating to a host that is suspected to be the malicious host. To fulfil this aim, once the SecureFAS Master Agent had received a list of virtual airline hosts' addresses from the SecureFAS user, the list has to be filtered by the SecureFAS Master Agent by comparing it with the SecureFAS host's blacklist address database. The SecureFAS Master Agent will remove the host address that matches the SecureFAS host's blacklist address database, and forward only "clean" addresses (addresses that do not match with the SecureFAS host's blacklist addresses) to the next stage.

Cryptographic key retrieval

Since the SecureFAS prototype system implements the public key infrastructure (PKI) to securely and privately exchange data in the Internet using a public and a private cryptographic key, the public key of each participating virtual airline hosts needs to be obtained from a CA host. This is essential for encrypting the SecureFAS data before the data can be dispatched to a particular virtual airline host together with the SecureFAS Slave Agent. However, the SecureFAS Master Agent will obtain only a list of clean host's public keys from the CA host and these keys are forwarded to the next stage.

Obfuscating data

To prevent the malicious host from spying on the SecureFAS Slave Agent's confidential data during the SecureFAS Slave Agent execution session inside the malicious host execution environment, the SecureFAS Master Agent uses an obfuscating mechanism. The obfuscation mechanism is used by the SecureFAS Master Agent to obfuscate the user's maximum budget. Before the SecureFAS Slave Agent is dispatched by the SecureFAS Master Agent to the remote host to execute it's given tasks, the user's maximum budget is converted by the obfuscating mechanism into an obfuscated value. The obfuscated value is carried out by the SecureFAS Slave Agent into the remote host execution environment. The true value of the user's maximum budget will remain with the SecureFAS Master Agent inside the SecureFAS host. The obfuscated value from this stage is forwarded to the next stage for further process.

Storing read-only data

In order to prevent unauthorised parties from eavesdropping on SecureFAS data, especially the read-only data that contains the user specific purchase requirement, in transit or gaining access to the data by pretending to be the destination host, the SecureFAS read-only data is stored in the RecordedReadOnly container in an encrypted form, after being digitally signed by the SecureFAS Master Agent. The read-only data is digitally signed by the SecureFAS Master Agent using the SecureFAS host's private key and encrypted using the destination host's public key that was obtained from the CA host. Only the SecureFAS host's public key can be used to verify the read-only data integrity and only the destination host's private key can be used to decrypt the encrypted read-only data. Once the read-only data has been digitally signed and encrypted by the

SecureFAS Master Agent, the RecordedReadOnly container is forwarded to the next stage for further process.

Generating and dispatching Slave Agent

The use of the master-slave agent architecture in the SecureFAS prototype application allows the SecureFAS Master Agent to delegate tasks to SecureFAS Slave Agents in order to increase the SecureFAS performance in term of its processing speed. Once the SecureFAS Master Agent receives a list of the virtual airlines' public keys from the cryptographic key retrieval phase, the SecureFAS Master Agent starts generating the SecureFAS Slave Agents to delegate the user tasks. The number of the SecureFAS Slave Agents generated by the SecureFAS Master Agent is based on the number of the virtual airline host's public keys received from the cryptographic key retrieval phase. In order to enable each of the SecureFAS Slave Agents to execute the given tasks inside a particular virtual airline host execution environment, each of the SecureFAS Slave Agents is equipped with the Recorded State mechanism, which contains the RecordedReadOnly container, the RecordedExecuteOnly container and the RecordedCollectOnly container, the obfuscation algorithm, and the SecureFAS host's public key. The SecureFAS Master Agent then dispatches each of the SecureFAS Slave Agents in parallel to a particular virtual airline hosts and waits until the SecureFAS Slave Agent finishes its task and returns to the SecureFAS host.

Evaluation

The evaluation process of the SecureFAS prototype system is required in order to carry out the evaluation phase of the Recorded State Mechanism (see to Section 4.2.2). This process is started when the SecureFAS Master Agent receives the returning SecureFAS Slave Agent. The evaluation process on the returning SecureFAS Slave Agent can be divided into two main processes as follows:

1. the SecureFAS Evaluation Agent (generated by the SecureFAS Master Agent for the evaluation process) verifies the signature on three Recorded State Mechanism

containers, the RecordedReadOnly, the RecordedExecuteOnly and the RecordedCollectOnly using the visited virtual airline host's public key obtained by the SecureFAS Master Agent from the CA host. If any of the signatures could not be verified, the SecureFAS Slave Agent, including its data and state, will be discarded and the visited virtual airline host address is added to the blacklist address database. This action is taken in order to filter any malicious host interference with the SecureFAS Slave Agent result. Once both Recorded State Mechanism containers have been verified, the SecureFAS Evaluation Agent will start the final evaluation process on the SecureFAS Slave Agent.

2. the SecureFAS Evaluation Agent analyses the contents of the RecordedExecuteOnly container and RecordedCollectOnly container that were recorded during the SecureFAS Slave Agent execution process and also the contents of the Master Agent's RecordedReadOnly container that were created by the Master Agent before the SecureFAS Slave Agent was dispatched.

   The evaluation process is done by the master agent by comparing, for example, the slave agent's ID recorded inside the RecordedReadOnly container with the slave agent's ID recorded inside the RecordedExecuteOnly container. If any mismatch result is produced from the process, the SecureFAS Slave Agent, including its data and state, will be discarded and the visited virtual airline host address is added to the blacklist address database. This is to guarantee that the SecureFAS Slave Agent result is genuine and free from any malicious host attacks. On the other hand, if the analysis process is successful, the offer and the virtual airline host information are extracted from the SecureFAS Slave Agent and stored in the SecureFAS Master Agent's result container. For other examples on detecting the malicious host attack, see section 4.2.2.

The evaluation process on the returning SecureFAS Slave Agent will continue until all the SecureFAS Slave Agents have returned or the evaluation time set by the SecureFAS Master Agent has expired.

Compare offers collected by the slave agents

This process begins once the SecureFAS Master Agent has finished evaluating the SecureFAS Slave Agents. In order to select the best offer, that is the lowest flight price, the SecureFAS Master Agent browses all the SecureFAS Slave Agent's results in the result container. Only the best is selected and the virtual airline host information for the best offer is extracted from the SecureFAS Master Agent result container. This information is displayed to the SecureFAS user.

5.3.2 The SecureFAS Slave Agent system

This section describes the design of the SecureFAS Slave Agent system shown in Figure 5.2. The SecureFAS Slave Agent system process consists of:

- Initialisation,
- Establish connection,
- Decrypt and verify,
- Search flight date,
- Search flight, and
- Return to SecureFAS host.

Initialisation

The initialisation process of the SecureFAS Slave Agent system begins when the destination host starts up the SecureFAS Slave Agent application. Once the SecureFAS Slave Agent has been started up, each of the processes executed by the SecureFAS Slave Agent in the destination host execution environment will be recorded in the RecordedExecuteOnly container and any input received from the destination host will be recorded in the RecordedCollectOnly container, until the SecureFAS Slave Agent leaves the destination host execution environment to return to the SecureFAS host.

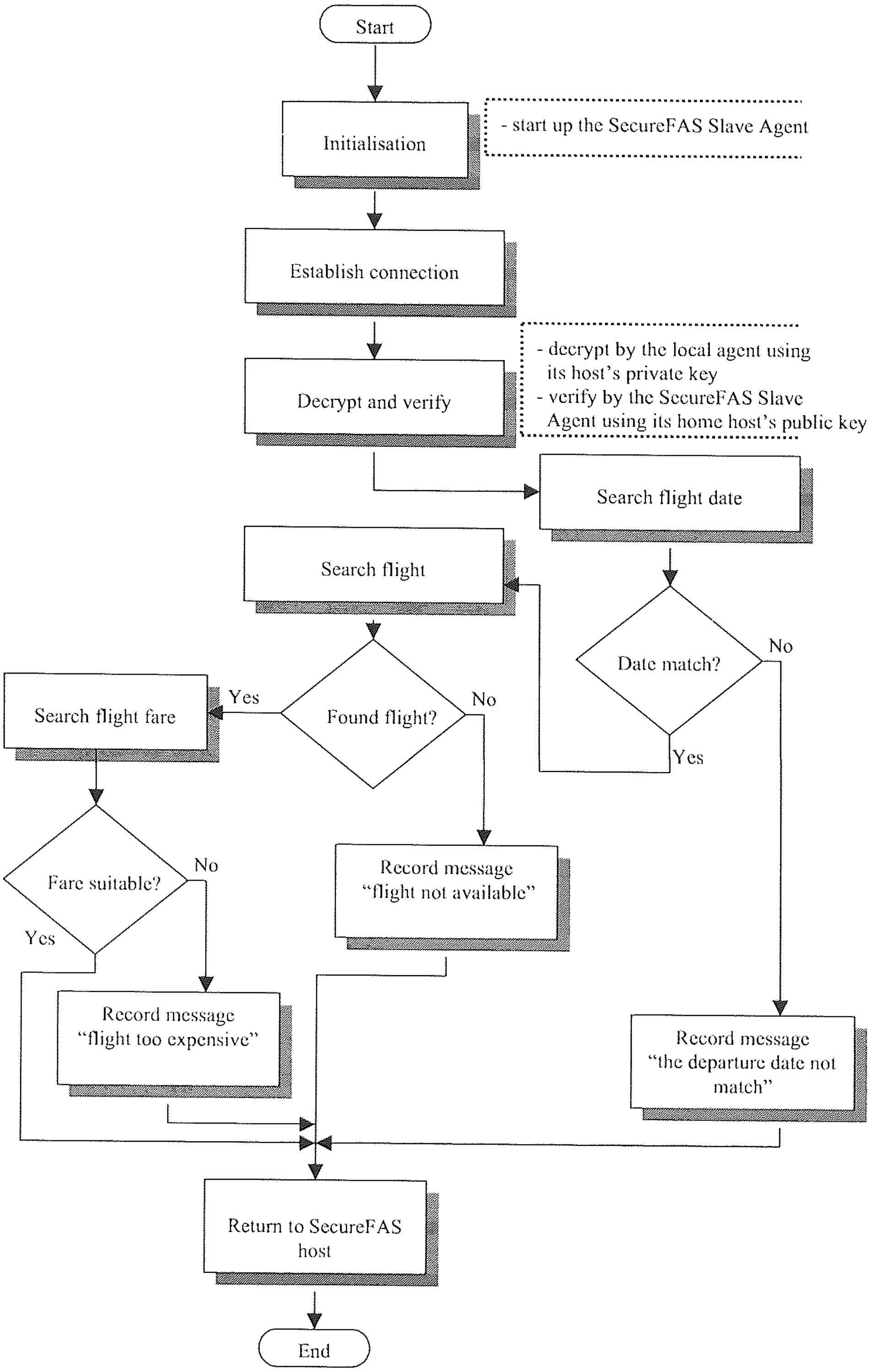

Start
Initialisation
- start up the SecureFAS Slave Agent
Establish connection
Decrypt and verify
- decrypt by the local agent using its host's private key
- verify by the SecureFAS Slave Agent using its home host's public key
Search flight date
Date match?
No
Yes
Search flight
Found flight?
Yes
No
Search flight fare
Fare suitable?
No
Yes
Record message "flight not available"
Record message "flight too expensive"
Record message "the departure date not match"
Return to SecureFAS host
End

Figure 5.2: The SecureFAS Slave Agent System

The recorded data in both containers allow the SecureFAS Evaluation Agent in the SecureFAS host to analyse the SecureFAS Slave Agent, in order to detect any malicious activities that have tried to tamper with the SecureFAS Slave Agent process during SecureFAS Slave Agent execution session inside the destination host execution environment.

Establish connection

As in the client-server concept, in order to enable the SecureFAS Slave Agent to communicate or execute any transaction with the destination host, the SecureFAS Slave Agent has to establish a connection with the destination host's local agent. The difference between these two concepts is in the way the connection is established. In the client-server, the connection is established over the network, while in the agent-based system, the connection is established inside the destination host execution environment, allowing the visiting agent to take advantage of exploiting resource near the data source and thus reducing network traffic (*Wang et al.*, 2002). In the SecureFAS Slave Agent system, once the SecureFAS Slave Agent's initialisation process has completed, the SecureFAS Slave Agent starts searching for the destination host's local agent. Once the SecureFAS Slave Agent found it, the SecureFAS Slave Agent will establish a connection with the local agent and will start executing its given tasks.

Decrypt and verify

In the SecureFAS prototype system, the RecordedReadOnly container that contains the user's specific purchase requirement data is encrypted by the SecureFAS Master Agent using the destination host's public key and digitally signed using the SecureFAS host's private key before being dispatched to the destination host. This is to protect the data of the RecordedReadOnly container from being accessed by unauthorised parties and also to protect its integrity.

In order to enable the SecureFAS Slave Agent to use the data inside the RecordedReadOnly container during its execution session inside the destination host execution environment, the data of the RecordedReadOnly container needs to be decrypted and verified by the local agent of the destination host. Since the data in the RecordedReadOnly container was encrypted by the SecureFAS Master Agent using the destination host's public key, the data of the RecordedReadOnly container can only be decrypted by the local agent of the destination host using the destination host's private key and verified using the SecureFAS host's public key. To decrypt the data of the RecordedReadOnly container, the SecureFAS Slave Agent needs to send the encrypted RecordedReadOnly container to the local agent of the destination host. Once the local agent of the destination host receives the RecordedReadOnly container, the local agent starts decrypting the data of the container using its host private key and verifying it using the SecureFAS host's public key, before sending the data back to the SecureFAS Slave Agent for further actions. Once the SecureFAS Slave Agent receives the version of unencrypted RecordedReadOnly container from the local agent, the SecureFAS Slave Agent has to verify the RecordedReadOnly container using its SecureFAS host public key to ensure that the data in the RecordedReadOnly container has not been tampered with by the destination host or unauthorised parties during the process of dispatching the SecureFAS Slave Agent from the SecureFAS host to the destination host and during the process of the RecordedReadOnly data decryption inside the destination host execution environment. Once verified, the SecureFAS Slave Agent continues to execute the next stage.

Search flight date

The SecureFAS Slave Agent executes the search flight date process in order to check the flight availability that suits the SecureFAS user requested date. If the local agent responds with the message "the departure date not match", the SecureFAS Slave Agent will stop it execution process and return to the SecureFAS host with the message "the departure date not match" recorded in the RecordedCollectOnly container. On the other hand, the SecureFAS Slave Agent continues to execute the next stage.

Search flight

The search flight process started by the SecureFAS Slave Agent by sending a flight request that contains the user flight information such as departure place, and destination place to the local agent of the destination host and wait for the local agent to respond. If the local agent responds with the message "flight not available", the SecureFAS Slave Agent will stop its execution process and return to its SecureFAS host with the message "flight not available" recorded in the RecordedCollectOnly container. Otherwise, if the local agent responds with the message "flight found", the local agent will supply the SecureFAS Slave Agent with the flight information such as the flight number, the flight departure and arrival date, and the number of seats available in each cabin. Once the flight information has been received from the local agent, the SecureFAS Slave Agent continues to execute the next phase.

Search flight fare

The SecureFAS Slave Agent starts this process by checking for the flight cabin that can accommodate the number of passengers requested by the user, starting from the first class cabin and sends a request to the local agent to enquire for the flight fare. The local agent will respond to the SecureFAS Slave Agent enquiry by providing the SecureFAS Slave Agent with the lowest flight fare available. The SecureFAS Slave Agent then compares it with the user maximum budget. If the fare is higher than the user maximum budget, the SecureFAS Slave Agent will downgrade the cabin class, for example, if the previous cabin class selected was the first class cabin, the SecureFAS Slave Agent will downgrade it to the business class cabin and so on. If the cabin class selected can accommodate the number of passengers requested by the user, the SecureFAS Slave Agent will send a new request to enquire for the flight fare. Otherwise, the SecureFAS Slave Agent will downgrade the cabin class again until it reaches the economy class cabin. If the flight fare is still higher than the user maximum budget, the SecureFAS Slave Agent will return to the SecureFAS host with the message "the flight is too expensive" recorded in the RecordedCollectOnly container. Otherwise, the

SecureFAS Slave Agent will accept the flight fare offered and forward it to the next phase.

When looking for the first, business and economic class cabin offer, if the SecureFAS Slave Agent stopped searching after it found the matching offer at the first class cabin offer, the remote host could assume that the SecureFAS Slave Agent can afford the most expensive flight offer. Therefore the SecureFAS Slave Agent will continue to search for business and economic offers to prevent a malicious host from inferring anything.

Return to SecureFAS host

Once the SecureFAS Slave Agent has completed its tasks at the destination host, the SecureFAS Slave Agent has to return to its SecureFAS host in order to deliver the result that was gathered at the destination host. However, before the SecureFAS Slave Agent, together with its data in the RecordedState mechanism can return, the SecureFAS Slave Agent has to ask the local agent to digitally sign its data in the Recorded State Mechanism containers. This is to guarantee the integrity of the data and to provide a proof for the SecureFAS host in order to overcome the problem of the destination host trying to deny offering any flight fare or flight information to the SecureFAS Slave Agent (repudiation attack). Once the local agent has digitally signed the SecureFAS Slave Agent's data, the SecureFAS Slave Agent together with its data then returns to the SecureFAS host for further actions.

5.4 The SecureFAS implementation

The SecureFAS prototype has been developed based on the design that was described in the earlier section. The prototype is implemented on an environment that consists of the home host (SecureFAS host), the certificate authority host and a few simulated airline hosts. Figure 5.3 shows the overall implementation of the SecureFAS prototype.

The home host is the host of the SecureFAS user. This host is used by the user to initiate the SecureFAS transactions in finding the best flight offer among the airline hosts in the Internet. The user initiates the SecureFAS transactions by starting up

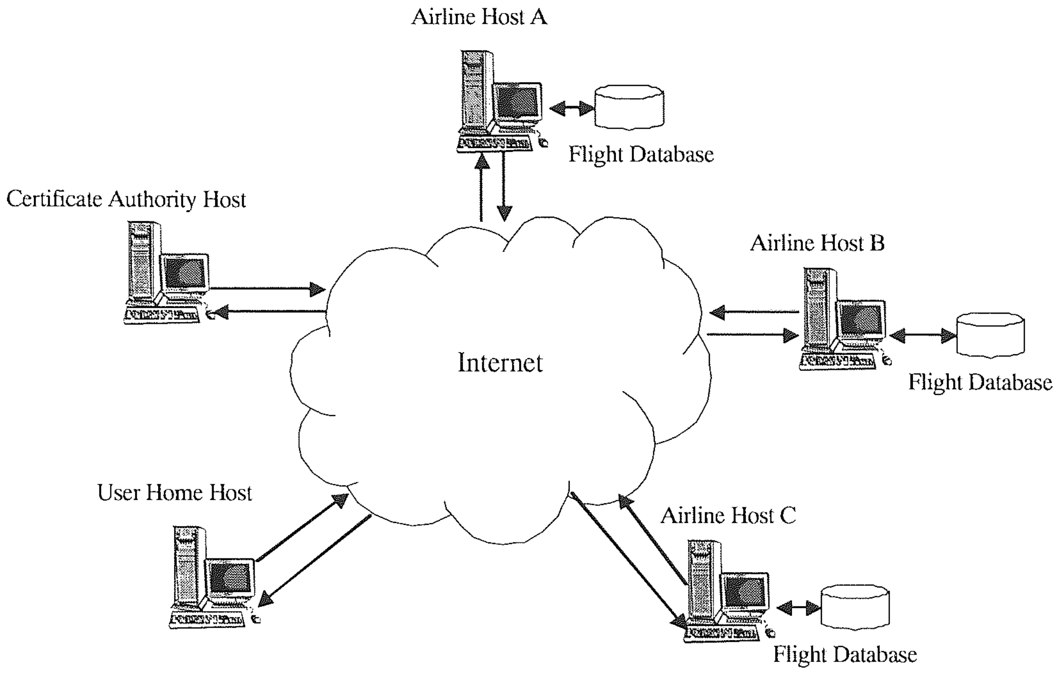


Figure 5.3: The SecureFAS implementation environment

the master agent to execute inside the home host execution environment to generate and dispatch slave agents to execute tasks, on behalf of the user, inside the airline host execution environment. The certificate authority host is the host responsible for maintaining the public keys of all the participating hosts. Each of the participating hosts has to register their own public keys with the certificate authority host before they are allowed to execute any transactions with other participating hosts. The airline host is the host that offers flight services, such as flight information, flight fare etc. These hosts are competing among each other, resulting in some of the services, such as the flight fare offered being different.

To carry out any transaction in the SecureFAS implementation environment, seven different types of agent as given in Figure 5.4 are used as follows:

The Master Agent(MA) is the agent that is started up by the user and only executes inside the home host execution environment. This agent is used as an interface between the user and the SecureFAS prototype and function to manage all the user transactions in order to find the best flight offer among the participating airline hosts in the Internet.

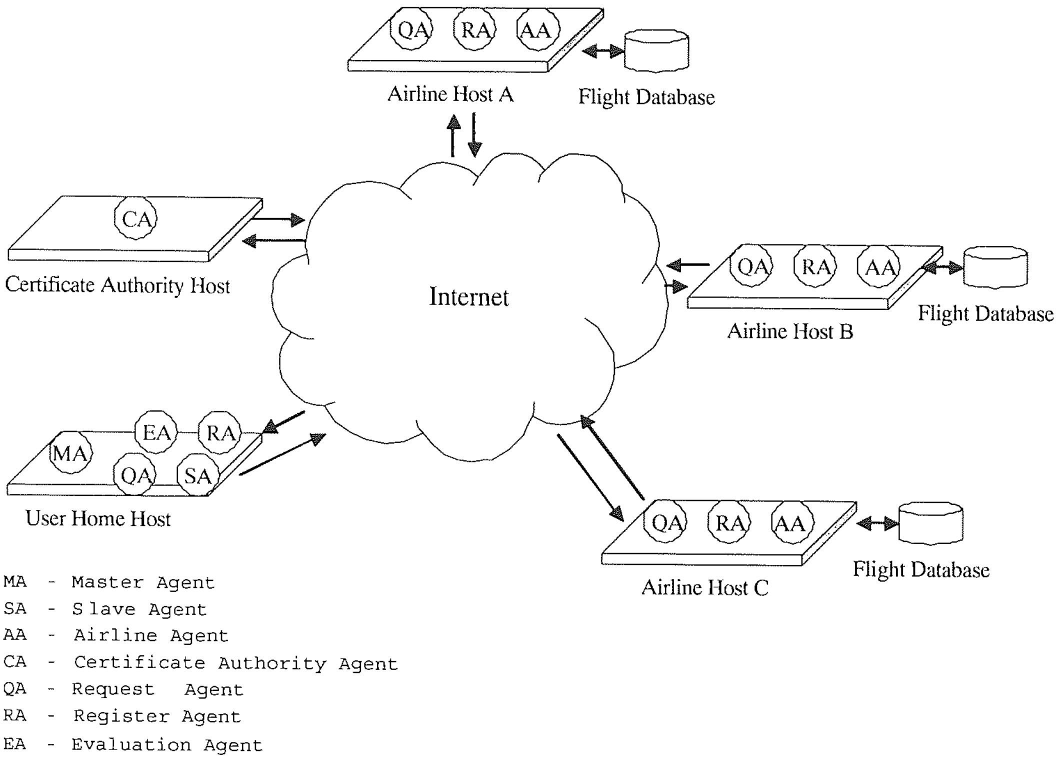


Figure 5.4: The conceptual view of the participating agents

The Slave Agent(SA) is started up by the master agent. This agent is used by the master agent to execute the user tasks on the participating airline hosts execution environment. In each transaction, the master agent will generate n slave agents to serve n participating airline hosts and will dispatch each of the agents to a particular airline host to execute the user tasks and will eventually return to it home host with the result.

The Airline Agent(AA) is the agent that manages all the airline host services. This agent acts as an interface between the airline host and other agent or party that wants to interact with the airline host. This agent is started up by the owner of the airline host and only executes inside the airline host execution environment, waiting for any request from the visiting agent or party.

The Certificate Authority Agent(CA) is an agent that manages the operation of the certificate authority host. This agent is used by other visiting agent or party to register or request for a particular host's public key from the certificate authority host database. This agent is started up by the authority that maintains the certificate authority host.

The Request Agent(QA) is the agent that is used for requesting public key from the certificate authority host. This type of agent is used by the master agent and the airline agent during the flight finding operation.

The Register Agent(RA) on the other hand is used for registering public key at the certificate authority host. Each of the participating hosts has to register their own public key at the certificate authority host to enable them to participate in the flight finding operation.

The Evaluation Agent(EA) is the agent that is responsible for processing and analysing the returning slave agent's data and state. This agent will be started up by the master agent upon receiving the first returning slave agent. This agent will remain resident inside the home host execution environment until all the slave agents have returned or the evaluation time that is set once the evaluation agent is started up by the master agent has expired, whichever comes first.

5.4.1 The SecureFAS application

The SecureFAS prototype application is designed to execute on the Unix and Windows platform, and use either a Unix workstation or a personal computer as an execution host to execute the participating agents.

The SecureFAS application can be divided into two main modules, the master agent module and the slave agent module. The master agent is the module that is responsible in coordinating the SecureFAS process from the initial process of dealing with the user requirement until the final process of evaluating the slave agent to choose the best offer and purchase the flight ticket. On the other hand, the slave agent is the module that is created and dispatched by the master agent module to execute tasks on the remote host execution environment.

The agent execution environment

The agent execution environment is a place where agents execute and operate. It provides a uniform set of services for the executing agent to perform it tasks, where it can be regarded as the operating system for the agent (*Lange and Oshima*, 1998). This research used the Tahiti aglet server (*Lange and Oshima*, 1998) as an agent execution environment for controlling the agents.

The Tahiti aglet server is a Java application that allows the user to manage, receive and send the agents to other computers. It can be started by executing the " agletsd -f my_aglets.props " command, which will produce a window of the Tahiti Aglet Server (see section A.5).

In the SecureFAS implementation environment, each of the participating computer host needs to start the Tahiti Aglet Server. This is to enable the participating computer host to manage, receive and send agents. Once the participating computer host have starts up the Tahiti Aglet Server, the user of the computer host can create and execute their agents.

To execute the SecureFAS prototype, the user has to execute the Master Agent module on the Tahiti Aglet Server running on the user's computer host (home host).

Once the Master Agent of the SecureFAS prototype is executed, the master agent will execute the initialisation function, cryptographic key registration function, filtering bad addresses function, cryptographic key retrieval function, obfuscating data function, storing read-only data function, generating and dispatching slave function, evaluation function, and display result function as mentioned in section 5.3.1.

The Slave Agent module will be created and executed by the Master Agent during the execution of the generating and dispatching slave agent function. The executing Slave Agent on the Tahiti Aglet Server on the home host will be dispatched by the Master Agent to execute the user's tasks on the remote host. The Slave Agent will returned to the Master Agent's computer host for further action once all it given tasks have been executed on the remote host execution environment.

The Aglets Software Development Kit (ASDK) package

A set of Java classes and interfaces for creating a SecureFAS prototype system is provided by the ASDK package. This package contains methods for initialising an agent and message handling, as well as dispatching, deactivating/activating, retracting, cloning, and disposing an agent (*Lange and Oshima*, 1998). For examples, the dispatch method using "dispatch(new URL("atp://name.host.com/context"));" causes an agent to move from the local host to the destination host, the deactivate method using "deactivate(300 * SECONDS);" allows an agent to be stored in secondary storage, and the clone method (addCloneListener(CloneListener listener)) spawns a new instance of the agent, which has the state of the original agent. A complete ASDK package can be referred in (*Lange and Oshima*, 1998)

5.5 Concluding remarks

This chapter presented the development of the SecureFAS prototype system. This includes the implementation of the enabling methods: the master-slave agent architecture, the distributed migration pattern, and the cryptographic protocol, and the implementation of the security mechanisms: the Random Sequence 3-level obfuscation algorithm and the Recorded State Mechanism. The implementations are to show the acceptability and the functionality of the security mechanisms on an agent-based system application. Finally, the chapter presented a discussion on the SecureFAS prototype implementation that includes explanation on the steps to execute the SecureFAS application.

Part IV

Experimental Analysis

Chapter 6

Experimental Results

6.1 Introduction

This chapter presents two sets of experimental results. First are the results of the experiments done on:

- the itinerary migration pattern, and
- the distributed migration pattern.

This is to measure the migration and migration+computation time overheads. These results will be analysed and the faster migration pattern will be chosen.

Second are the results of the experiments done to measure the security time overhead. Experiments are done on:

- plain agents - agents without security mechanisms,
- agents with conventional cryptographic security mechanisms (e.g. encryption and digital signature mechanisms),
- agents with the random sequence 3-level obfuscation algorithm, and
- agents with security and recorded state mechanisms.

This experiment is to measure the security overhead for the agents with security mechanism compared to the agent without the security mechanism.

In this thesis, the experimental results, for both sets of experimental test, are measured using the "System.currentTimeMillis()" method in the Java language. This method produces a specific instant in time with millisecond precision (*Sun Microsystems*, 2004). The experimental results are gathered by taking the difference between the start time (time taken starting from sending the agent to the remote host) and the end time (time taken when receiving the agent from the remote host).

6.2 Comparison Experiment Between The Itinerary and Distributed Migration Pattern

This section describes the comparison experiment between the itinerary and the distributed migration pattern in order to examine their performances in terms of the migration and migration+computation speed. This comparison experiment is used to investigate the hypothesis that the performance of an agent-based application that uses a distributed migration pattern with master-slave agent architecture, is faster than an agent-based application with the itinerary migration pattern using single agent architecture, even though the first agent-based application is required to generate and dispatch more than one agent to execute a transaction, which does increase network overhead at the originating host.

6.2.1 The experimental migration pattern

In this research, the itinerary and distributed migration pattern have been selected as the experimental migration pattern for this comparison experiment. This is because both migration patterns are widely used in agent-based applications (*Hohl*, 2000; *Kotzanikolaou et al.*, 2000; *Vigna*, 1998)

The itinerary migration pattern is a migration pattern that allows a single agent to travel from one host to another in sequence (see figure 6.1). The implementation of this pattern needs the agent to maintain a list of destinations and to always know where to go next (*Lange and Oshima*, 1998).

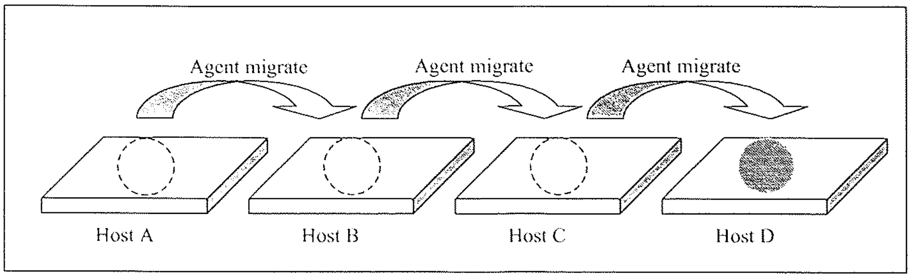


Figure 6.1: Itinerary Migration Pattern

The distributed migration pattern, on the other hand is a pattern that allows more than one agent to travel to different remote hosts (see figure 6.2). This pattern allows agents to be dispatched in parallel to different remote hosts, with one agent is dispatched to each remote host (*Lange and Oshima*, 1998).

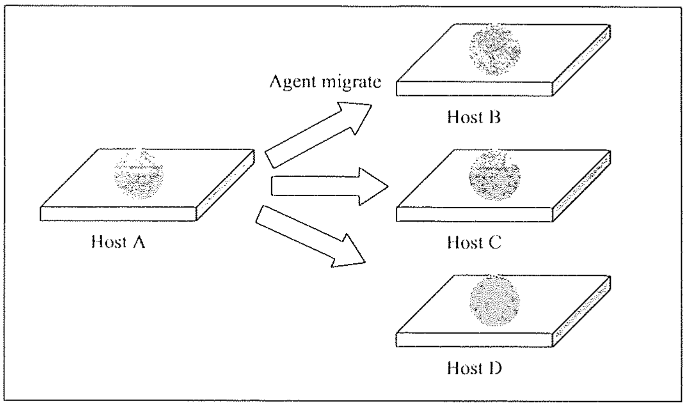


Figure 6.2: Distributed Migration Pattern

6.2.2 Experiment configuration and scenario

The comparison experiment on the itinerary and distributed migration pattern is executed on nine 400 MHz Sun Ultra Sparc 5 workstations with 128 MB of main memory. Each of the workstations is running the Solaris 8 operating system and is connected to the others using 100 Mbit/s UTP [1] cable. All of the workstations involved in this experiment were situated in the same room.

[1] Unshielded Twisted Pair Category 5e

To start up the comparison experiment, all of the workstations are required to execute the Tahiti Aglet Server (*Lange and Oshima*, 1998). This is to enable the participant workstations to have the capability to manage, dispatch and receive agents to or from other workstations. Figure 6.3 shows the experiment configuration. In this configuration, one workstation will be chosen among the nine workstations to be the home host for the agent, and only this host has the permission to manage and dispatch the agent. The rest of the workstations are assumed to be the remote host and only have the capability to receive and dispatch the agent back to its home host.

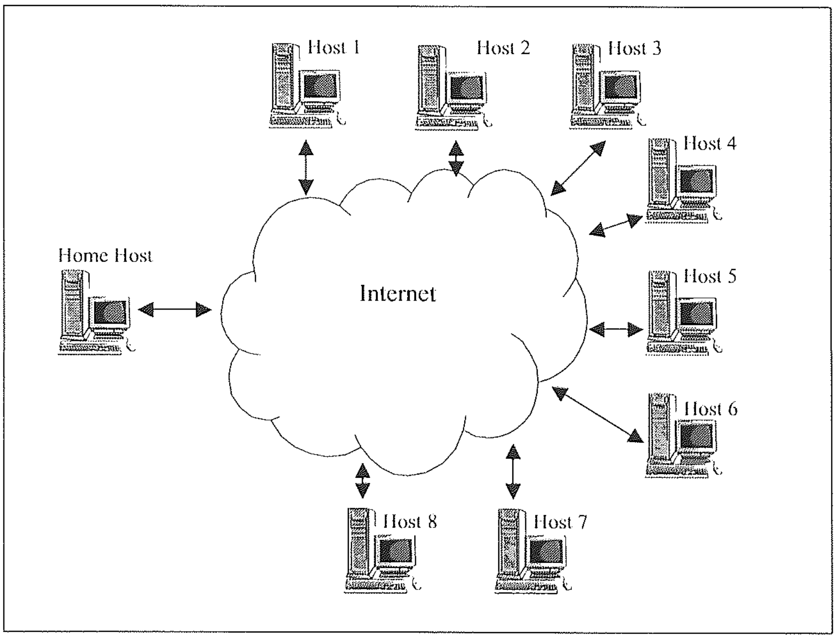


Figure 6.3: Comparison Experiment Configuration

To examine the migration overhead for both itinerary and distributed migration patterns, we measure the length of time starting from when the first agent is sent to the remote host and ending when the last agent returns. There are no processing activities required at the remote host, the agent simply return. On the other hand the migration+computation overhead for the itinerary and distributed migration patterns is examined by measuring the migration+computation time starting from sending the agent to the remote host, waiting for the agent to execute the function of numerical calculation (generating prime numbers) inside the remote host and ending when the last agent has returned from the remote host.

In each test, the agent will carry different amounts of data, varying from 100KB to 800KB with increments of 100KB. For example, in the first test the agent carry 100KB data, then in the second test the agent carry 200KB data, and so on.

Both experiments are repeated for 20 times and the result for each run is gathered in milliseconds. The mean result of all 20 runs is then converted into seconds by dividing by 1000. The result is then rounded and presented in two decimal places. From the author's observation, all the 20 runs in this experiment give very similar results and for this reason, 20 runs of the experiment are considered sufficient.

6.2.3 Experimental Results

There are two comparison experiments conducted on the itinerary and distributed migration pattern: the migration and the migration+computation experiment. *All time results in section 7.3 are given in seconds.*

Migration experiment

To evaluate the migration overhead of both migration patterns, four different experiments are done using one remote host, two remote hosts, three remote hosts and eight remote hosts respectively.

Based on the observation of the results gained through the experiments, it can be seen that the standard error and the standard deviation of the migration overhead gets larger as the number of bytes and remote host gets bigger. The assumption made is that, the greater the number of bytes carried by the agent during the migration process, the larger the network bandwidth consumed, therefore the network will be congested and the time taken will less predictable.

From the results given in Table 6.1 and illustrated in Figure 6.4, it can be seen that the mean of the migration overhead is almost the same for both migration patterns where the difference is just around 0.06 % to 0.85 % except for the 200KB and 800KB

Number of Bytes	Mean		Standard Error		Standard Deviation	
	Itinerary	Distributed	Itinerary	Distributed	Itinerary	Distributed
100KB	9.44	9.36	0.02	0.01	0.08	0.06
200KB	17.09	17.40	0.06	0.02	0.25	0.10
300KB	25.35	25.38	0.02	0.01	0.08	0.06
400KB	32.45	32.47	0.02	0.02	0.10	0.08
500KB	41.43	41.51	0.04	0.03	0.16	0.12
600KB	48.43	48.36	0.10	0.04	0.43	0.20
700KB	55.18	55.09	0.06	0.04	0.27	0.20
800KB	66.40	67.20	0.09	0.08	0.41	0.34

Table 6.1: Summary Statistics of Migration Overhead for 1 Remote Host

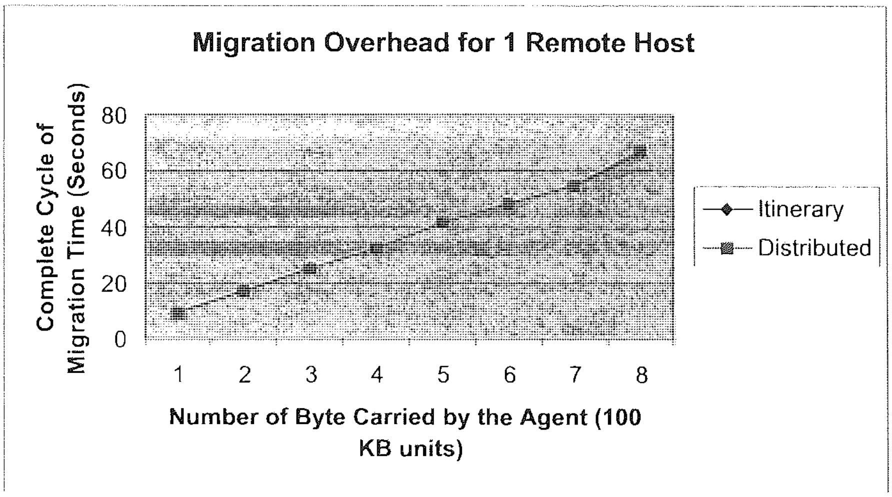


Figure 6.4: Migration Overhead for 1 Remote Host

data size where the mean of the overhead for the distributed migration pattern is higher by 1.81 % and 1.2 % respectively.

From Table 6.2 and Figure 6.5, considerable differences in the mean of the migration overhead can be seen starting from the 500KB. The highest difference is for 600KB where the overhead for the distributed migration pattern is 4.36 % lower than the itinerary migration pattern.

Number of	Mean		Standard Error		Standard Deviation	
Bytes	Itinerary	Distributed	Itinerary	Distributed	Itinerary	Distributed
100KB	17.53	17.17	0.01	0.03	0.07	0.14
200KB	32.84	33.88	0.06	0.05	0.26	0.21
300KB	48.20	46.89	0.07	0.05	0.31	0.23
400KB	62.03	61.20	0.08	0.03	0.37	0.15
500KB	82.68	79.32	0.31	0.19	1.37	0.85
600KB	95.69	91.52	0.20	0.29	0.91	1.29
700KB	113.01	108.43	0.33	0.31	1.47	1.40
800KB	132.76	129.80	1.42	0.35	6.33	1.55

Table 6.2: Summary Statistics of Migration Overhead for 2 Remote Host

Number of	Mean		Standard Error		Standard Deviation	
Bytes	Itinerary	Distributed	Itinerary	Distributed	Itinerary	Distributed
100KB	25.62	23.13	0.03	0.03	0.15	0.12
200KB	47.59	45.51	0.09	0.08	0.40	0.38
300KB	70.18	65.88	0.12	0.28	0.56	1.25
400KB	90.24	86.08	0.19	0.28	0.86	1.26
500KB	121.90	105.41	0.31	0.43	1.41	1.92
600KB	141.48	126.07	0.54	0.65	2.41	2.91
700KB	165.77	154.72	0.58	3.40	2.58	15.20
800KB	194.54	181.43	0.81	0.91	3.61	4.09

Table 6.3: Summary Statistics of Migration Overhead for 3 Remote Host

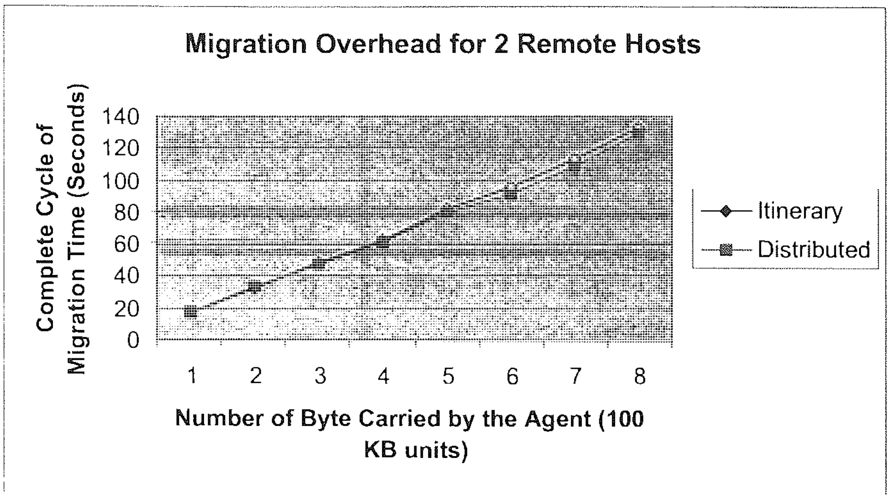


Figure 6.5: Migration Overhead for 2 Remote Hosts

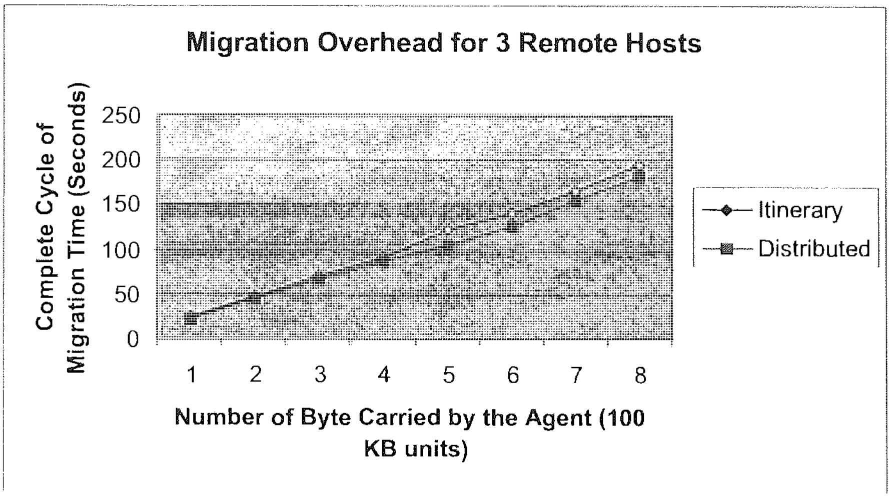


Figure 6.6: Migration Overhead for 3 Remote Hosts

For three remote hosts, the results in Table 6.3 and Figure 6.6 show that the mean of the migration overhead for the distributed migration pattern is lower even from the beginning but the highest difference in the mean of the overhead is for 500KB where the overhead for the distributed migration pattern is 13.53 % lower than the itinerary migration pattern and the lowest difference is for 200KB where the overhead for distributed migration pattern is 4.37 % lower than the itinerary migration pattern.

As for eight remote hosts (see Table 6.4 and Figure 6.7), the difference in the mean

Number of Bytes	Mean		Standard Error		Standard Deviation	
	Itinerary	Distributed	Itinerary	Distributed	Itinerary	Distributed
100KB	71.58	65.12	0.24	0.24	1.06	1.07
200KB	134.93	126.18	0.54	0.43	2.41	1.93
300KB	194.60	181.70	0.72	1.43	3.22	6.38
400KB	265.89	243.36	2.12	4.08	9.49	18.26
500KB	341.07	292.73	1.36	4.59	6.07	20.53
600KB	404.07	363.25	1.65	8.32	7.39	37.22
700KB	467.62	439.44	3.11	5.64	13.89	25.23
800KB	526.85	497.47	3.00	8.60	13.44	38.48

Table 6.4: Summary Statistics of Migration Overhead for 8 Remote Host

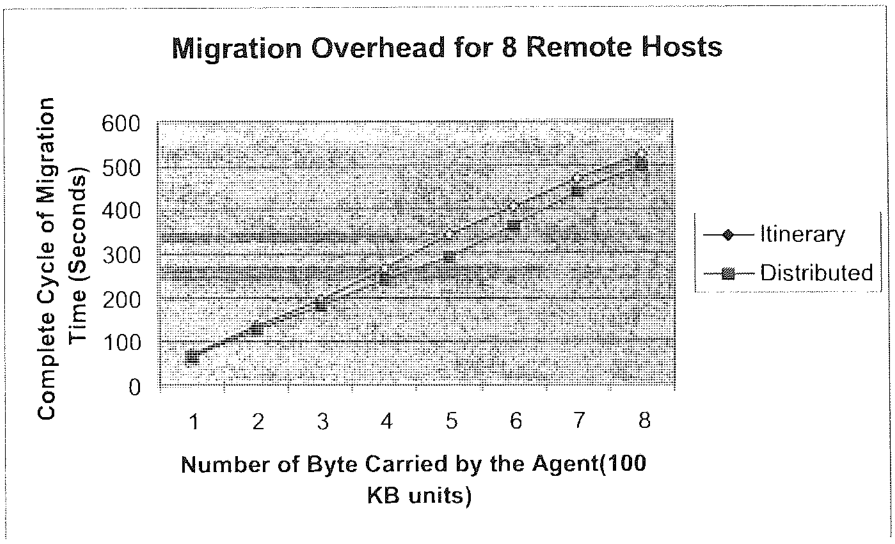


Figure 6.7: Migration Overhead for 8 Remote Hosts

of the migration overhead for both the migration pattern is more obvious. Even for 100KB, it can be seen that the mean of the overhead for the distributed migration pattern is much lower that is 9.02 % lower than the mean of the overhead for the itinerary migration pattern.

6.2.4 Summary of experimental results

From the graphs given in Figure 6.4 to 6.6, it can be seen that there is not much difference in the migration overhead for the distributed and itinerary migration pattern. For a small number of remote hosts, both migration patterns perform equally where the highest difference between the two overhead is just 4.36 %, but as the number of remote hosts increases as given in Figure 6.6 to 6.7, it can be seen that the distributed migration pattern gives lower migration overhead of up to 14.17 %. For both migration patterns, the time taken increases approximately linearly as the number of bytes increases. In conclusion, the distributed migration pattern is better than the itinerary migration pattern in term of reduced migration overhead.

Migration+Computation experiment

To evaluate the migration+computation overhead of both the migration patterns, four different experiments are done starting with one remote host, two remote hosts, three remote hosts and eight remote hosts.

Based on the observation on the results gained through the experiments done, it can be seen that the standard error and the standard deviation of the migration+computation overhead are similar regardless of the number of prime generated or the number of remote host. The assumption made is that, the smaller the number of bytes carried by the agent, the less network bandwidth consumed, therefore the probability that network congestion occurs is low and the time taken will be consistent.

From the results given in Table 6.5 and illustrated in Figure 6.8, it can be seen that the mean of the migration+computation overhead is almost the same for both migration patterns since the experiment is done on only one remote host.

From Table 6.6 and Figure 6.9, the difference in the mean of the migration+computation overhead between the two migration patterns can be seen clearly.

For three remote hosts and eight remote hosts, the results in Table 6.7, Table 6.8, Figure 6.10 and Figure 6.11 show that the difference in mean of the migration+computation overhead for both the migration patterns is getting bigger, where

Number of Prime	Mean		Standard Error		Standard Deviation	
	Itinerary	Distributed	Itinerary	Distributed	Itinerary	Distributed
10000	2.39	2.31	0.01	0.002	0.03	0.01
20000	3.67	3.58	0.01	0.01	0.04	0.02
30000	5.4	5.38	0.01	0.01	0.04	0.07
40000	7.43	7.32	0.01	0.01	0.05	0.06
50000	9.41	9.5	0.004	0.01	0.02	0.03
60000	12.17	12.11	0.01	0.01	0.04	0.03
70000	14.83	14.82	0.01	0.02	0.04	0.07
80000	17.84	17.83	0.01	0.01	0.04	0.05
90000	21.11	21	0.03	0.01	0.11	0.04
100000	24.42	24.28	0.03	0.02	0.15	0.07

Table 6.5: Summary Statistics of Migration+Computation Overhead for 1 Remote Host

Number of Prime	Mean		Standard Error		Standard Deviation	
	Itinerary	Distributed	Itinerary	Distributed	Itinerary	Distributed
10000	5.74	3.59	0.01	0.01	0.03	0.03
20000	10.13	5	0.05	0.01	0.22	0.05
30000	15.66	6.41	0.01	0.01	0.04	0.06
40000	22.79	8.48	0.03	0.02	0.14	0.07
50000	29.78	10.75	0.05	0.01	0.21	0.04
60000	37.92	13.36	0.14	0.01	0.63	0.06
70000	46.52	16.13	0.04	0.01	0.19	0.05
80000	55.73	19.23	0.04	0.01	0.17	0.06
90000	65.28	22.19	0.01	0.01	0.07	0.05
100000	75.77	25.41	0.06	0.02	0.26	0.07

Table 6.6: Summary Statistics of Migration+Computation Overhead for 2 Remote Host

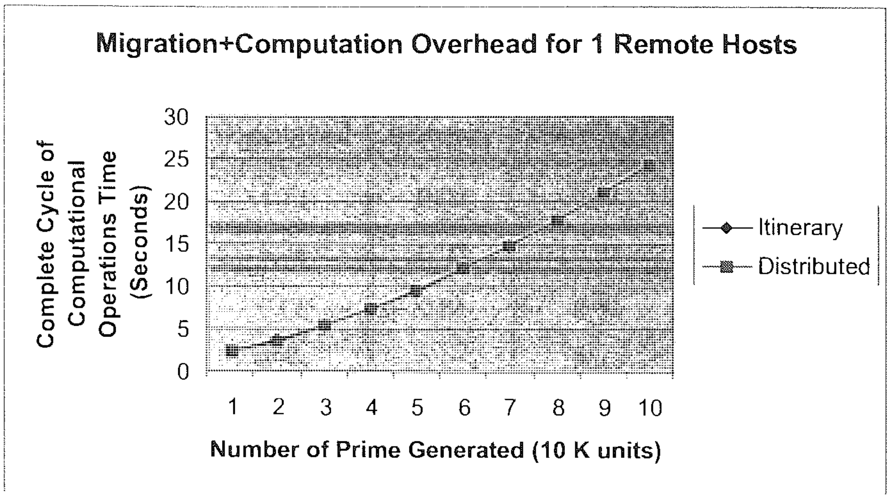


Figure 6.8: Migration+Computation Overhead for 1 Remote Host

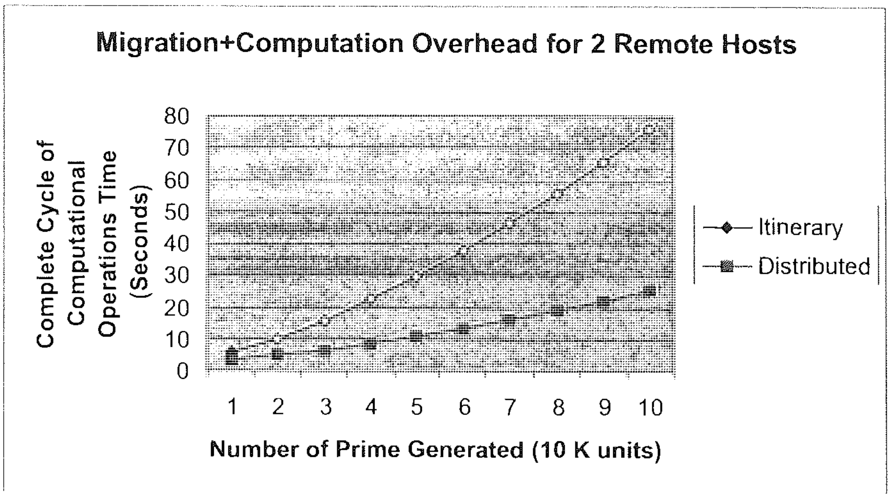


Figure 6.9: Migration+Computation Overhead for 2 Remote Hosts

the migration+computation overhead for the distributed migration pattern is up to 95.65 % lower than the itinerary migration pattern.

Summary of experimental results

It can be seen from the graphs 6.8 to 6.11 that the migration+computation overhead for the distributed migration pattern is much lower than the migration+computation overhead for the itinerary migration pattern (up to 95.65 %) except for one remote host

Number of Prime	Mean		Standard Error		Standard Deviation	
	Itinerary	Distributed	Itinerary	Distributed	Itinerary	Distributed
10000	10.59	4.73	0.02	0.01	0.09	0.03
20000	19.69	6.36	0.03	0.01	0.12	0.06
30000	30.91	7.65	0.04	0.01	0.18	0.04
40000	44.37	9.68	0.21	0.01	0.93	0.03
50000	57.92	11.79	0.07	0.003	0.32	0.01
60000	73.53	14.59	0.04	0.01	0.20	0.03
70000	90.59	17.46	0.03	0.02	0.15	0.08
80000	107.92	20.06	0.07	0.02	0.32	0.09
90000	127.57	23.38	0.20	0.02	0.88	0.09
100000	147.15	26.75	0.20	0.03	0.88	0.13

Table 6.7: Summary Statistics of Migration+Computation Overhead for 3 Remote Host

Number of Prime	Mean		Standard Error		Standard Deviation	
	Itinerary	Distributed	Itinerary	Distributed	Itinerary	Distributed
10000	48.96	11.45	0.14	0.02	0.63	0.07
20000	99.92	12.65	0.19	0.01	0.83	0.06
30000	160.32	14.46	0.33	0.02	1.47	0.07
40000	232.67	16.39	0.22	0.02	0.99	0.1
50000	307.9	18.38	0.18	0.02	0.82	0.09
60000	390.06	20.87	0.53	0.05	2.37	0.22
70000	476.83	23.58	0.46	0.02	2.06	0.11
80000	569.98	26.46	0.68	0.02	3.05	0.08
90000	668.44	29.59	0.38	0.01	1.7	0.03
100000	769	33.46	1.1	0.02	4.9	0.07

Table 6.8: Summary Statistics of Migration+Computation Overhead for 8 Remote Host

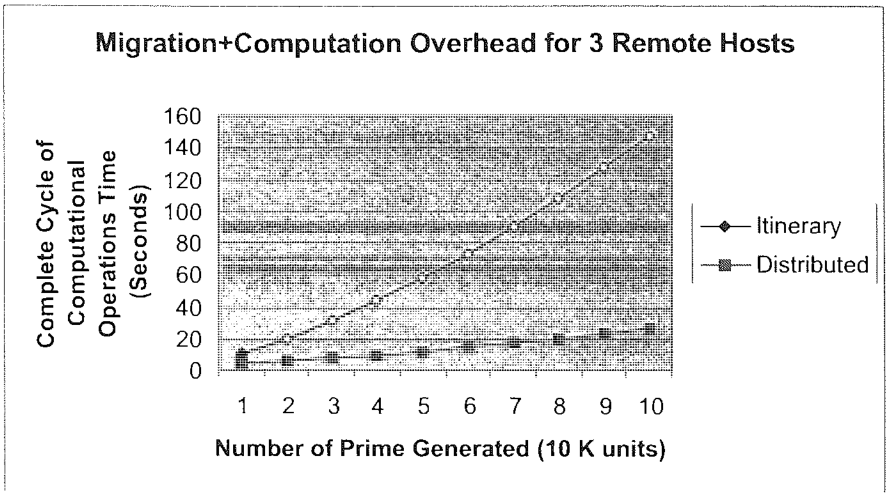


Figure 6.10: Migration+Computation Overhead for 3 Remote Hosts

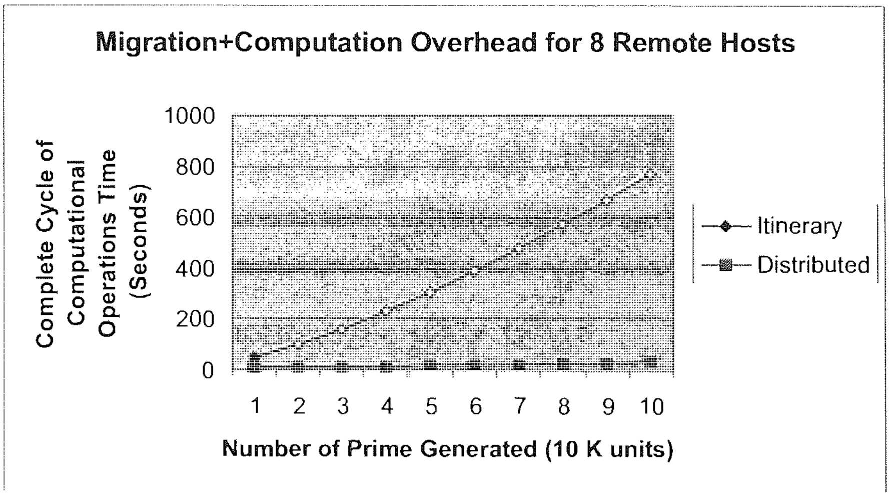


Figure 6.11: Migration+Computation Overhead for 8 Remote Hosts

where the migration+computation overhead for both migration patterns is almost the same (the highest difference is only 3.35 %).

These results are as expected because in the itinerary migration pattern, only one agent migrates in sequence for all the remote hosts, whereas, in distributed migration pattern, many agents depending on the number of remote host are used in parallel at one time. However, for the distributed migration pattern, the home host need to collate the information that was brought back by the agents and therefore, another small or negligible overhead will be incurred during this process. In conclusion, the distributed migration pattern is better than the itinerary migration pattern in terms of reduced migration+computation overhead.

6.3 The Overhead of Implementing Security Protection

This section describes the experiment conducted in order to measure the overhead of implementing security protection mechanism into an agent-based application. This experiment is used to investigate the hypothesis that the performance of an agent-based application equipped with security mechanisms is slower than the agent-based application without security mechanisms. Taking into consideration both migration and migration+computation overheads, the distributed migration pattern has been confirmed as the most appropriate migration pattern and therefore, for simplicity, the security overhead will only be tested on this migration pattern.

6.3.1 The experimental security protection mechanisms

The Random Sequence 3-level obfuscation algorithm and the Recorded State Mechanism are two security protection mechanisms that are used in the experimental test to measure the overhead of implementing security protection in an agent-based application.

The Random Sequence 3-level obfuscation algorithm is a security protection mechanism that is used to protect the confidentiality of an agent from being spied on by the malicious host. The mechanism uses multiple polynomial functions for obfuscating the agent's data to an obfuscated value that is meaningless to the malicious host.

The Recorded State Mechanism, on the other hand is a security protection mechanism that is used to protect the integrity of an agent. This mechanism is used to detect any modification attacks from the malicious host by examining the state information of an agent that has been recorded during the agent execution process.

6.3.2 Experiment configuration and scenario

The experiments to measure the overhead of implementing security protection in an agent-based application are conducted using the same experiment configuration as in section 6.2.2. However, due to the renovation of the computer laboratory, there were fewer workstations available, so only 6 workstations are used in this experiment, where 1 workstation will be chosen to be the home host and other 5 workstations are assumed to be the remote hosts.

To examine the security overhead for implementing both the Random Sequence 3-level obfuscation algorithm and the Recorded State Mechanism in an agent-based application, security overhead times are taken starting from sending of the agents to the remote hosts, executing the security algorithm or mechanism at the remote hosts and ending by receiving the agents back from the remote hosts. However, the Recorded State Mechanism required extra processing overhead at the originating host to simulate the evaluation process on the returning agent.

Both experiments are repeated 20 times and the result for each run is gathered in milliseconds. From the author's observation, all the 20 runs in this experiment give very similar results and for this reason, 20 runs of the experiment are considered sufficient. The mean result of all 20 runs is then converted into seconds by dividing by 1000. The result is then rounded and presented in two decimal places.

6.3.3 Experiment Results

The Random Sequence 3-level obfuscation algorithm

To evaluate the security overhead for implementing the Random Sequence 3-level obfuscation algorithm, four different experiments are conducted starting with one remote host, two remote hosts, three remote hosts and five remote hosts on two different pairs of agents that are plain agent and agent with the Random Sequence 3-level obfuscation algorithm (RS3) or plain agent and agent with the Random Sequence 3-level obfuscation algorithm with noise code (RS3N).

Number of	Mean		
Remote Hosts	Plain	RS3	RS3N
1	1.49	1.5	1.69
2	2.45	2.52	2.67
3	3.36	3.56	3.63
5	5.42	5.43	5.66

Table 6.9: Summary Statistics of The Random Sequence 3-Level Obfuscation Algorithm (1 Cycle, 1 Obfuscation Value Experiment(without noise code) and 1000 Obfuscation Value Experiment(with noise code))

The results of the security overhead of the Random Sequence 3-level obfuscation algorithm without noise (RS3) and the Random Sequence 3-level obfuscation algorithm with noise are compared to the plain agent as shown in Table 6.9. From these results, it can be seen that there is not much difference in the security overhead for the RS3 without noise and RS3 with noise for all numbers of remote hosts. The highest difference is for one remote host, where the security overhead for RS3 with noise is just 12.67 % higher than the security overhead of RS3 without noise. Therefore, it is assumed that RS3 with noise will add a negligible overhead. More detail of the results now follows:

Firstly, consider plain and RS3, then plain and RS3N experimental results.

Number of	Mean		Standard Error		Standard Deviation	
Remote Hosts	Plain	RS3	Plain	RS3	Plain	RS3
1	1.49	1.5	0.001	0.001	0.003	0.004
2	2.45	2.52	0.003	0.005	0.011	0.022
3	3.36	3.56	0.003	0.004	0.011	0.02
5	5.42	5.43	0.007	0.006	0.031	0.025

Table 6.10: Summary Statistics of The Random Sequence 3-Level Obfuscation Algorithm Overhead (1 Cycle and 1 Obfuscation Value Experiment(without noise code))

From the results given in Tables 6.10 and 6.11 and illustrated in Figure 6.12 and

Number of	Mean		Standard Error		Standard Deviation	
Remote Hosts	Plain	RS3N	Plain	RS3N	Plain	RS3N
1	1.49	1.54	0.001	0.001	0.003	0.003
2	2.45	2.52	0.002	0.007	0.011	0.031
3	3.36	3.51	0.003	0.005	0.011	0.022
5	5.42	5.44	0.007	0.007	0.031	0.031

Table 6.11: Summary Statistics of The Random Sequence 3-Level Obfuscation Algorithm (1 Cycle and 100 Obfuscation Value Experiment(with noise code))

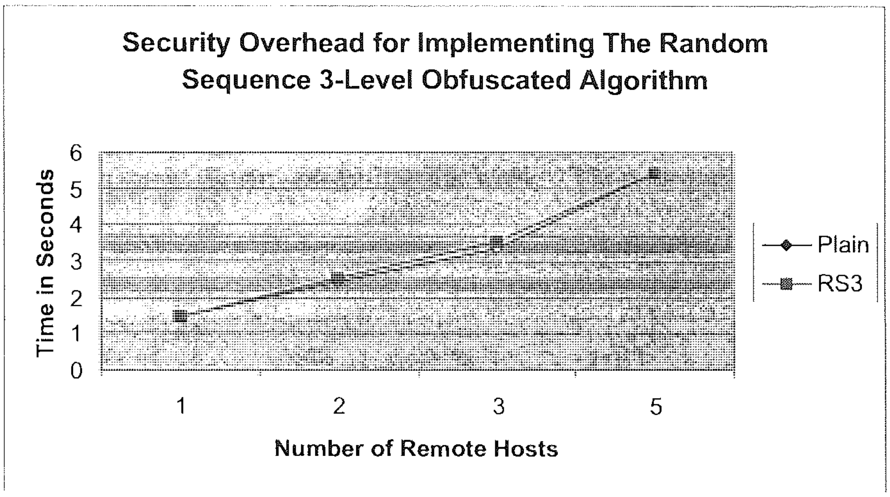


Figure 6.12: Security Overhead of The Random Sequence 3-Level Obfuscation Algorithm (1 Cycle and 1 Obfuscation Value Experiment(without noise code))

6.13 respectively, it can be seen that the mean of the security overhead for a plain agent is almost the same as the security overhead for RS3 and RS3N, where when comparing with RS3, the highest difference is just 5.95 % and 4.46 % when comparing with RS3N. However, from Table 6.12 and illustrated in Figure 6.14, it can be seen that as the number of noise codes is increased from 0 to 99 and to 999, the difference becomes larger, where the highest difference is 13.42 %, but it is still considered negligible.

Surprisingly, the result for RS3 using three remote host, which is shown in Table 6.10 is higher than the result of the RS3N using three remote host (in Table 6.11). Supposedly, the RS3N should produce higher result than just the RS3. One possibility

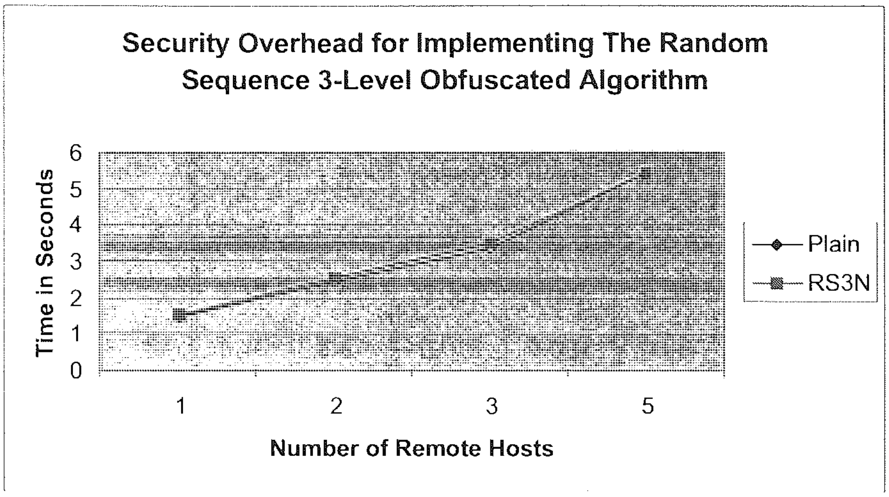


Figure 6.13: Security Overhead of The Random Sequence 3-Level Obfuscation Algorithm (1 Cycle and 100 Obfuscation Value Experiment(with noise code))

is that, the experiment is run on public network. Therefore, the results produce is unpredictable.

Number of	Mean		Standard Error		Standard Deviation	
Remote Hosts	Plain	RS3N	Plain	RS3N	Plain	RS3N
1	1.49	1.69	0.001	0.004	0.003	0.016
2	2.45	2.67	0.003	0.006	0.011	0.028
3	3.36	3.63	0.003	0.003	0.011	0.015
5	5.42	5.66	0.007	0.006	0.031	0.028

Table 6.12: Summary Statistics of The Random Sequence 3-Level Obfuscation Algorithm (1 Cycle and 1000 Obfuscation Value Experiment(with noise code))

From Table 6.13 and Figure 6.15, the security overhead for both the agents is almost the same as the security overhead given in Tables 6.10 to 6.12, even though now the number of cycles has been increased to 100.

Based on the observation on the results gained through the experiments done, it can be seen that the standard error and the standard deviation of the security overhead for both pairs of agents (plain and RS3) and (plain and RS3N) are similar since the

Number of Remote Hosts	Mean		Standard Error		Standard Deviation	
	Plain	RS3N	Plain	RS3N	Plain	RS3N
1	1.48	1.65	0.0003	0.002	0.001	0.008
2	2.46	2.65	0.002	0.003	0.01	0.014
3	3.45	3.67	0.007	0.005	0.03	0.021
5	5.46	5.67	0.013	0.004	0.06	0.02

Table 6.13: Summary Statistics of The Random Sequence 3-Level Obfuscation Algorithm (100 Cycle and 1000 Obfuscation Value Experiment(with noise code))

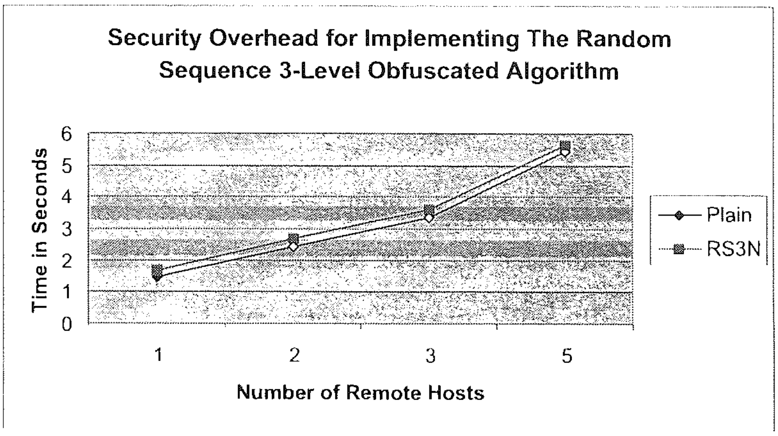


Figure 6.14: Security Overhead of The Random Sequence 3-Level Obfuscation Algorithm (1 Cycle and 1000 Obfuscation Value Experiment(with noise code))

agent with the Random Sequence 3-level obfuscation algorithm (RS3) and the agent with the Random Sequence 3-level obfuscation algorithm with noise (RS3N) only needs to execute a simple task (obfuscating method).

Summary of experimental results

It can be seen from the results shown in Tables 6.10 to 6.13 and illustrated in Figures 6.12 to 6.15 that the implementation of the Random Sequence 3-level obfuscation algorithm and the Random Sequence 3-level obfuscation algorithm with noise does increase the overhead by up to 13.42 % compared to the plain agent. The noise code adds little to the overhead.

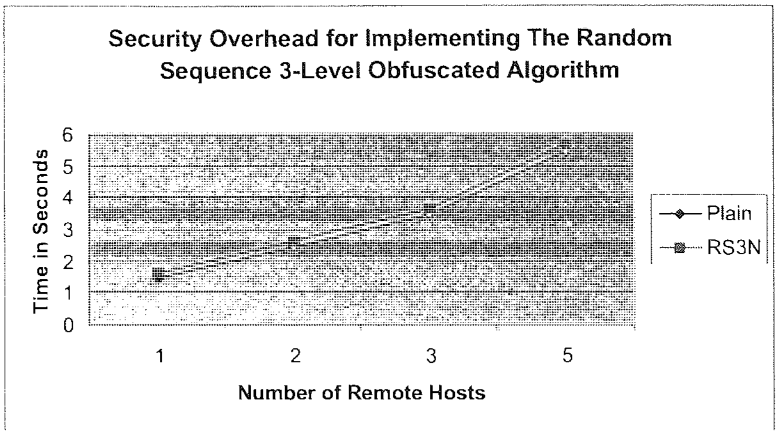


Figure 6.15: Security Overhead of The Random Sequence 3-Level Obfuscation Algorithm (100 Cycle and 1000 Obfuscation Value Experiment(with noise code))

The Recorded State Mechanism

To evaluate the security overhead for implementing the Recorded State Mechanism, four different experiments are conducted starting with one remote host, two remote hosts, three remote hosts and five remote hosts on three different types of agent: plain agent, agent with conventional cryptographic security mechanism and agent with these security mechanisms and the recorded state mechanism.

Plain agent is an agent application without any security mechanism implementation. This agent executes normal agent process such as migration and remote execution during its execution process. Whereas, agent with conventional cryptographic security mechanism is an agent application with encryption and digital signature implementation. This agent will execute these mechanisms during its execution process. In addition, agent with these security mechanisms and the recorded state mechanism is an agent application with the implementation of conventional cryptographic security mechanism and the recorded state mechanism. This agent will execute encryption and digital signature mechanisms, together with the recorded state mechanism during its execution process.

Based on the observation on the results gained through the experiments done, it can be seen that the standard error and the standard deviation of the security overhead are similar regarding the number of remote host but different between agents. Agents with security mechanism give larger standard error since the agents have to execute many tasks such as generate the cryptography key, generate digital signature, verify digital signature and execute encryption and decryption.

Number of Remote Hosts	Mean			Standard Error			Standard Deviation		
	Plain	Sec	Sec+Rec	Plain	Sec	Sec+Rec	Plain	Sec	Sec+Rec
1	1.54	29.15	28.91	0.003	2.18	1.23	0.01	9.73	5.52
2	2.49	30.89	31.11	0.003	2	1.38	0.02	8.93	6.19
3	3.37	31	32	0.004	0.93	0.98	0.02	4.15	4.38
5	5.35	36.36	36.03	0.011	1.5	1.87	0.05	6.7	8.37

Table 6.14: Summary Statistics of The Recorded State Mechanism Overhead (1 Input and 1 Cycle Experiment)

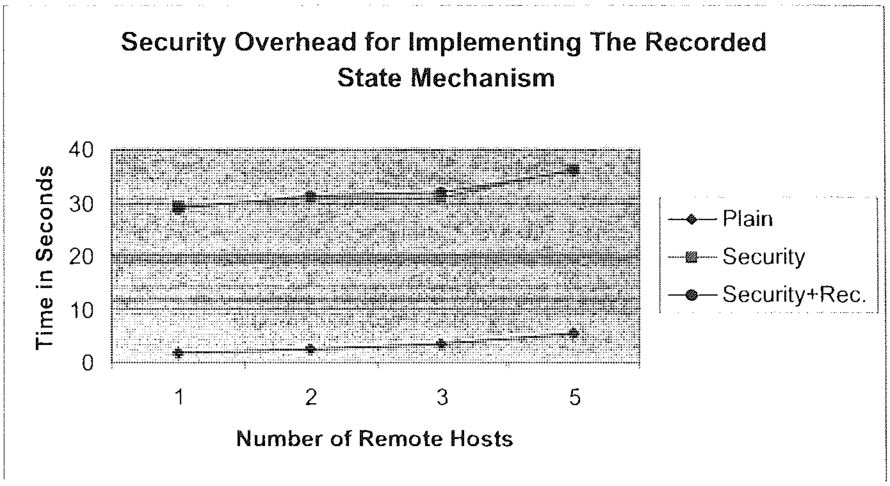


Figure 6.16: Security Overhead of The Recorded State Mechanism (1 Input and 1 Cycle Experiment)

From the results given in Table 6.14 and illustrated in Figure 6.16, it can be seen that the mean of the security overhead is almost the same for agents with security mechanism and agents with security mechanism plus RSM, where the security overhead for the agent with security mechanism plus RSM is just 7.82 % higher than the overhead for the agent with security mechanism. However, both agent's security overheads are higher by up to 1792.86 % than the overhead for the plain agent.

Number of	Mean			Standard Error			Standard Deviation		
Remote Hosts	Plain	Sec	Sec+Rec	Plain	Sec	Sec+Rec	Plain	Sec	Sec+Rec
1	1.55	45.43	45.32	0.003	1.21	1.19	0.01	5.42	5.34
2	2.53	45.99	48.21	0.005	1.35	1.83	0.02	6.03	8.18
3	3.37	49.76	49.42	0.002	1.13	1.26	0.01	5.03	5.65
5	5.43	53.09	53.14	0.005	1.09	1.76	0.02	4.86	7.85

Table 6.15: Summary Statistics of The Recorded State Mechanism Overhead (100 Input and 1 Cycle Experiment)

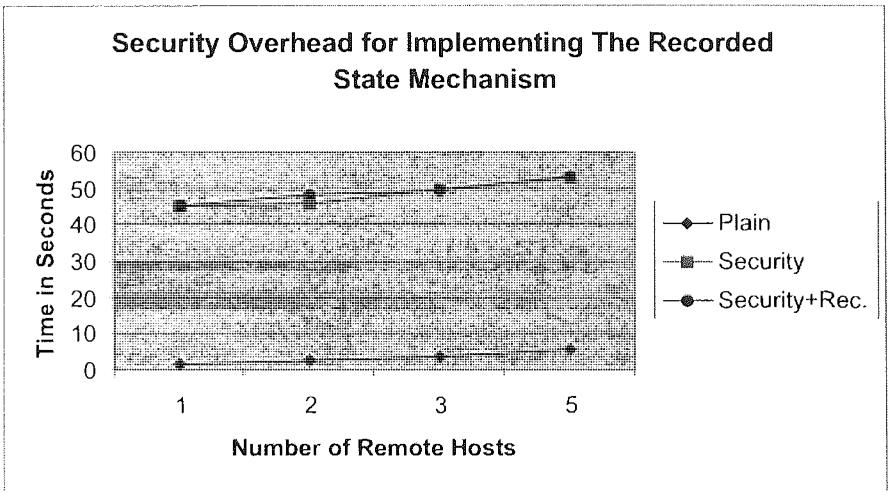


Figure 6.17: Security Overhead of The Recorded State Mechanism (100 Input and 1 Cycle Experiment)

From Table 6.15 and Figure 6.17, the security overhead for the plain agent is almost the same as with one input given in Table 6.14 and Figure 6.16, but the security overhead for the agents with security mechanism is increased by up to 60.52 % along the security overhead with one input.

Results in Table 6.16, Table 6.17, Figure 6.18 and Figure 6.19 show that the security overhead for all the agents is similar to the security overhead of the agents with the same number of input but different number of cycle given in Table 6.14, Table 6.15, Figure 6.16 and Figure 6.17 respectively. Therefore, it is worth noting that number of cycles does not affect the security overhead of the agents.

Number of	Mean			Standard Error			Standard Deviation		
Remote Hosts	Plain	Sec	Sec+Rec	Plain	Sec	Sec+Rec	Plain	Sec	Sec+Rec
1	2.41	27.65	28.94	0.008	0.77	1.55	0.04	3.45	6.94
2	3.53	30.91	31.31	0.008	1.38	1.68	0.04	6.18	7.53
3	5.17	30.94	33.36	0.005	1	1.04	0.02	4.46	4.65
5	7.34	38.46	37.57	0.01	1.29	1.69	0.05	5.76	7.55

Table 6.16: Summary Statistics of The Recorded State Mechanism Overhead (1 Input and 10000 Cycle Experiment)

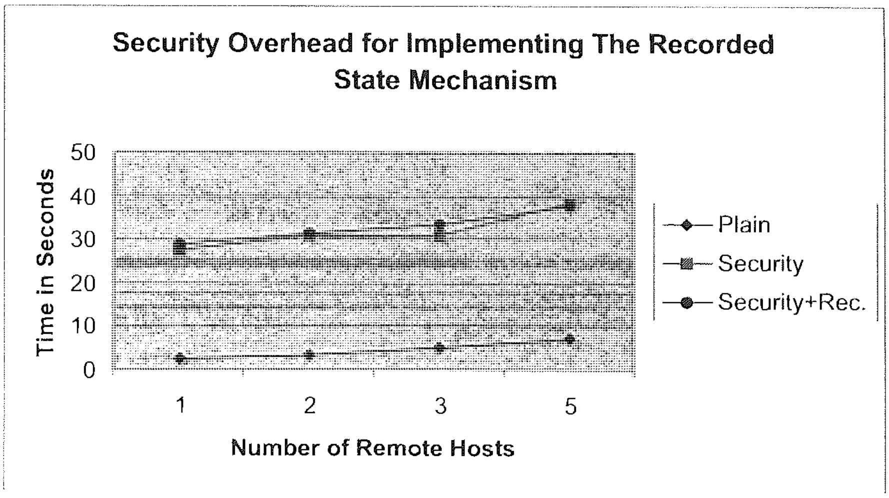


Figure 6.18: Security Overhead of The Recorded State Mechanism (1 Input and 10000 Cycle Experiment)

Summary of experimental results

It can be seen from the results shown in Tables 6.14 to 6.17 and illustrated in Figures 6.16 to 6.19 that the implementation of the Recorded State Mechanism does increase

Number of	Mean			Standard Error			Standard Deviation		
Remote Hosts	Plain	Sec	Sec+Rec	Plain	Sec	Sec+Rec	Plain	Sec	Sec+Rec
1	2.39	45.93	47.26	0.005	1.06	1.11	0.02	4.76	4.97
2	3.56	48.63	48.85	0.005	0.73	1.42	0.02	3.28	6.36
3	5.17	50.86	51.54	0.008	1.52	1.57	0.03	6.82	7.01
5	7.4	54.36	54.48	0.015	1.8	1.94	0.07	8.05	8.66

Table 6.17: Summary Statistics of The Recorded State Mechanism Overhead (100 Input and 10000 Cycle Experiment)

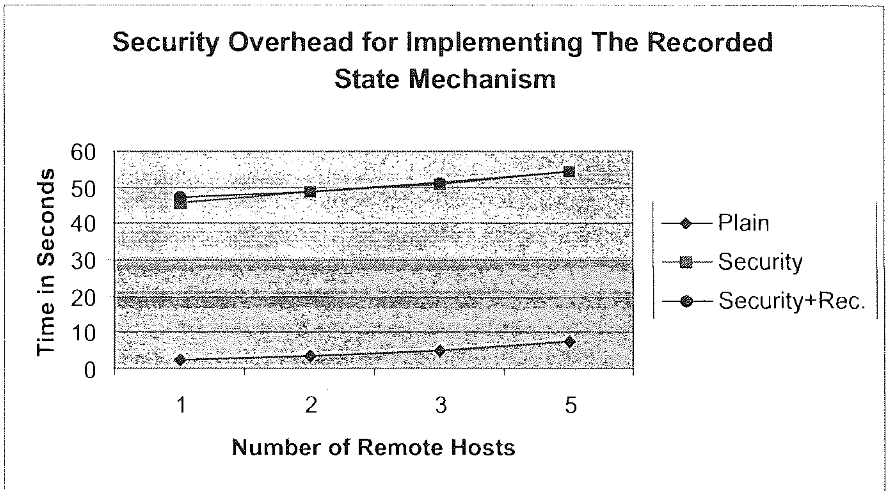


Figure 6.19: Security Overhead of The Recorded State Mechanism (100 Input and 10000 Cycle Experiment)

the overhead by only up to an acceptable 7.82 % when compared to the agent with security mechanism but 2830.96 % when compared to the plain agent. However, in a real world application, the low overhead of the plain agent is not important since the plain agent does not have any security protection. Unsecured agent in a real world application will be vulnerable to attacks and it is consider unacceptable to not provide any security. So in practice, plain agents would never be used. On the other hand, the 7.82 % increase in overhead when comparing with the agent with security mechanism is still negotiable since more protection is offered for a very small increase in overhead.

6.4 Concluding remarks

In this chapter, thirty two experiments were done in order to test two hypotheses that are the distributed migration pattern has a lower network overhead than the itinerary migration pattern and the performance of an agent-based application equipped with security mechanisms is slower than the agent-based application without security mechanisms. The first hypothesis is used to choose the better migration pattern in term of its migration and migration+computation speeds, while the second hypothesis is use to examine the level of overhead for implementing the Random 3-level obfuscation algorithm and the Recorded State Mechanism, to see whether is acceptable or not. Based on the results obtained, both the hypotheses are proven to be true.

Part V

Conclusion

Chapter 7

Evaluations and Conclusions

7.1 Introduction

This chapter summarises the findings of the thesis. These consist of evaluation of the research, suggestions for further research and conclusion.

7.2 Aims of Research

This section reiterates the aims of this research. The aims of this thesis were:

1. to propose security mechanisms for protecting agents against malicious host attack, which has resulted in;

   (a) the Random Sequence 3-level obfuscation algorithm, to prevent spying attack, and

   (b) the Recorded State Mechanism, to offer protection against manipulation attack.

2. to assess the effectiveness of the security mechanisms in protecting the agent;

   (a) the evaluation of the Random Sequence 3-level obfuscation algorithm protection capabilities, and

   (b) the evaluation of the Recorded State Mechanism protection capabilities.

3. to develop and carry out tests for evaluating the security mechanisms in terms of their overhead;
   (a) the evaluation of the migration and migration+computation overheads of the itinerary and distributed migration patterns, and
   (b) the evaluation of the security overhead of the security mechanisms.
4. to develop a prototype system to implement the security mechanisms;
   (a) to test the feasibility of the security mechanisms in real agent-based applications.

7.3 Evaluation

This section presents the evaluation of this research. The evaluation includes the literature review, the proposed security mechanism, the experiments and the prototype system.

7.3.1 Evaluation of the security mechanisms

Referring to the definition of attack in Section 3.2, the malicious host could launch an attack on the executing agent's code, data or state during its execution process inside the malicious host environment. This definition leads to interpret an attack as either an unexpected change in the behaviour of an agent or unauthorised access to information.

In this research, two security mechanisms, the Random Sequence 3-level obfuscation algorithm to prevent unauthorised access to information and the Recorded State Mechanism to detect unexpected change in the behaviour of an agent are proposed. Both security mechanisms will be evaluated in the next subsections.

The Random Sequence 3-level obfuscation algorithm

The Random Sequence 3-level obfuscation algorithm is an algorithm that is used to prevent spying attack on agent's critical data by a malicious host. The algorithm obfuscates the actual value of the agent's critical data to an obfuscated value that is meaningless to the malicious host. This makes it difficult for the malicious host to spy on the agent's critical data, thus preventing malicious host spying attack. However, there are a few attacks in which the malicious host could attempt to break the Random Sequence 3-level obfuscation algorithm. Therefore the evaluation on the protection capabilities of the Random Sequence 3-level obfuscation algorithm will be discussed below.

- **Spying on agent's data**

  If the malicious host is given enough time to execute, the malicious host can attack the RS3 obfuscation algorithm using binary search, which uses different result values and watch the pattern of the RS3 obfuscation algorithm outcomes (which result value the agent accepts and which it rejects) to guess the actual value of agent's critical data. Furthermore, if the malicious host does not care about the cost of breaking the RS3 obfuscation algorithm, the malicious host can employ several computers to execute the RS3 obfuscation algorithm using different result values in parallel and watch the pattern of the RS3 obfuscation algorithm outcomes to guess the actual value of agent's critical data.

  This attack can be ruled out by introducing a time factor to the agent that carries the obfuscation value to make the agent valid only for a limited period of time (*Hohl*, 1998a). However, the user cannot decide the effective protection interval of the RS3 obfuscation algorithm for each transaction. In order to overcome the problem of determining the effective protection interval for the RS3 obfuscation algorithm, noise codes are introduced to the agent application that is executing in the remote host to make it more difficult for the malicious host to guess the actual value of the user maximum budget. In addition, the implementation of the noise code could also delay the analysing process of the obfuscation algorithm by the malicious host. Therefore, the use of an effective protection interval to enhance the level of obfuscation algorithm protection is less important.

The malicious host can discover the actual value of an agent's critical data in advance (in the case where a remote host dispatches successive agents to perform the same kind of task) by spying on the previous visiting agent's code, data and state, and use it to analyse the RS3 obfuscation algorithm to attack the next visiting agent that comes from the same host.

This attack can be ruled out by using a different obfuscated value for each time a slave agent is dispatched to execute a transaction with the remote host. The different obfuscated value can be generated using a new random number for each agent to convert agent's critical data.

In addition, if the conversion process and the knowledge about the actual value of agent's critical data are not removed from the agent's code, data and state, before the agent migrates to the remote host, the malicious host can spy on the initial conversion process of the actual value of agent's critical data to the obfuscated value.

This attack can be ruled out by using the master-slave agent architecture in implementing the RS3 obfuscation algorithm. This is because by using master-slave agent architecture, the initial conversion process can only be allowed to execute inside the owner host and by the master agent that is not directly in transaction with the remote hosts. Only the slave agent will transact with the remote hosts on behalf of the master agent and no knowledge of the actual value of the agent's critical data is revealed to the slave agent in order to protect the confidentiality of the actual value and to protect the value from malicious host spying attack.

The Recorded State Mechanism

The Recorded State mechanism is designed to detect manipulation attacks by detecting inconsistencies or changes in the agent's data or state. However, there are a few attacks in which a malicious host could use to get around the Recorded State Mechanism. Therefore the evaluation on the protection capabilities of Recorded State Mechanism will be discussed below.

- **Manipulation attack on agent's data and state**

  The malicious host could make subtle changes to the read-only data inside the RecordedReadOnly container to enable it to achieve its objective, in such a way that the owner's digital signature that was signed in the RecordedReadOnly container still remains valid.

  This attack can be ruled out, because the digital signature (using SHA1), which is used in the Recorded State Mechanism is secured against brute-force collision and inversion attacks, where by using the SHA1 as the digital signature function could make the attacker computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest.

  The malicious host could also make subtle changes to both the read-only data and the digital signature of the RecordedReadOnly container, in order to make both of them appear to be valid.

  This possible attack can also be ruled out because in order to create a new digital signature that will be valid for other host, the malicious host needs to have a private key of the agent owner. However, only the key owner has this key and no other entity can produce this key from a modified hash value.

  In addition, the malicious host could tamper with the agent state recorded in the RecordedExecuteOnly and RecordedCollectOnly container by modifying the agent state before the state is recorded into both containers. This is due to the fact that the recorded process of the Recorded State Mechanism is under the malicious host's control and therefore, the malicious host can do anything to it.

  This attack can be ruled out because the malicious host has to use its own private key that contains its identity in order to compute and re-compute the digital signature of the tampered state, thus revealing itself during the Recorded State Mechanism evaluation process.

- **Manipulation attack on agent's input data**

  The malicious host could lie about the input data, which is recorded in the RecordedExecuteOnly and the RecordedCollectOnly containers in order to deceive the owner of the agent.

  The attack can be ruled out because the owner of the agent knows the identity of the host, which supplies the input data to the agent because all the data and state are digitally signed by the execution host before the data and state left the execution host. Thus, the owner of the agent knows which execution host is responsible for supplying the false input data.

- **Collaboration attack**

  The malicious host could attack the agent by launching collaboration attacks in cooperation with two or more consecutive hosts in order to deny the checking process for detecting any malicious host attack from the previous visit or to remove any agent state that records the changes made by the previous host on the agent during its execution session.

  The attack can be ruled out since the used of master-slave agent architecture in implementing the Recorded State Mechanism only allows different agents to be sent and served by different remote hosts. An agent only visits one host, thus precluding the collaboration attack.

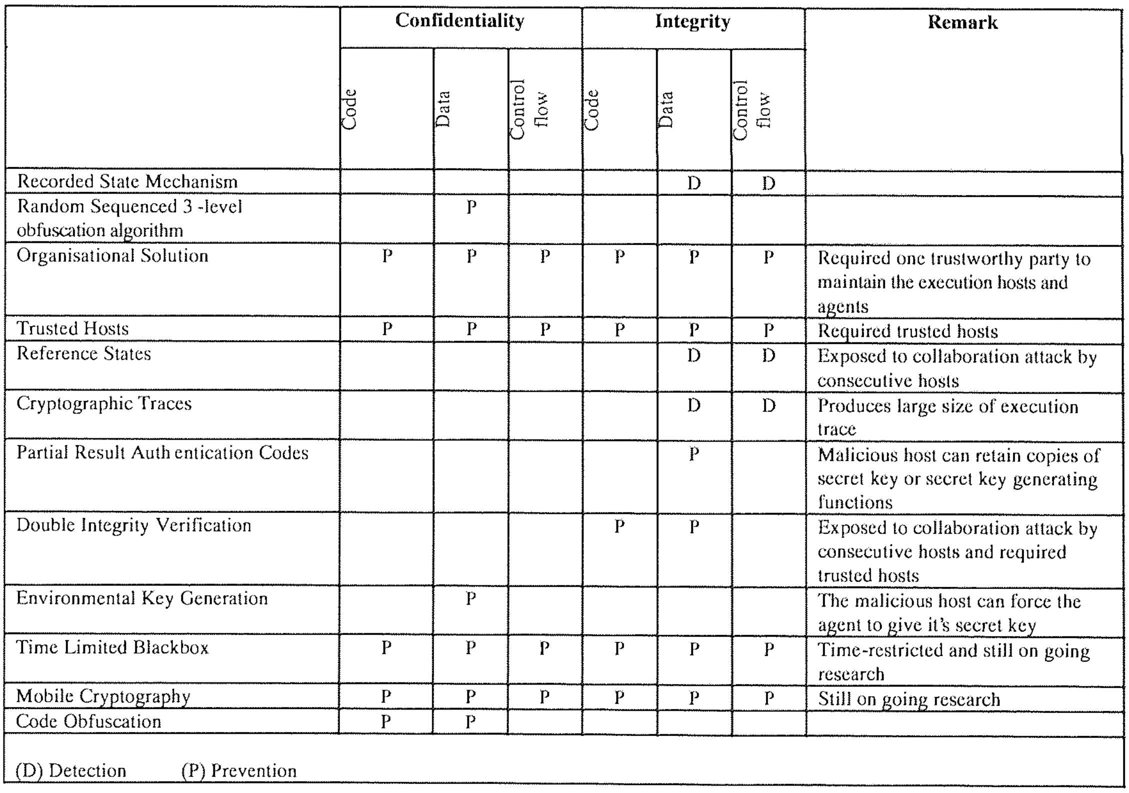
- **Incorrect execution of code attack**

  The malicious host could also alter some agent codes and execute them in many different ways.

  This attack can be ruled out since the Recorded State Mechanism will check the results gathered by the returning slave agent by re-executing the same execution process done by the slave agent inside the remote host execution environment.

Summary of evaluation of security mechanisms

Table 7.1 shows the summary of the protection abilities for the Random Sequence 3-level obfuscation algorithm and the Recorded State Mechanism, together with other security approaches. Although in Table 7.1, there are four security approaches: Organisational Solution, Trusted Hosts, Time Limited Blackbox and Mobile Cryptographic, that are capable to providing full security protection for the agent, these approaches are not mature enough to be used. However, other approaches such as Random Sequence 3-level obfuscation algorithm, Recorded State Mechanism, Reference States, Cryptographic Traces, Partial Result Authentication Codes, Double Integrity Verification, Environmental Key Generation and Code Obfuscation, are only capable of providing partial protection for the agent.



	Confidentiality			Integrity			Remark
	Code	Data	Control flow	Code	Data	Control flow	
Recorded State Mechanism					D	D	
Random Sequenced 3 -level obfuscation algorithm		P					
Organisational Solution	P	P	P	P	P	P	Required one trustworthy party to maintain the execution hosts and agents
Trusted Hosts	P	P	P	P	P	P	Required trusted hosts
Reference States					D	D	Exposed to collaboration attack by consecutive hosts
Cryptographic Traces					D	D	Produces large size of execution trace
Partial Result Auth entication Codes					P		Malicious host can retain copies of secret key or secret key generating functions
Double Integrity Verification				P	P		Exposed to collaboration attack by consecutive hosts and required trusted hosts
Environmental Key Generation		P					The malicious host can force the agent to give it's secret key
Time Limited Blackbox	P	P	P	P	P	P	Time-restricted and still on going research
Mobile Cryptography	P	P	P	P	P	P	Still on going research
Code Obfuscation	P	P					

(D) Detection (P) Prevention

Table 7.1: Security mechanisms and their protection abilities

In this thesis, confidentiality and integrity protection are two main requirements for protecting agents against a malicious host attacks. These requirements are successfully fulfilled[1] by the Random Sequence 3-level obfuscation algorithm and the Recorded State Mechanism. Both security mechanisms are able to protect the confidentiality and integrity of the agents from most of the malicious host attacks.

7.3.2 Evaluation of the experimental tests

There are two sets of experiments conducted in this thesis. The first set of the experiments was the comparison experiment between the Itinerary and the Distributed Migration Pattern. This comparison experiment is conducted to examine the performance of both migration patterns in terms of their migration and migration+computation overhead. The second set of the experiments was conducted to measure the overhead of implementing security protection mechanisms into an agent-based application. These experiments were done to analyse the security overhead of the Random Sequence 3-Level Obfuscation Algorithm and the Recorded State Mechanism.

The comparison experiment between the Itinerary and Distributed Migration Pattern is used to investigate the hypothesis that the performance of the proposed model that uses the Distributed Migration Pattern with master-slave agent architecture is faster than the model of Itinerary Migration Pattern using single agent architecture, even though the proposed model is required to generate and dispatch more than one agents to execute a transaction, which does increase the network overhead at the originating host. The experimental results gathered from both experiments prove that the hypothesis is true. This is shown by the experimental results that both migration pattern perform equally but as the number of remote host increases, the Distributed Migration Pattern gives lower migration overhead by up to 14.17% and the migration+computation overhead for the Distributed Migration Pattern is up to 95.65% lower than the migration+computation overhead for the Itinerary Migration Pattern except for one remote host experiment where the migration+computation overhead for both migration pattern is almost the same where the difference is just within 3.35%.

[1] blocked some attacks and made others more difficult

The second set of experiments is used to investigate the hypothesis that the performance of an agent-based application equipped with security mechanisms is slower than the agent-based application without security mechanisms. The experimental results gathered from both experiments prove that this hypothesis is also true. This proof is shown by the experimental results for both the Random Sequence 3-level Obfuscation Algorithm and the Recorded State Mechanism experiments, where both security protection mechanisms does increase the overhead but the agent confidentiality and integrity can be protected against malicious host attacks.

7.3.3 Evaluation of the prototype application

The Secure Flight Finder Agent-based System (SecureFAS) prototype was constructed to implement the proposed security mechanisms in order to develop a secure agent-based application that is able to detect and prevent the malicious host attacks. The prototype was developed using the Java programming language, due to the benefit that the Java language is designed to operate in heterogeneous environments because it has no platform-dependent aspects. In addition, the Aglets Software Development Kit (ASDK) that is used for developing an agent-based application in this research also uses the Java language as its programming language.

The prototype was also constructed to prove that:

- the proposed security mechanisms as developed are fully functional, and
- the proposed security mechanism can be applied into a real world agent-based application to secure the application against the malicious host attack.

Summary of evaluation of the prototype application

The SecureFAS prototype developed is successful in testing the functionality of the proposed security mechanisms and in performing tasks of protecting an agent against a malicious host attacks. In addition, the prototype has showed that the proposed security mechanisms can be applied in real world applications.

7.3.4 Evaluation of the suitability of the proposed security mechanisms for different agents applications

The Random Sequence 3-level obfuscation algorithm and the Recorded State Mechanism are two security mechanisms that were proposed to protect the confidentiality and the integrity of an agent against a malicious host attack. These security mechanisms are useful especially for an agent-based application such as e-commerce, and monitoring and notification application that might carry sensitive information for their execution process inside the remote host execution environment, which could be a malicious host that might try to attack or abuse the agent. By implementing the proposed security mechanisms, the malicious host attacks such as spying, manipulation, extraction of information, collaboration and repudiation could be addressed. On the other hand, it is not recommended to implement these security mechanisms to applications such as parallel processing, information gathering and dissemination because they do not carry any sensitive information that could attract the malicious host to attack. Therefore, the overhead incurred for implementing the security mechanisms will only be an extra burden.

7.4 Recommendations for future work

There are a number of areas for future development in this area. The list is provided below:

- The Random Sequence 3-level Obfuscation algorithm that is currently used to protect the confidentiality of the agent, can only obfuscate numbers and not characters. Therefore, this obfuscation algorithm is only applicable to an agent-based application that carries numbers, such as a shopper agent that buys goods based on the user budget. This algorithm can be expanded further to include facilities that can obfuscate characters and characters with numbers or vice versa by finding the way to obfuscate ASCII code that represents the characters that need to be obfuscated. This is to enable the algorithm to be used in other agent-based application.

- Another limitation of the Random Sequence 3-level Obfuscation algorithm is that the algorithm has to be carried by the agent that is executed inside the remote host to enable the agent to execute the comparison process between the user budget obfuscated value and remote host offer obfuscated value, thus exposing the complete RS3 obfuscation algorithm to the attacker (the malicious host). This problem can be overcome by obfuscating the RS3 obfuscation algorithm itself, so that the attacker does not have any knowledge about the algorithm even though the algorithm is carried by the agent that execute inside the attacker host.

- Furthermore, the Random Sequence 3-level Obfuscation algorithm is vulnerable to multiple execution attack from the malicious host, where the malicious host can execute the algorithm many times or create multiple copies of the algorithm and execute them concurrently to discover the actual value that was obfuscated. This problem can be improved further by designing a method that only allow the algorithm to be executed once and preventing copies of the algorithm using unique serial number that is only valid for one execution process.

- The confidentiality and integrity protection, has currently been implemented on the prototype system. It is possible to extend this protection to protect the agent against other malicious host attacks such as masquerading and denial of execution.

- Further tests to evaluate the strength of the proposed mechanism and algorithm could also be conducted by simulating the malicious host attacks on the agents.

- The SecureFAS prototype system can be extended to a second phase, to include the purchasing process when the best offer has been found. In this phase, a negotiation process can be introduced. In negotiation process, a shopping agent can negotiate with the virtual airline host to reduce the flight fare as low as possible.

7.5 Conclusion

This research has investigated the problem of malicious host attacking the agents and has proposed security mechanisms to increase the security of agents by making malicious host attack more difficult and forcing at best possibility identification of secret information. The experimental result shows that the distributed migration pattern does enhance the performance of an agent-based system and the agent security mechanisms impose an acceptable small time overhead. From the analysis of the proposed mechanisms under well-known attack scenarios, it can be shown that the proposed mechanisms prevent or detect some attacks and make other attacks more difficult. Security mechanisms proposed are applicable to an agent-based application such as e-commerce, and monitoring and notification but would not benefit applications such as parallel processing, and information gathering and dissemination. This research has offered significant advances in protection of agents against malicious host attack, and is suitable for use in real world applications.

Bibliography

Abu Bakar, K., and B. S. Doherty, A New Model for Protecting Mobile Agents against Malicious Host, in *Proceedings of the IADIS International Conference WWW/Internet*, pp. 780 – 784, IADIS, 2002.

Abu Bakar, K., and B. S. Doherty, A Random Sequence 3-Level Obfuscated Algorithm for Protecting Mobile Agents Against Malicious Hosts, in *Proceedings of the 2003 International Conference on Informatics, Cybernetics and Systems*, pp. 525 – 530, I-Shou University, 2003a.

Abu Bakar, K., and B. S. Doherty, Protecting Mobile Agents Against A Malicious Host Attacks Using Recorded State Mechanism, in *Proceedings of the 2003 International Conference on Informatics, Cybernetics and Systems*, pp. 396 – 401, I-Shou University, 2003b.

Bacon, J., *Concurrent Systems : Operating Systems, Database and Distributed Systems: An Integrated Approach*, 2nd ed., Addison-Wesley, United States, 1997.

Berkovits, S., J. D. Guttman, and V. Swarup, Authentication for Mobile Agents, in *G. Vigna(Ed.): Mobile Agents and Security*, vol. 1419, pp. 114 – 136, Springer Verlag, 1998.

Biehl, I., B. Meyer, and S. Wetzel, Ensuring the Integrity of Agent-Based Computations by Short Proofs, in *Mobile Agents Second International Workshop MA'98 Proceedings*, pp. 183 – 194, Springer Verlag, 1998.

Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons Ltd., 1996.

Chan, A., C. Wong, T. Wong, and M. Lyu, Design, implementation, and experimentation on mobile agent security for electronic commerce applications, in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, vol. 4, pp. 1871 – 1877, CSREA Press, 2000.

Chauhan, D., JAFMAS: A Java-based Agent Framework for Multiagent System Development and Implementation, Ph.D. thesis, ECECS Department,University of Cincinnati, United State, 1997.

Chavez, A., and P. Maes, Kasbah: An agent marketplace for buying and selling goods, in *First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96)*, pp. 75–90, Practical Application Company, 1996.

Chess, D., Security Issues in Mobile Code Systems, in *G. Vigna(Ed.): Mobile Agents and Security*, vol. 1419, pp. 1 – 14, Springer Verlag, 1998.

Chess, D., C. Harrison, and A. Kershenbaum, Mobile Agents: Are They a Good Idea?, *IBM Research Report*, Mar., 1995, http://www.research.ibm.com/iagents/publications.html.

Cockburn, D., and N. R. Jennings, ARCHON: A Distributed Artificial Intelligence System for Industrial Applications, in *Foundations of Distributed Artificial Intelligence*, edited by G. M. P. O'Hare and N. R. Jennings, pp. 319–344, John Wiley & Sons, 1996.

Collberg, C., and C. Thomborson, Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection, in *IEEE Transaction on Software Engineering*, pp. 735 – 746, IEEE, 2002.

Collberg, C., C. Thomborson, and D. Low, A Taxonomy of Obfuscating Transformations, *Tech. Rep. 148*, 1997, http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html.

Corradi, A., M. Cremonini, R. Montanari, and C. Stefanelli, Mobile Agents Integrity for Electronic Commerce Application, in *Information System*, pp. 519 – 533, Elsevier Science, 1999a.

Corradi, A., R. Montanari, and C. Stefanelli, Security Issues in Mobile Agent Technology, in *Proceedings 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, pp. 3 – 8, IEEE Computer Society, 1999b.

Coulouris, G., J. Dollimore, and T. Kindberg, *Distributed Systems, Concept and Design*, 3rd ed., Pearson Education Ltd., England, 2001.

Couriertravel, http://www.couriertravel.org/flightfinder.asp, 2004.

Devargas, M., *Network Security*, NCC Blackwell Ltd., United Kingdom, 1993.

Diaz, J., D. Gutierrez, and J. Lovelle, An Implementation of A Secure Java2-Based Mobile Agent System, in *Proceedings of The Second International Conference on The Practical Application of Java*, pp. 125 – 142, Practical Application Company, 2000.

Farmer, W., J. Guttman, and V. Swarup, Security for Mobile Agents: Issues and Requirements, in *Proceedings of the 19th National Information System Security Conference*, pp. 591 – 597, Baltimore, 1996a.

Farmer, W., J. Guttman, and V. Swarup, Security for Mobile Agents: Authentication and State Appraisal, in *Computer Security-ESORICS 96 4th European Symposium on Research in Computer Security Proceedings*, pp. 118 – 130, Springer Verlag, 1996b.

FlightFound, http://www.flightfound.com/eng/index.asp, 2004.

Foner, L., What is an Agent Anyway? A Sociological Case Study, *Agents Memo 93-01*, 1993, http://foner.www.media.mit.edu/people/foner/Reports/Julia/Agents-Julia.ps.

Ford, W., and M. Baum, *Secure Electronic Commerce*, 2nd ed., Prentice Hall, New Jersey, United State, 2001.

Franklin, S., and A. Graesser, Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents, in *Intelligent Agent III. Agent Theories, Architectures and Languages.*, pp. 21 – 35, Springer Verlag, 1997.

Funfrocken, S., Transparent Migration of Java-based Mobile Agents: Capturing and re-establishing the State of Java Programs, in *Personal Technologies*, vol. 1.2, pp. 109 – 116, Springer Verlag, 1998.

Gray, R., D. Kotz, G. Cybenko, and D. Rus, D'Agents: Security in a Multiple-Language, Mobile-Agent System, in *G. Vigna(Ed.): Mobile Agents and Security*, vol. 1419, pp. 154 – 187, Springer Verlag, 1998.

Guan, X., Y. Yang, and J. You, POM - A Mobile Agent Security Model against Malicious Hosts, in *Proceedings of IS & N'99*, pp. 155 – 167, Spring Verlag, 1999.

Hayzelden, A., and J. Bigham, Agent Technology in Communication Systems: An Overview, in *The Knowledge Engineering Review*, vol. 14, pp. 341 – 375, Cambridge University Press, 1999.

Hohl, F., An Approach to Solve the Problem of Malicious Hosts, *Tech. rep.*, Institute of Parallel and Distributed High-Performance Systems (IPVR), University of Stuttgart, Germany, 1997.

Hohl, F., Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts, in *In: G. Vigna (Ed.). Mobile Agent and Security. Lecture note in Computer Science*, vol. 1419, pp. 92 – 113, Springer Verlag, Berlin, 1998a.

Hohl, F., A Model of Attacks of Malicious Hosts Against Mobile Agents, *In 4th ECOOP Workshop on Mobile Object Systems (MOS'98): Secure Internet Mobile Computations*, 1998b, http://mole.informatik.uni-stuttgart.de/papers.html.

Hohl, F., A Protocol to Detect Malicious Hosts Attacks by Using Reference States, *Tech. rep.*, Institute of Parallel and Distributed High-Performance Systems (IPVR), University of Stuttgart, Germany, 1999.

Hohl, F., A Framework to Protect Mobile Agents by Using Reference States, in *In: Proceedings of the 20th international conference on distributed computing systems (ICDCS 2000)*, pp. 410 – 417, IEEE Computer Society, 2000.

Hughes, A., Is Remote Evaluation a realistic alternative to Remote Procedure Call in a portable distributed application?, Computing Science Honors Degree Thesis, 2001, http://citeseer.nj.nec.com/hughes01is.html.

IBM, The Aglets Software Development Kit, Version 1.1 Beta 3, http://www.trl.ibm.com/aglets/idoagree11b.htm, 1998.

Ince, D., *Developing Distributed and E-commerce Applications*, Addison-Wesley, United States, 2002.

Jansen, W., Countermeasures for mobile agent security, in *Computer Communications*, vol. 23, pp. 1667 – 1676, Elsevier Science, 2000.

Kotzanikolaou, P., M. Burmester, and V. Chrissikopoulos, Secure Transactions with Mobile Agents in Hostile Environments, in *Proceedings of the 5th Australasian Conference (ACISP 2000)*, vol. 1841, pp. 289 – 297, Springer Verlag, 2000.

Krulwich, B., The BargainFinder agent: Comparison price shopping on the Internet, in *Bots and other Internet Beasties*, pp. 257 – 263, Macmillan Computing Publishing, 1996.

Kun, Y., G. Xin, and L. Dayou, Security in Mobile Agent System: Problems and Approaches, in *Operating System Review*, vol. 34, pp. 21 – 28, ACM, 2000.

Lange, D., and M. Oshima, *Programming and Deploying Java Mobile Agent with Aglets*, Addison Wesley, United States, 1998.

Lange, D., and M. Oshima, Seven good reasons for mobile agents, in *Communication of The ACM*, vol. 42, pp. 88 – 89, ACM, 1999.

Luckham, D., *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, Addison Wesley, 2002.

Maes, P., Artificial Intelligence meets Entertainment: Lifelike Autonomous Agents, in *Communication Of The ACM*, vol. 38, pp. 108 – 114, ACM, 1995.

Mandry, T., G. Pernul, and A. Rohm, Mobile Agents in Electronic Markets: Opportunities, Risks, Agent Protection, in *International Journal of Electronic Commerce*, pp. 47 – 60, M.E. Sharpe, 2001.

Marques, P. J., L. M. Silva, and J. G. Silva, Establishing a Secure Open-Environment for Using Mobile Agents in Electronic Commerce, in *First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents*, pp. 268 – 269, IEEE, 1998.

Meadows, C., Detecting attacks on Mobile Agents, in *Proceedings 1997 Foundations for Secure Mobile Code Workshop*, pp. 64 – 65, 1997, http://citeseer.nj.nec.com/meadows97detecting.html.

Minar, N., Designing an Ecology of Distributed Agents, Master's thesis, Massachusetts Institute of Technology, 1998.

Muller, P., *Instant UML*, Wrox Press Ltd., Canada, 2000.

Necula, G., and P. Lee, Safe, Untrusted Agents Using Proof-Carrying Code, in *G. Vigna(Ed.): Mobile Agents and Security*, vol. 1419, pp. 61 – 91, Springer Verlag, 1998.

Ng, S. K., and K. W. Cheung, Protecting Mobile Agents against Malicious Hosts by Intention Spreading, in *In H. Arabnia (ed.), Proc. International Conference on*

Parallel and Distributed Processing Techniques and Applications (PDPTA'99), pp. 725 – 729, CSREA, 1999a.

Ng, S. K., and K. W. Cheung, Intention Spreading: An extensible theme to protect mobile agents from read attack hoisted by malicious hosts, in *In Jimming Liu, Ning Zhong(ed.), Intelligent Agent Technology: Systems, Methodologies and Tools*, pp. 406 – 415, World Scientific, 1999b.

NIST, Secure Hash Standard (SHS), in *National Institute of Standards and Technology*, Federal Information Processing Standards Publication (FIPS Pub 180), 1993, http://www.itl.nist.gov/fipspubs/fip180-1.htm.

NIST, Announcement of Technical Correction To Secure Hash Standard, in *National Institute of Standards and Technology*, 1994, http://www.nist.gov/public_affairs/releases/hashstan.htm.

Noble, B., and M. Satyanarayanan, Experience with adaptive mobile applications in Odyssey, in *Mobile Networks and Applications*, vol. 4, pp. 245 – 254, Baltzer Science, 1999.

Nwana, H., Software Agents: An Overview, in *The Knowledge Engineering Review*, vol. 11, pp. 205 – 244, Cambridge University Press, 1996.

ObjectSpace, *ObjectSpace Voyager Core Technology*, 1st ed., ObjectSpace, Inc., 1997.

Oppliger, R., Security issues related to mobile code and agent-based systems, in *Computer Communication*, vol. 22, pp. 1165 – 1170, Elsevier Science, 1999.

Ousterhout, J., *Tcl and the Tk toolkit*, Addison-Wesley, New Jersey, United States, 1994.

Perraju, T., Agents and Autonomous Distributed Systems, in *Proceedings. Fourth International Symposium on Autonomous Decentralized Systems - Integration of Heterogeneous System*, pp. 264 – 266, IEEE Computer Society, 1999.

Peterson, L., and B. Davie, *Computer Networks: A Systems Approach*, 3rd ed., Morgan Kaufmann, 2003.

Pfleeger, C., *Security in Computing*, 2nd ed., Prentice-Hall Inc., New Jersey, United States, 1997.

Reisner, J., and E. Donkor, Protecting Software Agents from Malicious Hosts using Quantum Computing, in *Proceedings of SPIE - The International Society for Optical Engineering*, pp. 50 – 57, IEE, 2000.

Riordan, J., and B. Schneier, Environmental Key Generation Towards Clueless Agents, in *In: G. Vigna (Ed.). Mobile Agent and Security. Lecture note in Computer Science*, vol. 1419, pp. 15 – 24, Springer Verlag, Berlin, 1998.

Rivest, R. L., A. Shamir, and L. M. Adleman, A method for obtaining digital signatures and public-key cryptosystems, in *Communication of The ACM*, vol. 2, pp. 120 – 126, ACM, 1998.

Roff, J. T., *UML: A Beginner's Guide*, McGraw-Hill, United States, 2003.

Rus, D., R. Gray, and D. Kotz, Transportable Information Agents, in *Proceedings of the First ACM International Conference on Autonomous Agents*, pp. 228 – 236, ACM, 1997.

Russell, D., and G. Gangemi, *Computer Security Basics*, O' Reilly & Associates, Inc., United States, 1991.

Sander, T., and C. Tschudin, Protecting Mobile Agent Against Malicious Hosts, in *In: G. Vigna (Ed.). Mobile Agent and Security. Lecture note in Computer Science*, vol. 1419, pp. 44 – 60, Springer Verlag, Berlin, 1998a.

Sander, T., and C. Tschudin, Towards Mobile Cryptography, in *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, Oakland, CA, USA, 1998b.

Sander, T., and C. Tschudin, On Software Protection via Function Hiding, in *Proceedings of the Information Hiding. Second International Workshop, IH'98.*, pp. 111–123, Springer Verlag, 1998c.

Schelderup, K., and J. Olnes, Mobile Agent Security: Issues and Directions, in *Proceedings of IS&N'99*, pp. 155 – 167, Springer Verlag, 1999.

Schneier, S., *Applied Cryptography*, 2nd ed., Wiley & Son, New York, United States, 1996.

Schoder, D., and T. Eymann, The Real Challenges of Mobile Agents, in *Communication of The ACM*, vol. 43, pp. 111 – 112, ACM, 2000.

Silva, L., G. Soares, P. Martins, V. Batista, and L. Santos, Comparing the performance of mobile agent systems: a study of Benchmarking, in *Computer Communication*, vol. 23, pp. 769 – 778, Elsevier Science, 2000.

Stallings, W., *Cryptography and Network Security : Principle and Practice*, 2nd ed., Prentice-Hall Inc., New Jersey, United States, 1999.

Stamos, J. W., and D. K. Gifford, Remote Evaluation, in *ACM Transaction on Programming Languages and Systems (TOPLAS)*, vol. 12, pp. 537 – 564, ACM Press, 1990.

Sun Microsystems, I., Java 2 Platform Std. Ed. V1.3.1, http://java.sun.com/j2se/1.3/docs/api/index.html, 2004.

Sundsted, T., An introduction to agents, *JavaWorld*, 1998, http://www.javaworld.com/javaworld/jw-06-1998/jw-06-howto.html.

Tai, H., The Aglets Project, in *Communication of The ACM*, vol. 42, pp. 100 – 101, ACM, 1999.

Travelocity, http://www.travelocity.co.uk/, 2004.

Tripathi, A., and N. Karnik, Protected Resource Access for Mobile Agent-based Distributed Computing, in *Workshop on Architectural and Operation Systems Support for Multimedia Applications*, pp. 144 – 153, IEEE Computer Society, 1998.

Tripathi, A., N. Karnik, T. Ahmad, R. Singh, A. Prakash, V. Kakani, M. K. Vora, and M. Pathak, Design of the Ajanta System for mobile agent programming, in *The Journal of Systems and Software*, vol. 62, pp. 123 – 140, Elsevier Science, 2002.

Vigna, G., Cryptographic Traces for Mobile Agents, in *In: G. Vigna (Ed.). Mobile Agent and Security. Lecture note in Computer Science*, vol. 1419, pp. 137 – 153, Springer Verlag, Berlin, 1998.

Wang, T., S. Guan, and T. Chan, Integrity protection for Code-on-Demand mobile agents in e-commerce, in *The Journal of Systems and Software*, vol. 60, pp. 211 – 221, Elsevier, 2002.

West, R., and J. Gloudon, User-Level Sandboxing: a Safe and Efficient Mechanism for Extensibility, *Tech. rep.*, Computer Science Department, Boston University, Boston, 2003.

Wong, D., N. Paciorek, and D. Moore, Java-based Mobile Agents, in *Communication of The ACM*, vol. 42, pp. 92 – 102, ACM, 1999.

Yee, B. S., A Sanctuary for Mobile Agents, *Tech. rep.*, University of California in San Diego, 1997, http://www.cse.ucsd.edu/users/bsy/index.html.

Zhu, F., S. Guan, and Y. Yang, SAFER e-commerce: secure agent fabrication, evolution and roaming for e-commerce., in *Internet Commerce and Software Agents: Cases, Technologies and Opportunities.*, pp. 190 – 206, IDEA Group Publishing, 2000.

Part VI

Appendices

Appendix A

The Aglets Software Development Kit Configuration

This manual provides information on how to configure the Aglets Software Development Kit (ASDK) version 2.0 that was used in developing the SecureFAS prototype system in this research. The content of this manual is based on the Aglets Software Development Kit documentation version 1.1 Beta 3 released by the IBM Corporation (*IBM*, 1998) with some modification to suit the configuration of the ASDK version 2.0.

There are five configuration steps to follow in configuring the Aglets Software Development Kit:

1. Downloaded the ASDK Installation Source
2. System Requirements
3. Configure Server Properties
4. Installation
5. Start Up The Tahiti Server

A.1 Step 1. Downloaded the ASDK Installation Source

The Aglets Software Development Kit version 2.0 installation source can be downloaded from http://www.trl.ibm.com/aglets/. This installation source is licensed under the IBM Public License and can be downloaded for free.

A.2 Step 2. System Requirements

There are two system requirements for the Aglets Software Development Kit version 2.0:

1. The Aglets Software Development Kit version 2.0 required Java Development Kit (JDK) 1.2 or above to be installed, and
2. The ASDK with JDK 1.2 or above is available for SPARC/Solaris 2.5 or above, Windows 95/98/2000/NT, AIX 4.1.4 and *OS/2 Warp 4.

A.3 Step 3: Configure Server Properties

The server properties file can be created by manipulating the "sample_aglets.props" file that was included in the ASDK version 2.0 distributed package. In the "sample_aglets.prop" file, user is required to specify the "aglets.home" property, while other properties can be left unspecified because they have a default value. An example of "aglets.home" property for Windows 95/98/2000/NT is presented as:

```
aglets.home=C:\\Aglets2.0
```

and for Unix as:

```
aglets.home=/home/username/Aglets2.0
```

A.4 Step 4: Installation

Install the ASDK version 2.0 using the following installation steps:

For Windows 95/98/2000,NT, OS/2

1. Unzip the ASDK distribution package and extract the distribution into local computer directory, for example "C:\Aglets2.0"
2. Open MS-DOS Prompt
3. Change directory to the directory that contains the ASDK script, for example "C:\Aglets2.0\bin"
4. Set Java home, for example "C:\ >set JDK_HOME=C:\JDK1.3.1"
5. Set Aglet home, for example "C:\ > set AGLET_HOME=C:\aglets2.0"
6. Run ANT script that was given in the ASDK distribution package by typing "C:\ >ant"
7. Run ANT script again by typing "C:\ >ant install-home"

For Unix(csh,tcsh)

1. Unzip the ASDK distribution package and extract the distribution into local Unix directory, for example "/usr/local/abubakak/Aglets2.0"
2. Change directory to the directory that contains the ASDK script, for example "/usr/local/abubakak/Aglets2.0/bin"
3. Set Java home, for example "% setenv JDK_HOME /usr/local/abubakak/jdk1.3.1"
4. Set Aglet home, for example "% setenv AGLET_HOME /usr/local/abubakak/Aglets2.0"
5. Run ANT script that was given in the ASDK distribution package by typing "%ant"

6. Run ANT script again by typing "%ant install-home"

For Unix(sh,ksh,bash)

1. Unzip the ASDK distribution package and extract the distribution into local Unix directory, for example "/usr/local/abubakak/Aglets2.0"
2. Change directory to the directory that contains the ASDK script, for example "/usr/local/abubakak/Aglets2.0/bin"
3. Set Java home, for example "%JDK_HOME=/usr/local/abubakak/jdk1.3.1; export JDK_HOME"
4. Set Aglet home, for example "%AGLET_HOME=/usr/local/abubakak/Aglets2.0; export AGLET_HOME"
5. Run ANT script that was given in the ASDK distribution package by typing "%ant"
6. Run ANT script again by typing "%ant install-home"

A.5 Step 5: Start Up The Tahiti Server

To start up the Tahiti Server, execute "agletsd" script, which is included in the ASDK distribution package as the following:

For Windows 95/98/2000,NT, OS/2

```
C:\ > %AGLET_HOME % \bin\agletsd -f sample_aglets.prop
```

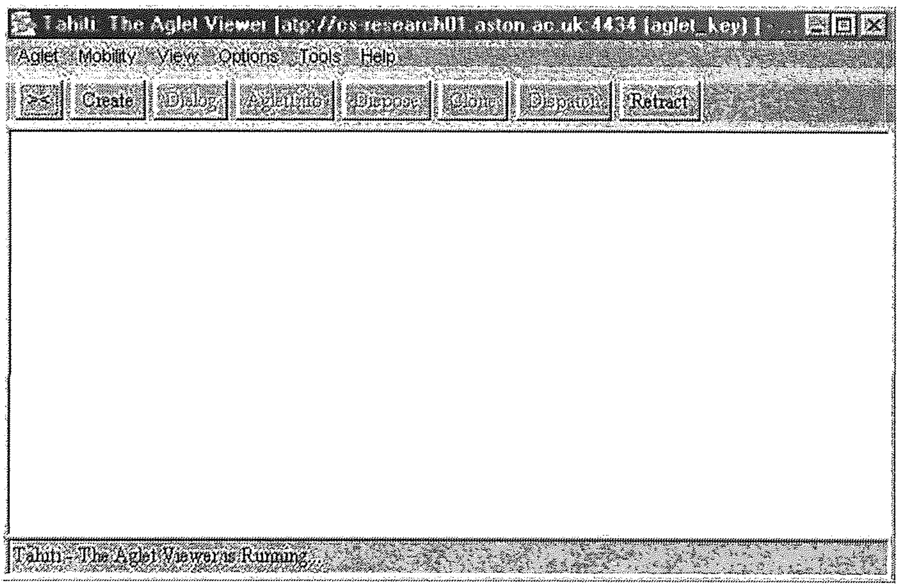
For Unix(csh,tcsh)

```
% $AGLET_HOME/bin/agletsd -f sample_aglets.prop
```

For Unix(sh,ksh,bash)

```
% $AGLET_HOME/bin/agletsd -f sample_aglets.prop
```

Once the Tahiti server has been successfully started up, a Tahiti window will appear as in figure A.1. This Tahiti server provides a user interface for monitoring, creating, dispatching, and disposing of agents and for setting the agent access privileges for the agent server.



Tahiti - The Aglet Viewer is Running...

Figure A.1: A Tahiti Window

Appendix B

SecureFAS User Manual

B.1 Introduction

This manual provides information on how to start up and use the SecureFAS prototype.

B.2 Start Up The SecureFAS Environment

There are three systems that need to be started up before the SecureFAS prototype can be used:

- The Certificate Authority system,
- The Airline system, and
- The SecureFAS prototype system.

B.2.1 The Certificate Authority System

The Certificate Authority system can be started up using the following instructions:

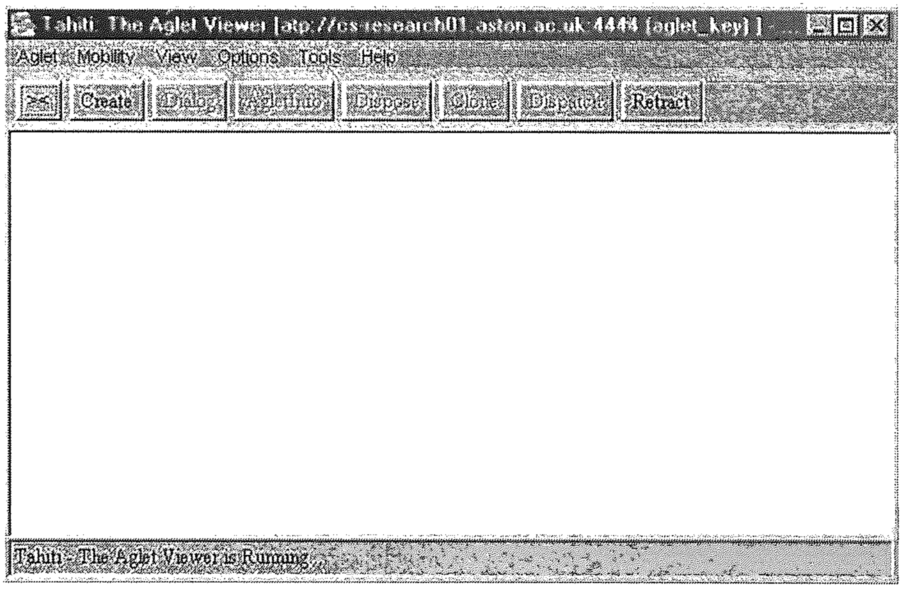
1. Open MS-DOS Prompt
2. Run the Tahiti Server using the following commands:

   2.1 C:\ >cd aglets2.0\bin <enter>

2.2 C:\aglets2.0\bin>agletsd -f my_aglets.props -port 4444 <enter> (Port number can be changed to any number)

3. Start up the Certificate Authority System using the following steps:

3.1 Once the Tahiti Server is running (see Figure B.1), click the **Create** button to start up the Certificate Authority system,



Tahiti - The Aglet Viewer is Running

Figure B.1: A Tahiti Window

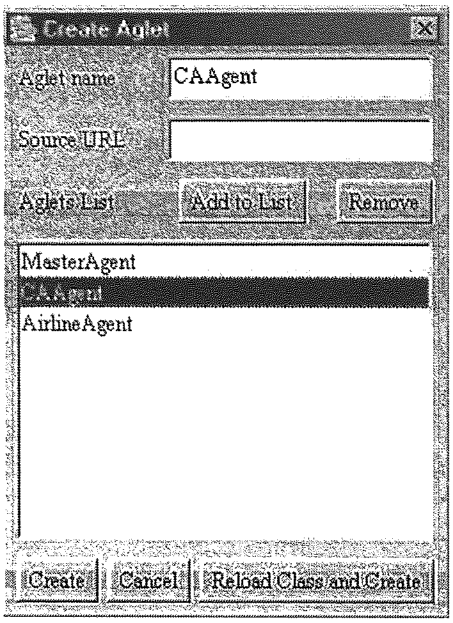
3.2 When a create window appears as in Figure B.2, select CAAgent from the Aglets List and click the **Create** button to start up the Certificate Authority system. Figure B.3 shows the Certificate Authority system is running on the Tahiti Server.

B.2.2 The Airline System

To start up the Airline System, follow the following instructions:

1. Open MS-DOS Prompt (New MS-DOS Prompt should be opened for each Airline System)

2. Run the Tahiti Server using the following steps:

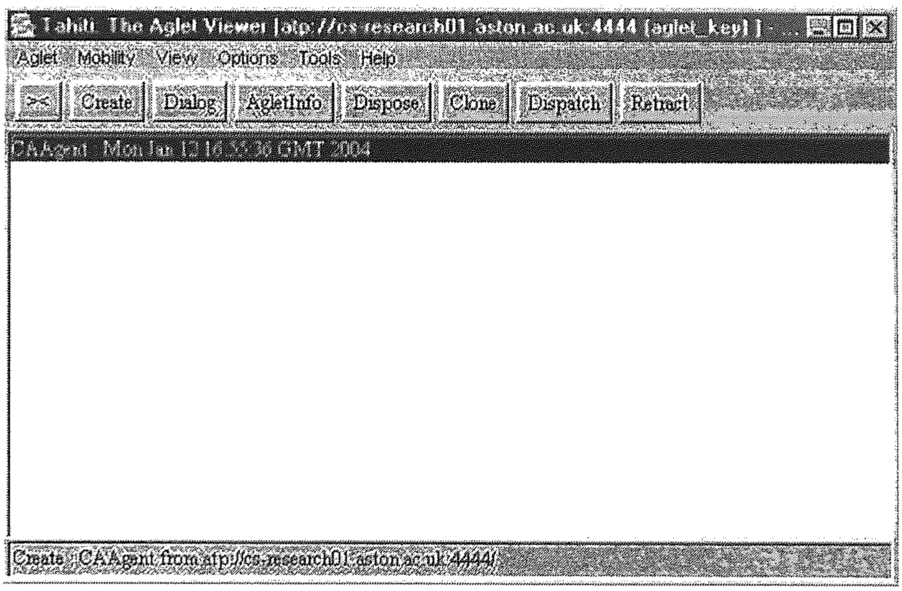


Aglet name: CAAgent

Source URL:

MasterAgent
CAAgent
AirlineAgent

Figure B.2: A Create Window



CAAgent : Mon Jan 12 16:55:36 GMT 2004

Create : CAAgent from atp://cs-research01.aston.ac.uk:4444/

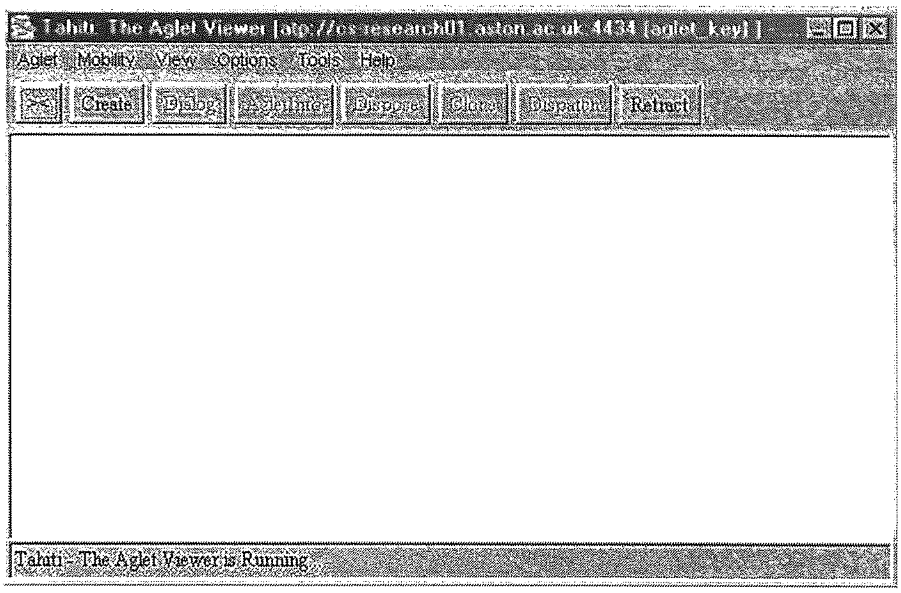
Figure B.3: The Certificate Authority System running on the Tahiti Server

2.1 C:\ >cd aglets2.0\bin <enter>

2.2 C:\aglets2.0\bin>agletsd -f my_aglets.props -port 4434 <enter> (Port number can be changed to any number)

3. Start up the Airline System using the following steps:

3.1 Once the Tahiti Server is running, a Tahiti window will appear as in Figure B.4. The user then needs to click the **Create** button to start up the Airline system.



Tahiti - The Aglet Viewer is Running

Figure B.4: A Tahiti Window

3.2 When a create window appears as in Figure B.5, select AirlineAgent from the Aglets List and click the **Create** button to start up the Airline system. Figure B.6 shows the Airline system is running on the Tahiti Server.

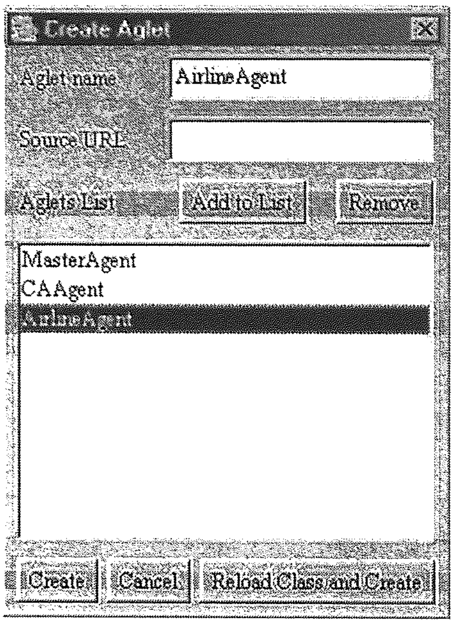
B.2.3 The SecureFAS Prototype System

Once both the Certificate Authority system and the Airline system have been successfully started up, the SecureFAS prototype system then can be started up using the following instruction:

1. Open MS-DOS Prompt (New MS-DOS Prompt should be opened for each Airline System)

2. Run the Tahiti Server using the following steps:

   2.1 C:\ >cd aglets2.0\bin <enter>

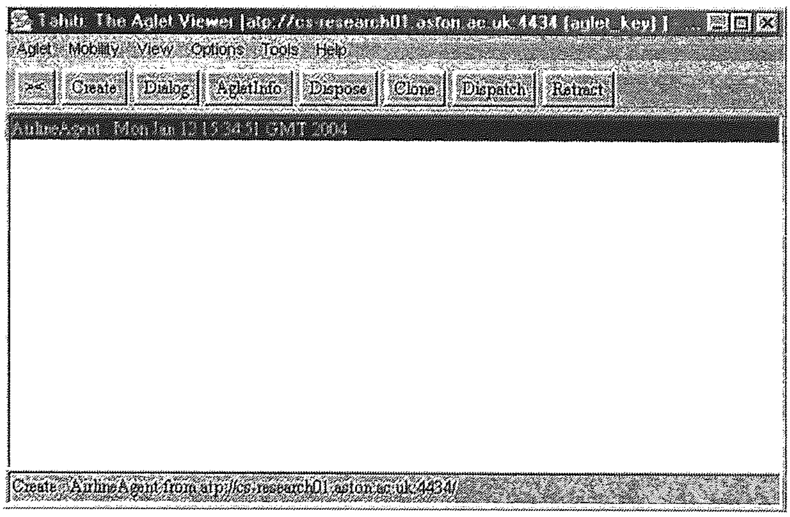


Aglet name: AirlineAgent

Source URL:

MasterAgent
CAAgent
AirlineAgent

Figure B.5: A Create Window



Tahiti: The Aglet Viewer [atp://cs-research01.aston.ac.uk:4434 [aglet_key]]

AirlineAgent Mon Jan 12 15:34:51 GMT 2004

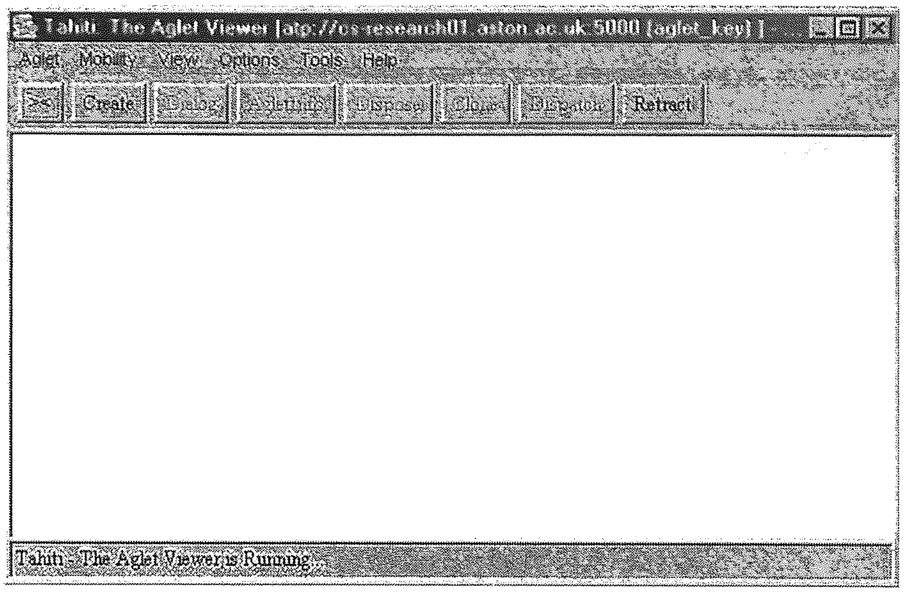
Create : AirlineAgent from atp://cs-research01.aston.ac.uk:4434/

Figure B.6: The Airline System running on the Tahiti Server

2.2 C:\aglets2.0\bin>agletsd -f my_aglets.props -port 5000 <enter> (Port number can be changed to any number)

3. Start up the SecureFAS Prototype System using the following steps:

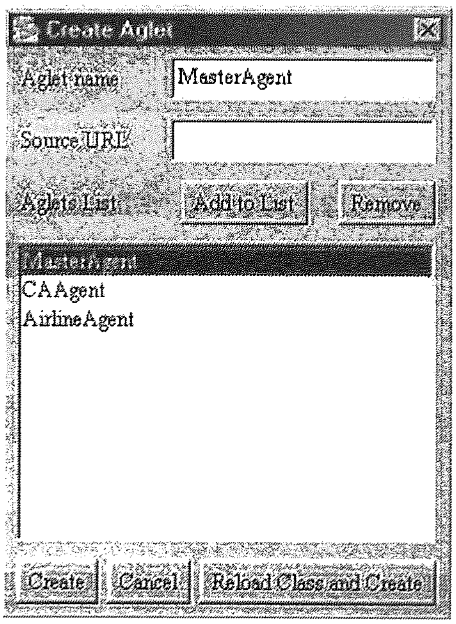
3.1 Once a Tahiti Server window appear as in Figure B.7, click the **Create** button to start up the SecureFAS.



Tahiti - The Aglet Viewer is Running...

Figure B.7: A Tahiti Window

3.2 When a create window appear (see Figure B.8), select MasterAgent from the Aglets List and then click the **Create** button to start up the SecureFAS (see Figure B.9).



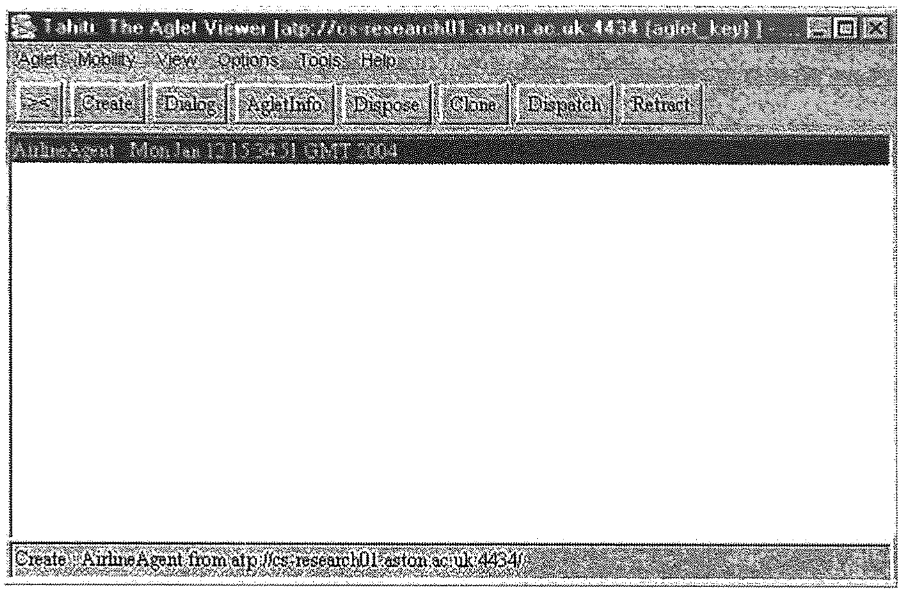
Aglet name: MasterAgent

Source URL:

Aglets List:

MasterAgent
CAAgent
AirlineAgent

Figure B.8: A Create Window


Tahiti: The Aglet Viewer [atp://cs-research01.aston.ac.uk:4434 (aglet_key)]
Aglet Mobility View Options Tools Help
Create Dialog AgletInfo Dispose Clone Dispatch Retract

AirlineAgent : Mon Jan 12 15:34:51 GMT 2004

Create : AirlineAgent from atp://cs-research01.aston.ac.uk:4434/

Figure B.9: The SecureFAS running on the Tahiti Server

B.3 The SecureFAS Operation

Once the SecureFAS has successfully been started up, a new window will appear as in Figure B.10. This new window is a SecureFAS Graphical User Interface (GUI) that functions as an interface between the user and the SecureFAS prototype system to enable user interaction with the SecureFAS prototype system.

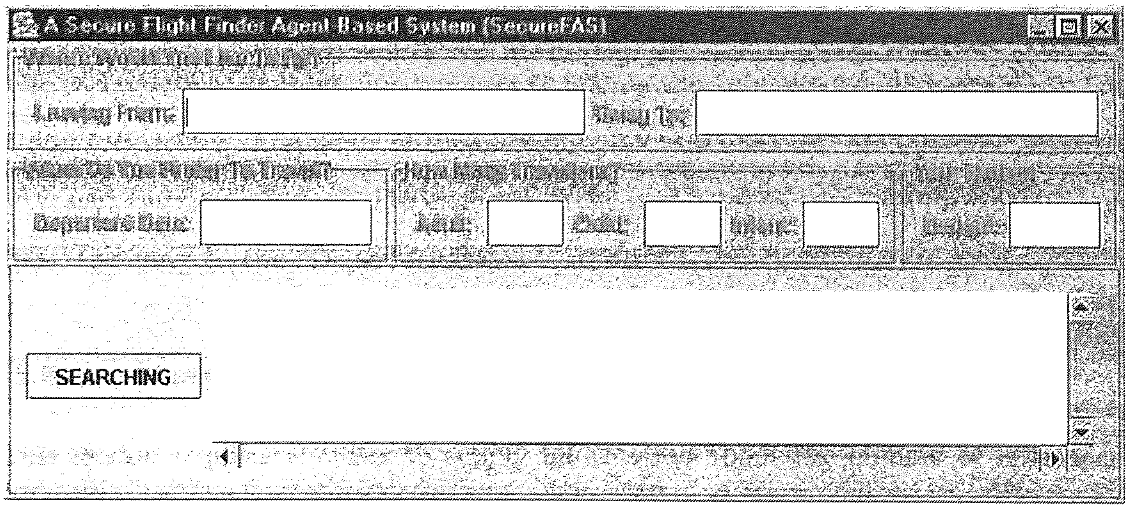


Figure B.10: The SecureFAS GUI

The SecureFAS GUI as in Figure B.10 is divided into six sections:

1. Travel information,
2. Travel date,
3. Passenger information,
4. User budget,
5. Searching button, and
6. SecureFAS result.

B.3.1 Travel Information

This section requires the user to key in information about the airport name where the user starts their travel, and their destination. For example, if the user is going to travel to Kuala Lumpur, Malaysia from Birmingham, United Kingdom, the entry for "Leaving From:" should be "Birmingham" and the entry for "Going To:" is "Kuala Lumpur".

B.3.2 Travel Date

The travel date section requires the user to supply the date of travèl to the SecureFAS prototype system. The input of the date of travel must use the "ddmmyy" format to represent the date of travel. For example, if the user want to travel on the 25 January 2004, the entry to the "Departure Date" is "250104".

B.3.3 Passenger Information

This section requires the user to supply information about the number of passenger who will be travelling according to three different categories: Adult, Child and Infant. For example, if the user wants to travel with his wife and child aged less than two years old, the entry for this section should be "Adult: 2", "Child: 0" and "Infant: 1".

B.3.4 User Budget

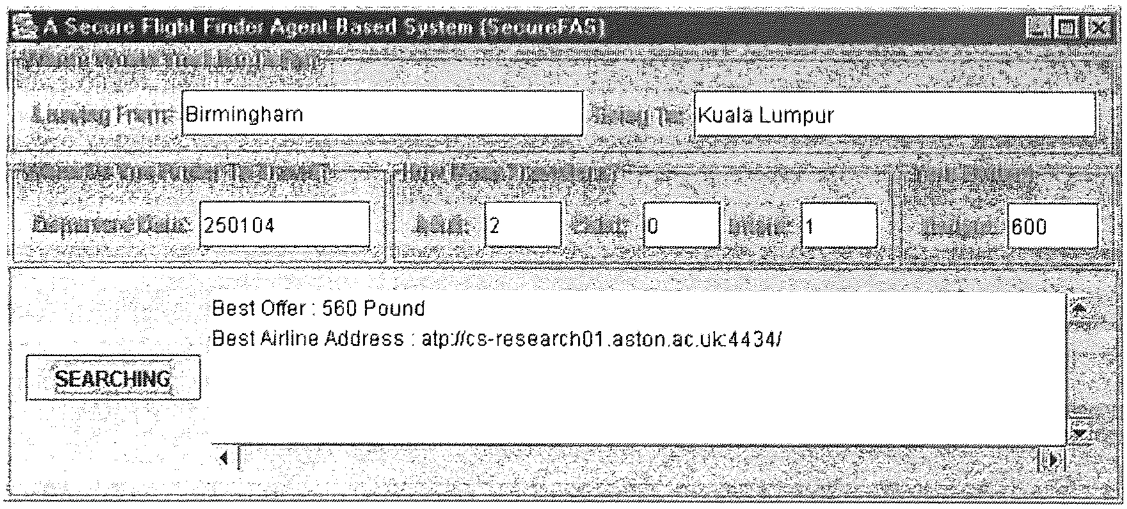
This section requires the user to enter his/her own budget that he/she can afford to pay for the travel. The entry should be any number such as 200, 300, or 500.

B.3.5 Searching Button

This section contains the button that starts the transaction phase of the SecureFAS prototype system. Once the user clicks on this button, the SecureFAS prototype system will start working to find the best offer among the virtual airline servers on the Internet.

B.3.6 SecureFAS Result

This section displays the result from the SecureFAS prototype system that was gathered from the SecureFAS transaction. The results that will be displayed contain the best offer and the address of the virtual airline server that made the best offer. For example, consider the situation where the SecureFAS prototype system only visited three virtual airline servers as below and searched for the best offer under 600 pound.



Leaving From: Birmingham

Going To: Kuala Lumpur

Departure Date: 250104

Adult: 2

Child: 0

Infant: 1

Budget: 600

Best Offer : 560 Pound
Best Airline Address : atp://cs-research01.aston.ac.uk:4434/

Figure B.11: SecureFAS GUI Display Results

1. Airline A (Address: atp://cs-research01.aston.ac.uk:4434/), makes an offer of 560 pound for Business Class,

2. Airline B (Address: atp://cs-research01.aston.ac.uk:4500/), makes an offer of 510 pound for Economic Class, and

3. Airline C (Address: atp://cs-research01.aston.ac.uk:5000/), makes an offer of 499 pound for Economic Class.

In this situation, the SecureFAS will display "560 Pound" and " atp://cs-research01.aston.ac.uk:4434/" as the best offer. Figure B.11 shows the SecureFAS GUI after the transaction is completed.

Appendix C

SecureFAS Use Case and Sequence Diagram

C.1 Introduction

This section describes the design of the SecureFAS prototype system model using the Unified Modeling Language (UML) (*Roff*, 2003; *Muller*, 2000).

C.2 SecureFAS Prototype System Model

The SecureFAS prototype system model shown in Figure C.1 involves three different actors:

- The SecureFAS User,
- The Certificate Authority Administrator, and
- The Virtual Airline Administrator.

The SecureFAS User is an actor that uses the SecureFAS prototype system in order to find the best flight offer among Virtual Airline hosts on the Internet. This actor is responsible to start up the SecureFAS prototype system and supplying the SecureFAS prototype system with its purchase requirement and Virtual Airline addresses to

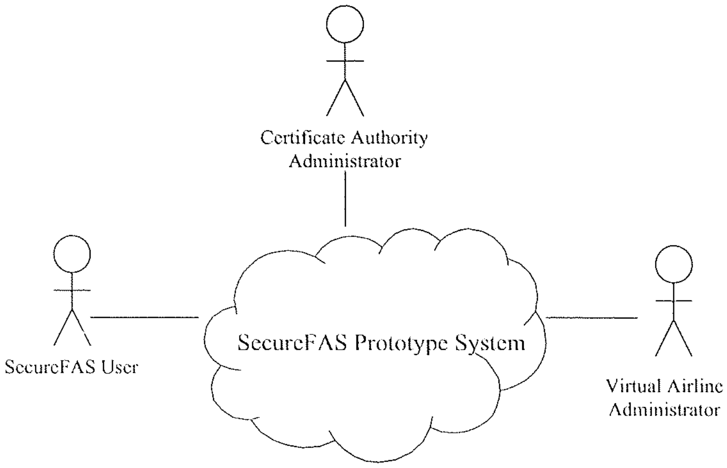

Certificate Authority Administrator
SecureFAS User
SecureFAS Prototype System
Virtual Airline Administrator

Figure C.1: The SecureFAS Prototype System Main Model

enable the SecureFAS prototype system to find the best flight offer. The Certificate Authority Administrator is an actor that is responsible for starting up and managing the Certificate Authority system and host. The Virtual Airline Administrator is an actor responsible for starting up and managing the Virtual Airline system and host.

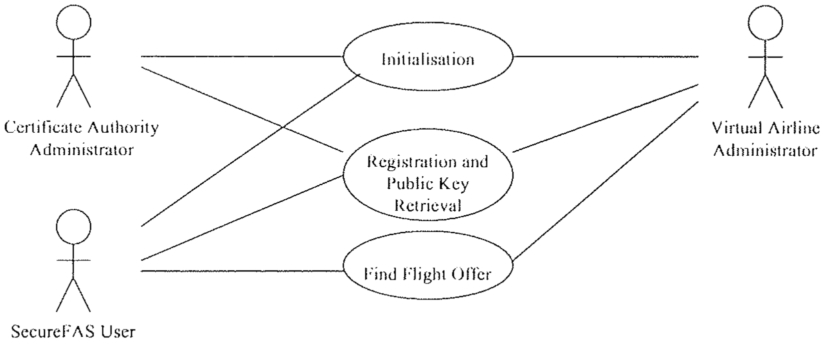

Certificate Authority Administrator
SecureFAS User
Initialisation
Registration and Public Key Retrieval
Find Flight Offer
Virtual Airline Administrator

Figure C.2: Use Case Diagram of the SecureFAS Prototype System

Figure C.2 shows the use case diagram of the SecureFAS prototype system model, which contains three main processes:

- The Initialisation,

- The Registration and Public Key Retrieval, and
- The Find Flight Offer.

C.2.1 Initialisation Process

This section describes in detail the initialisation process of three different systems:

- Certificate Authority System,
- SecureFAS Prototype System, and
- Virtual Airline System.

Certificate Authority System Initialisation Process

Figure C.3 shows the use case diagram of the Certificate Authority System initialisation process. In this figure, the Certificate Authority Administrator is responsible for starting the Certificate Authority System initialisation process.

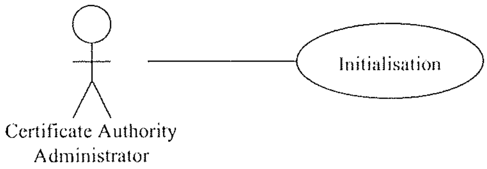


Figure C.3: Use Case Diagram of the Certificate Authority System Initialisation Process

The detail initialisation process of the Certificate Authority system is shown by the sequence diagram in Figure C.4. This figure shows the way the Certificate Authority Administrator starts up the Certificate Authority System.

SecureFAS Prototype System Initialisation Process

The SecureFAS prototype system initialisation process use case diagram is shown in Figure C.5. This figure shows the SecureFAS user executing the initialisation process of the SecureFAS prototype system.

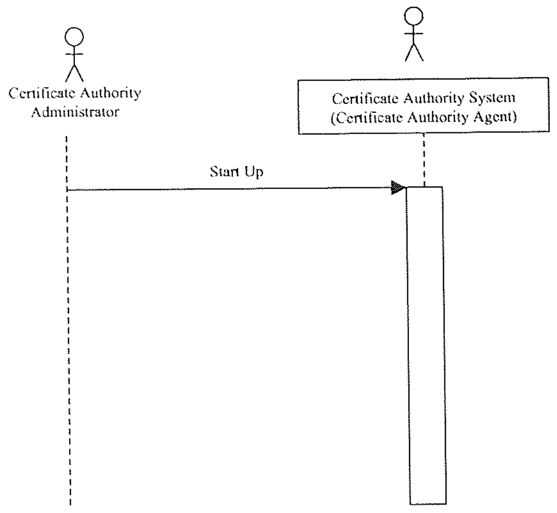

Certificate Authority Administrator
Certificate Authority System (Certificate Authority Agent)
Start Up

Figure C.4: Sequence Diagram of the Certificate Authority System Initialisation Process

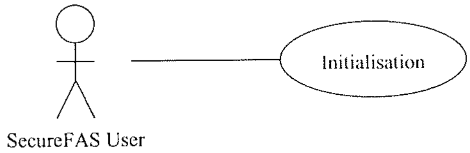


Figure C.5: Use Case Diagram of the SecureFAS Prototype System Initialisation Process

The detail initialisation process of the SecureFAS prototype is shown by the sequence diagram in Figure C.6. This figure shows how the SecureFAS user starts the initialisation process of the SecureFAS prototype system. Once the SecureFAS prototype system is started, the SecureFAS prototype system starts generating cryptographic keys to produce public and private key for the SecureFAS prototype system used.

Virtual Airline System Initialisation Process

Figure C.7 shows the use case of the Virtual Airline System initialisation process. This figure shows the Virtual Airline Administrator executes the initialisation process of the Virtual Airline System.

Detail initialisation process of the Virtual Airline System is shown using sequence

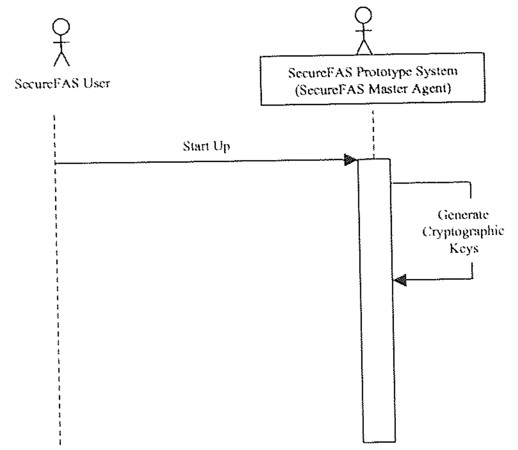

SecureFAS User
SecureFAS Prototype System (SecureFAS Master Agent)
Start Up
Generate Cryptographic Keys

Figure C.6: Sequence Diagram of the SecureFAS Prototype System Initialisation Process

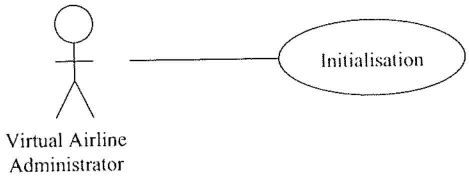


Figure C.7: Use Case Diagram of the Virtual Airline System Initialisation Process

diagram in Figure C.8. In this figure, once the Virtual Airline Administrator starts up the Virtual Airline System, the system starts generating cryptographic keys. This will produce public and private cryptographic key for the Virtual Airline System to enable the Virtual Airline System to securely and privately exchange data through the Internet.

C.2.2 Registration and Public Key Retrieval Process

This section describes the registration and public key retrieval process of two systems:

- SecureFAS Prototype System, and
- Virtual Airline System.

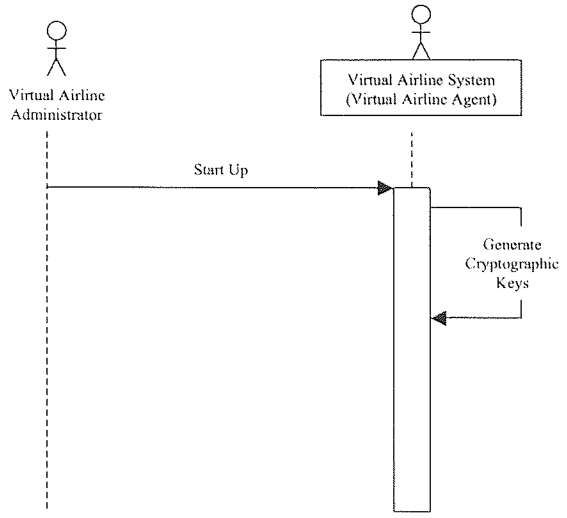

Virtual Airline Administrator
Virtual Airline System (Virtual Airline Agent)
Start Up
Generate Cryptographic Keys

Figure C.8: Sequence Diagram of the Virtual Airline System Initialisation Process

SecureFAS Prototype System Registration and Public Key Retrieval Process

Figure C.9 shows the registration and public key retrieval process of the SecureFAS prototype system using use case diagram.

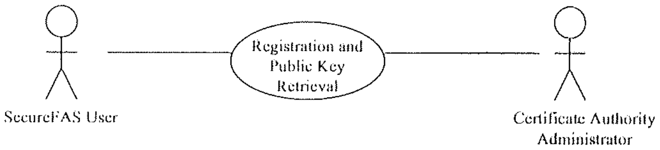

SecureFAS User
Registration and Public Key Retrieval
Certificate Authority Administrator

Figure C.9: Use Case Diagram of the SecureFAS Prototype System Registration and Public Key Retrieval Process

Detail of the registration and public key retrieval process is shown using sequence diagram in Figure C.10. There are two different processes exist in Figure C.10:

- the registration of SecureFAS public key process, and
- the retrieval of the Virtual Airline public key process.

In the registration process, in order to register the SecureFAS public key to the Certificate Authority system, the SecureFAS prototype system will create the Register

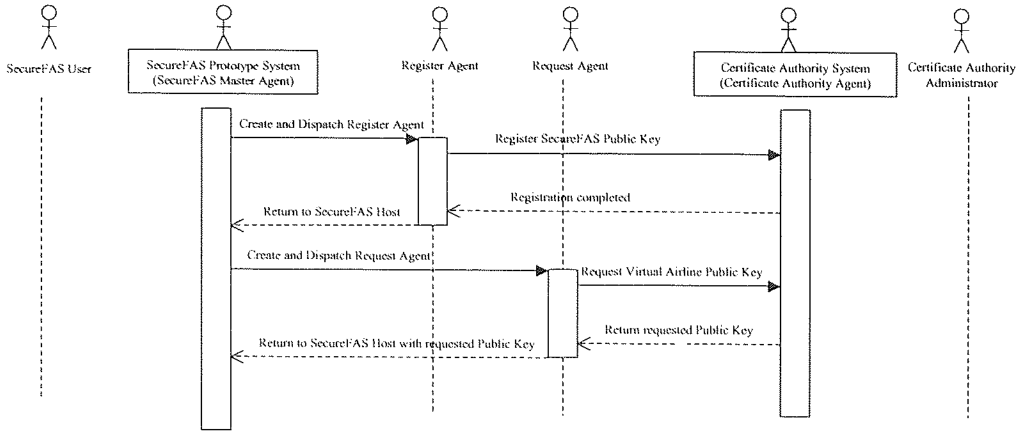

SecureFAS User
SecureFAS Prototype System (SecureFAS Master Agent)
Register Agent
Request Agent
Certificate Authority System (Certificate Authority Agent)
Certificate Authority Administrator
Create and Dispatch Register Agent
Register SecureFAS Public Key
Registration completed
Return to SecureFAS Host
Create and Dispatch Request Agent
Request Virtual Airline Public Key
Return requested Public Key
Return to SecureFAS Host with requested Public Key

Figure C.10: Sequence Diagram of the SecureFAS Prototype System Registration and Public Key Retrieval Process

Agent. Once created, the Register Agent will then be dispatched by the SecureFAS prototype system to the Certificate Authority host to register the SecureFAS public key. Once the registration process is done, the Register Agent returns to the SecureFAS host.

In the retrieval process, the Request Agent will be created by the SecureFAS prototype system in order to retrieve a particular Virtual Airline public key from the Certificate Authority system. This Request Agent will be dispatched by the SecureFAS prototype system to request a particular Virtual Airline public key at the Certificate Authority host. Once a particular Virtual Airline public key received, the Request Agent returns to the SecureFAS host.

Virtual Airline System Registration and Public Key Retrieval Process

Figure C.11 shows the registration and public key retrieval process of the Virtual Airline system using a use case diagram.

Detail of the registration and public key retrieval process for the Virtual Airline system is shown by sequence diagram in Figure C.12. There are also two different processes exist in Figure C.12:

- the registration of Virtual Airline public key process, and

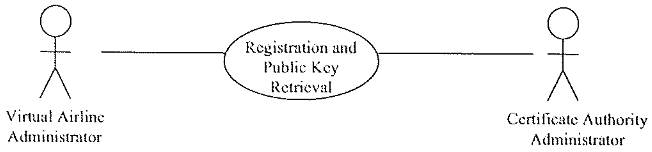

Registration and Public Key Retrieval
Virtual Airline Administrator
Certificate Authority Administrator

Figure C.11: Use Case Diagram of the Virtual Airline System Registration and Public Key Retrieval Process

- the retrieval of the SecureFAS public key process.

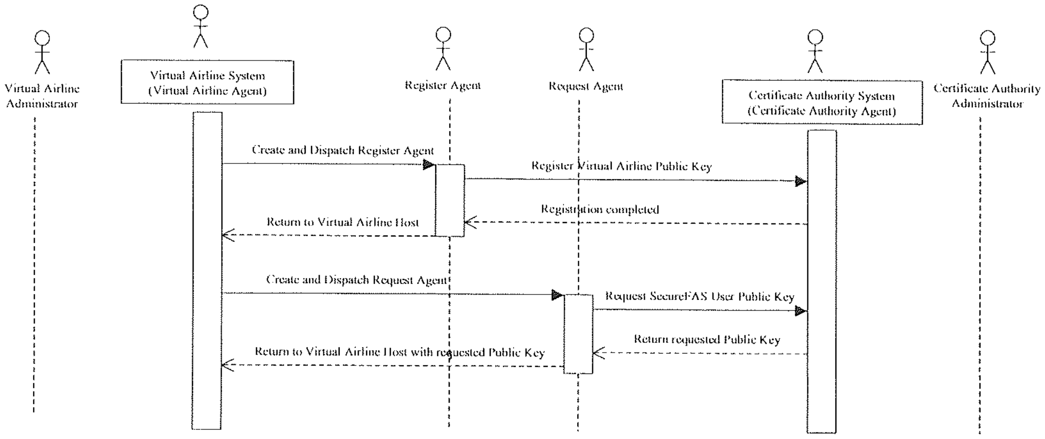

Virtual Airline Administrator
Virtual Airline System (Virtual Airline Agent)
Register Agent
Request Agent
Certificate Authority System (Certificate Authority Agent)
Certificate Authority Administrator
Create and Dispatch Register Agent
Register Virtual Airline Public Key
Registration completed
Return to Virtual Airline Host
Create and Dispatch Request Agent
Request SecureFAS User Public Key
Return requested Public Key
Return to Virtual Airline Host with requested Public Key

Figure C.12: Sequence Diagram of the Virtual Airline System Registration and Public Key Retrieval Process

In the registration process, the Register Agent will be created by the Virtual Airline system. This Register Agent will then be dispatched by the Virtual Airline system to the Certificate Authority host to register the Virtual Airline public key. Once the registration is done, the Register Agent will be returned to the Virtual Airline host.

In the retrieval process, the Request Agent will be created by the Virtual Airline system. This Request Agent will be used to retrieve the SecureFAS public key from the Certificate Authority system by dispatching the Request Agent to the Certificate Authority host. Once the Request Agent received the SecureFAS public key from the Certificate Authority system, the Request Agent returns to the Virtual Airline host.

C.2.3 Find Flight Offer Process

Figure C.13 shows the use case diagram for the SecureFAS prototype system Find Flight Offer process.

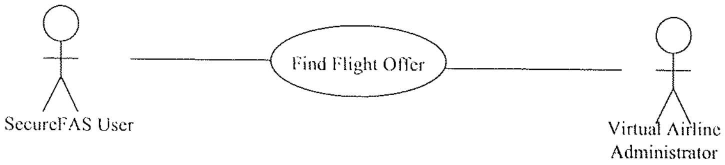


Figure C.13: Use Case Diagram of the SecureFAS Prototype System Find Flight Offer Process

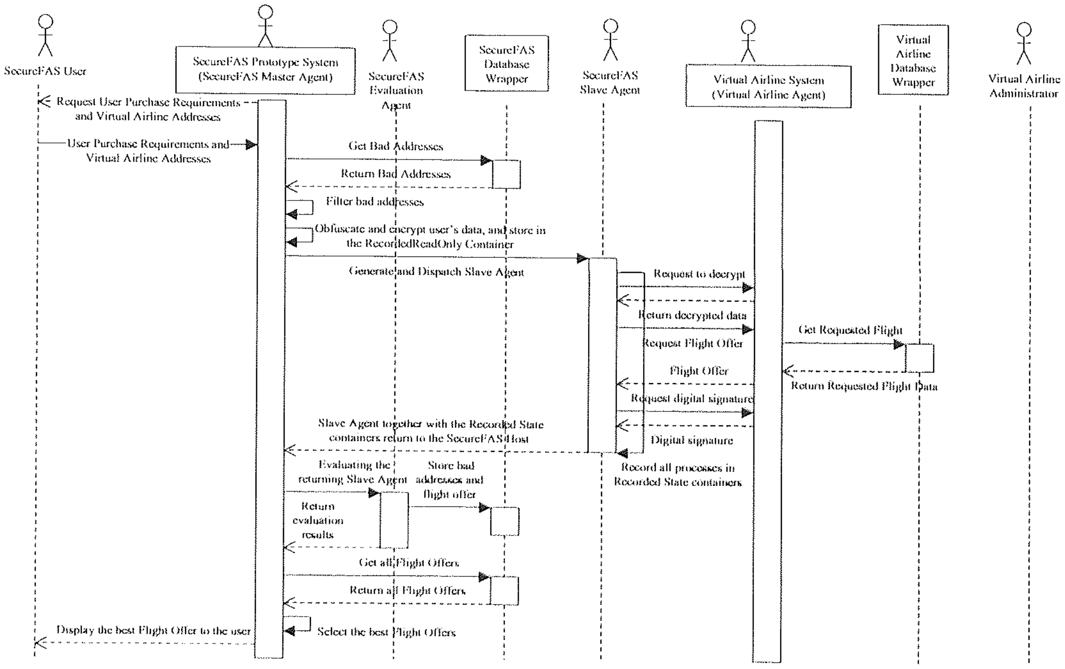


Figure C.14: Sequence Diagram of the SecureFAS Prototype System Find Flight Offer Process

Detail of the Find Flight Offer process is shown using a sequence diagram in Figure C.14. The process of finding flight offer starts when the user of the SecureFAS prototype system supplies the user purchase requirement and virtual airline addresses to the SecureFAS prototype system. The execution of the SecureFAS prototype system then executes the followings steps to find the best flight offer:

1. Retrieve the malicious Virtual Airline addresses (bad addresses) from the SecureFAS database.

2. Filter the malicious Virtual Airline addresses from the user's Virtual Airline addresses.

3. Obfuscate and encrypt user's data, and then store the data into the RecordedReadOnly container.

4. Generate Slave Agent and dispatch it to a particular Virtual Airline system.

5. On arrival at a particular Virtual Airline host, the Slave Agent request the Virtual Airline system to decrypt its data.

6. The Slave Agent then makes a flight offer request based on its user purchase requirement to the Virtual Airline system.

7. The Virtual Airline system checks the Slave Agent request with its database and return the result to the Slave Agent if available.

8. During the Slave Agent execution process inside the Virtual Airline host execution environment, all of the Slave Agent activities are recorded into Recorded State container.

9. Once received flight offer from the Virtual Airline system, the Slave Agent request the Virtual Airline system to digitally signed all the Recorded State container.

10. The Slave Agent return to the SecureFAS host together with the Recorded State container.

11. When the SecureFAS prototype system received the returning Slave Agent, the SecureFAS prototype system starts the evaluation process.

12. During the evaluation process on the returning Slave Agent, all of the containers carried by the Slave Agent will be verified. Any container that cannot be verified will be removed and the address of the Virtual Airline host that the Slave Agent

executed in will be stored in SecureFAS bad addresses database. On the other hand, the SecureFAS prototype system will store the flight offer gathered into the SecureFAS result database.

13. Once all the Slave Agent returned to the SecureFAS prototype system, the SecureFAS prototype system starts the process of selecting the best flight offer from its database.

14. The best flight offer found will then be displayed to the SecureFAS user for further actions.

Appendix D

Listings

D.1 SecureFAS Prototype System (MasterAgent.java)

```
import com.ibm.aglet.*;
import java.net.URL;
import java.util.*;
import java.text.*;
import java.io.*;
import java.lang.reflect.*;
import java.util.Random;
import java.sql.*;
import java.util.Vector;
import java.util.Enumeration;
import java.util.List;
import iaik.security.provider.IAIK;
import java.security.*;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.Cipher;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


public class MasterAgent extends Aglet {
    private AgletProxy Master_AgentProxy;
    private AgletProxy Slave_AgentProxy;
    private AgletInfo Master_AgentInfo;
    Vector serverAddress = new Vector();
    Vector readOnly = new Vector();
    Vector executeOnly = new Vector();
    Vector collectOnly = new Vector();
    SecretKey secKey = null;
    PrivateKey privKey = null;
    PublicKey pubKey = null;
    String[] userPurchaseReq = new String[10];
    List SAdetails = new ArrayList();
    List PubKeyList = new ArrayList();
    long startTime;
    long stopTime;
long intervalTime;
JFrame MenuFrame;
JLabel MainLabel;
JLabel Quest1Label, Quest2Label, Quest3Label, Quest4Label, Quest5Label, Quest6Label, Quest7Label;
JPanel panel1, panel2, panel3, panel4, panel5;
JTextField tempQuest1, tempQuest2, tempQuest3, tempQuest4, tempQuest5, tempQuest6, tempQuest7;
JTextArea textArea;
Random RN;
double UserBudget;
double ObfuscateValue1=0, ObfuscateValue2=0, ObfuscateValue3=0;
long tempTime;
String timeStr;
String Address;
String AirlineAddress="";        int index=0;
int indexIN=0;
int indexFail=0;
int actionStatus=0;
double bestOffer=99999999;
int RandN;
String StatusProcess;
String fromAddress;
public static final int MAXLEVEL = 3;
int NoiseNo=0;
static final String DB = “jdbc:odbc:BlacklistAddress”;
static final String USER = “”;
static final String PASSWORD = “”;
Connection conSelAdd;
static final String DB_Result = “jdbc:odbc:Result”;
static final String USER_Result = “”;
static final String PASSWORD_Result = “”;
Connection conResult;


//Constructor MasterAgent
public MasterAgent() {
    Security.insertProviderAt(new IAIK(), 2);
}


// Method onCreation
public void onCreation(Object args) {
    Master_AgentInfo = getAgletInfo();
    Master_AgentProxy = getAgletContext().getAgletProxy(Master_AgentInfo.getAgletID());
    Address = Master_AgentInfo.getOrigin().toString();

    try{
        Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
        conSelAdd = DriverManager.getConnection(DB, USER, PASSWORD);
    } catch (Exception e) {
    e.printStackTrace();
    }

    try{
        Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
        conResult = DriverManager.getConnection(DB_Result, USER_Result, PASSWORD);
    } catch (Exception e) {
    e.printStackTrace();
    }
}
```

```
// Method onDisposing
public void onDisposing() {
}


// Method run
public void run() {
      URL servAddr;
      URL remoteAddress;
      index=0;

      // Set the look and feel.
      try {
            UIManager.setLookAndFeel(
            UIManager.getCrossPlatformLookAndFeelClassName());
      } catch(Exception e) { }

      MasterAgent Menu = new MasterAgent();

      MenuFrame = new JFrame("A Secure Flight Finder Agent-Based System (SecureFAS)");

      // Method create the frame and container
      panel1 = new JPanel();
      panel1.setLayout(new BoxLayout(panel1, BoxLayout.X_AXIS));

      // Add component to panel 1
      addcomponent1();

      // Add the panel to the frame.
      MenuFrame.getContentPane().add(panel1, BorderLayout.NORTH);

      panel2 = new JPanel();
      panel2.setLayout(new GridLayout(1,1));

      //Add component to panel 2
      addcomponent2();

      // Add the panel to the frame.
      MenuFrame.getContentPane().add(panel2, BorderLayout.WEST);

      panel3 = new JPanel();
      panel3.setLayout(new GridLayout(1,1));

      //Add component to panel 3
      addcomponent3();

      // Add the panel to the frame.
      MenuFrame.getContentPane().add(panel3, BorderLayout.CENTER);

      panel4 = new JPanel();
      panel4.setLayout(new GridLayout(1,1));

      //Add component to panel 4
      addcomponent4();

      // Add the panel to the frame.
      MenuFrame.getContentPane().add(panel4, BorderLayout.EAST);

      panel5 = new JPanel();
```

```
panel5.setLayout(new BoxLayout(panel5, BoxLayout.X_AXIS));

//Add component to panel 5
addcomponent5();

// Add the panel to the frame.
MenuFrame.getContentPane().add(panel5, BorderLayout.SOUTH);

// Exit when the window is closed.
MenuFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Show the Menu.
MenuFrame.pack();
MenuFrame.setVisible(true);

// Wait until the user finished enter all the input
while (actionStatus == 0) {
} ;

generateKeys(); //generate crypto keys

//Register server address and public key to CA server
try {
      URL CAServer = new URL(“atp://cs-research01.aston.ac.uk:4444/”);
      Object arg[]=new Object[] { Address,pubKey} ;
      AgletProxy RegisterAgentProxy = getAgletContext().createAglet(getCodeBase(), “RegisterAgent”, arg);
      RegisterAgentProxy.dispatch(CAServer);
} catch(Exception e) { ;}

//Store Airline Address
try {
      serverAddress.addElement(new URL(“atp://cs-research01.aston.ac.uk:4434/”));
      serverAddress.addElement(new URL(“atp://cs-research01.aston.ac.uk:4454/”));
      serverAddress.addElement(new URL(“atp://cs-research01.aston.ac.uk:4464/”));
} catch (Exception e) { ;}

//Removed blacklisted address
Iterator i=serverAddress.iterator();
while (i.hasNext()) {
      try {
            remoteAddress = (URL)i.next();
            String querySel = “Select servAdd From serverAddress ” +
                                    “Where servAdd=’” + remoteAddress + “’”;

            Statement stmtSelAdd = conSelAdd.createStatement();
            ResultSet rsSelAdd = stmtSelAdd.executeQuery(querySel);

            if(rsSelAdd.next()) {
                  i.remove(); // removed bad server address
            }

            conSelAdd.close();
            stmtSelAdd.close();
      } catch (Exception e) { ;}
}

//Request remote servers Public Key from CA Server
Iterator goodAddress=serverAddress.iterator();
while(goodAddress.hasNext()) {
      try {
            URL Add = (URL)goodAddress.next();
            String SerAdd = Add.toString();
            URL CAServer = new URL("atp://cs-research01.aston.ac.uk:4444/");
            Object argPK[]=new Object[] { SerAdd,Master_AgentProxy} ;
            AgletProxy RequestAgentProxy = getAgletContext().createAglet(getCodeBase(),"RequestAgent", argPK);
            RequestAgentProxy.dispatch(CAServer);
        } catch (Exception e) { ;}
    }

    UserBudget= Double.parseDouble(tempQuest7.getText()); // user budget

    long systemTime = System.currentTimeMillis();
    String timeSt = new Long(systemTime).toString();
    RandN = Integer.parseInt(timeSt.substring(9,11));

    while (RandN == 0) {
        systemTime = System.currentTimeMillis();
        timeSt = new Long(systemTime).toString();
        RandN = Integer.parseInt(timeSt.substring(9,11));
    }

    ObfuscateValue1=rs3(UserBudget,RandN); // true obfuscated value
    ObfuscateValue2=rs3((UserBudget-RandN*10),RandN); // fake obfuscated value
    ObfuscateValue3=rs3((UserBudget+RandN*10),RandN); // fake obfuscated value

    userPurchaseReq[0]= tempQuest1.getText(); //origin
    userPurchaseReq[1]= tempQuest2.getText(); //destination
    userPurchaseReq[2]= tempQuest3.getText(); //depature date
    userPurchaseReq[3]= tempQuest4.getText(); //passanger type - Adult
    userPurchaseReq[4]= tempQuest5.getText(); //passanger type - Child
    userPurchaseReq[5]= tempQuest6.getText(); //passanger type - Infant
    userPurchaseReq[6]=new Double(ObfuscateValue1).toString(); //obfuscated user budget value
    userPurchaseReq[7]= new Double(ObfuscateValue2).toString(); //fake obfuscated value (Noise code)
    userPurchaseReq[8]= new Double(ObfuscateValue3).toString(); //fake obfuscated value (Noise code)
    userPurchaseReq[9]=new Integer(RandN).toString(); //RS3 First Random Number

    byte[] digitalSig = genSignature(privKey, userPurchaseReq); //Generate digital signature

    readOnly.addElement(RSAencrypt(privKey,userPurchaseReq)); //insert user's specific purchase requirement

    startTime = System.currentTimeMillis(); //Timed the Slave Agent process

    //Dispatch Slave Agent to remote server
    Enumeration enum = serverAddress.elements();
    while (enum.hasMoreElements()) {
        try {
            Object args[] = new Object[] { Master_AgentProxy,pubKey,readOnly,digitalSig} ;
            Slave_AgentProxy = getAgletContext().createAglet(getCodeBase(),"SlaveAgent", args);
            servAddr = (URL)enum.nextElement();
            Slave_AgentProxy.dispatch(servAddr);
            AgletInfo SlaveAgentInfo = Slave_AgentProxy.getAgletInfo();
            SAdetails.add(SlaveAgentInfo.getAgletID().toString());
            SAdetails.add(Master_AgentInfo.getOrigin().toString());
            SAdetails.add(servAddr.toString());
            SAdetails.add(new java.util.Date(SlaveAgentInfo.getCreationTime()).toString());
            index++;
        } catch (Exception e) { ;}
    }
}
```

```
// Method message handling
public boolean handleMessage(Message msg) {
    if (msg.sameKind(“PublicKey”)) {
        String recvSerAdd = (String)msg.getArg(“argServerAddress”);
        PublicKey recvPubKey = (PublicKey)msg.getArg(“argPublicKey”);

        PubKeyList.add(recvSerAdd);
        PubKeyList.add(recvPubKey);
        return true;
    } else
    if (msg.sameKind(“Process Fail”)) {
        NoiseNo = ((Integer)msg.getArg(“argNoiseNo”)).intValue();
        Vector recvES = (Vector)msg.getArg(“argES”);
        Vector recvCS = (Vector)msg.getArg(“argCS”);

        if(NoiseNo==1) {
            indexIN++;
            indexFail++;
            Enumeration ES = recvES.elements();
            while(ES.hasMoreElements()) {
                String ESData=ES.nextElement().toString();
                if (ESData.equals(“XFL”))
                    StatusProcess=“XFL”;
                if (ESData.equals(“XDT”))
                    StatusProcess=“XDT”;
                if (ESData.equals(“XF”))
                    StatusProcess=“XF”;
            }

            if((indexIN==index) && (indexFail==index)) {
                if (StatusProcess.equals(“XFL”))
                    textArea.append(“Requested Flight Is Not Available”);
                if (StatusProcess.equals(“XDT”))
                    textArea.append(“Requested Flight Date Is Not Available”);
                if (StatusProcess.equals(“XF”))
                    textArea.append(“Requested Flight Fare Is Not Available”);
            } else
            if (indexIN==index) {
                try {
                    String queryResult = “Select * From ResultData”;
                    Statement stmtResult = conResult.createStatement();
                    ResultSet rsResult = stmtResult.executeQuery(queryResult);

                    while (rsResult.next()) {
                        String slaveid = rsResult.getString(1);
                        fromAddress = rsResult.getString(2);
                        double bOffer = Double.parseDouble(rsResult.getString(3));

                        if(bestOffer > bOffer) {
                            bestOffer=bOffer;
                            AirlineAddress=fromAddress;
                        }
                    }
                    rsResult.close();
                    stmtResult.close();
                } catch (Exception e) { ;}
                textArea.append(“Best Offer : ” + bestOffer);
                textArea.append(“Best Airline Address : ” + AirlineAddress);
            }
        }
    return true;
} else
if (msg.sameKind(“Recorded State”)) {
    stopTime = System.currentTimeMillis();
    intervalTime = (stopTime - startTime) /1000;

    if(intervalTime < 60) {
        NoiseNo = ((Integer)msg.getArg(“argNoiseNo”)).intValue();
        String[] recvES = (String[])msg.getArg(“argES”);
        byte[] recvES_Sig = (byte[])msg.getArg(“argES_Sig”);
        String[] recvCS = (String[])msg.getArg(“argCS”);
        byte[] recvCS_Sig = (byte[])msg.getArg(“argCS_Sig”);

        if(NoiseNo==1) {
            try {
                //Master Agent creates Evaluation Agent
                Object args[]=new Object[] { Master_AgentProxy, recvES, recvES_Sig, recvCS, recvCS_Sig,
                                             userPurchaseReq, SAdetails, privKey, PubKeyList,
                                             new Double(UserBudget).toString()} ;
                AgletProxy EvaluationAgentProxy = getAgletContext().createAglet(getCodeBase(),
                                                  “EvaluationAgent”, args);
            } catch(Exception e) { ;}
        }
    } else {
    try {
        String queryResult = “Select * From ResultData”;

        Statement stmtResult = conResult.createStatement();
        ResultSet rsResult = stmtResult.executeQuery(queryResult);

        while (rsResult.next()) {
            String slaveid = rsResult.getString(1);
            fromAddress = rsResult.getString(2);
            double bOffer = Double.parseDouble(rsResult.getString(3));

            if(bestOffer > bOffer) {
                bestOffer=bOffer;
                AirlineAddress=fromAddress;
            }
        }
        rsResult.close();
        stmtResult.close();
    } catch (Exception e) { ;}
    textArea.append(“Best Offer : ” + bestOffer);
    textArea.append(“Best Airline Address : ” + AirlineAddress);
    }
    return true;
}
else
if (msg.sameKind(“Finished”)) {
    indexIN++;

    if(indexIN==index) {
        try {
            String queryResult = “Select * From ResultData”;

            Statement stmtResult = conResult.createStatement();
            ResultSet rsResult = stmtResult.executeQuery(queryResult);

            while (rsResult.next()) {
```

```
                        String slaveid = rsResult.getString(1);
                        fromAddress = rsResult.getString(2);
                        double bOffer = Double.parseDouble(rsResult.getString(3));

                        if(bestOffer > bOffer) {
                            bestOffer=bOffer;
                            AirlineAddress=fromAddress;
                        }
                    }
                    rsResult.close();
                    stmtResult.close();
                } catch (Exception e) { ;}
            }
            textArea.append(“Best Offer : ” + bestOffer);
            textArea.append(“Best Airline Address : ” + AirlineAddress);
            return true;
        } else
        return false;
    }


    //Method create and add component to Panel 1
    private void addcomponent1() {
        Quest1Label = new JLabel(“Leaving From:”);
        tempQuest1 = new JTextField(5);
        Quest2Label = new JLabel(“Going To:”);
        tempQuest2 = new JTextField(5);

        // Add component to panel
        panel1.add(Quest1Label);
        panel1.add(tempQuest1);
        panel1.add(Quest2Label);
        panel1.add(tempQuest2);

        Quest1Label.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
        Quest2Label.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

        panel1.setBorder(BorderFactory.createCompoundBorder(
        BorderFactory.createTitledBorder(“Where Would You Like To Fly?”),
        BorderFactory.createEmptyBorder(5,5,5,5)));
    }

    // Method create and add component to Panel 2
    private void addcomponent2() {
        Quest3Label = new JLabel(“Departure Date:”);
        tempQuest3 = new JTextField(5);

        // Add component to panel
        panel2.add(Quest3Label);
        panel2.add(tempQuest3);

        Quest3Label.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

        panel2.setBorder(BorderFactory.createCompoundBorder(
        BorderFactory.createTitledBorder(“When Do You Prefer To Travel?”),
        BorderFactory.createEmptyBorder(5,5,5,5)));
    }


    // Method create and add component to Panel 3
```

```
private void addcomponent3() {
      Quest4Label = new JLabel("Adult:");
      tempQuest4 = new JTextField(2);
      Quest5Label = new JLabel("Child:");
      tempQuest5 = new JTextField(2);
      Quest6Label = new JLabel("Infant:");
      tempQuest6 = new JTextField(2);

      // Add component to Panel 3
      panel3.add(Quest4Label);
      panel3.add(tempQuest4);

      panel3.add(Quest5Label);
      panel3.add(tempQuest5);

      panel3.add(Quest6Label);
      panel3.add(tempQuest6);

      Quest4Label.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
      Quest5Label.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
      Quest6Label.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

      panel3.setBorder(BorderFactory.createCompoundBorder(
      BorderFactory.createTitledBorder("How Many Travellers?"),
      BorderFactory.createEmptyBorder(5,5,5,5)));
}


// Method create and add component to Panel 4
private void addcomponent4() {
      Quest7Label = new JLabel("Budget:");
      tempQuest7 = new JTextField(3);

      // Add component to Panel 4
      panel4.add(Quest7Label);
      panel4.add(tempQuest7);

      Quest7Label.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

      panel4.setBorder(BorderFactory.createCompoundBorder(
      BorderFactory.createTitledBorder("Your Budget"),
      BorderFactory.createEmptyBorder(5,5,5,5)));
}


// Method create and add component to Panel 5
private void addcomponent5() {
      JButton Button1 = new JButton("SEARCHING");
      Button1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                  actionStatus=1;
            }
      } );

      textArea = new JTextArea(5, 20);
      textArea.setEditable(false);
      JScrollPane scrollPane = new JScrollPane(textArea,
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
```

```
    scrollPane.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

    // Add component to Panel 5
    panel5.add(Button1);
    panel5.add(scrollPane);

    panel5.setBorder(BorderFactory.createCompoundBorder(
    BorderFactory.createTitledBorder(“”),
    BorderFactory.createEmptyBorder(5,5,5,5)));
}


// Method generate crypto keys
public void generateKeys() {
    try {
        KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance(“RSA”,“IAIK”);

        keyPairGen.initialize(1024);

        KeyPair pair = keyPairGen.generateKeyPair();
        privKey = pair.getPrivate();
        pubKey = pair.getPublic();
    } catch (Exception e) {
    System.out.println(“[Generating RSA-PKI keypair error]” + e.toString());
    }
}


// Method RSA encryption
public byte[][] RSAencrypt(PrivateKey pvKey, String[] text) {
    byte[][] cipherText=new byte[java.lang.reflect.Array.getLength(text)][];
    byte[] textByte=null;
    String tempStr = “”;

    try {
        Cipher cipher = Cipher.getInstance(“RSA”,“IAIK”);
        cipher.init(Cipher.ENCRYPT_MODE,pvKey);
        for(int i=0; i<java.lang.reflect.Array.getLength(text); i++) {
            tempStr = addValidStr(text[i]);
            textByte = tempStr.getBytes();
            cipherText[i]=cipher.doFinal(textByte);
        }
    } catch(Exception e) {
    System.out.println(“[PKI encryption error] ” + e.toString());
    }
    return cipherText;
}


// Method RSA decryption
public String[] RSAdecrypt(PublicKey pbKey, byte[][] cipherText) {
    String[] plainText = new String[java.lang.reflect.Array.getLength(cipherText)];
    byte[] cipherByte=null;
    String tempStr = “”;

    try {
        Cipher cipher = Cipher.getInstance(“RSA”,“IAIK”);
        cipher.init(Cipher.DECRYPT_MODE,pbKey);
        for (int i=0; i<java.lang.reflect.Array.getLength(cipherText); i++) {
            cipherByte = cipher.doFinal(cipherText[i]);
```

```
                tempStr = new String(cipherByte);
                plainText[i] = cutValidStr(tempStr);
            }
        } catch(Exception e) {
        System.out.println(“[PKI decryption error] ” + e.toString());
        }
        return plainText;
    }


    // Method add string
    public String addValidStr(String rawStr){
        int chrAdded,i,strLength;

        if (rawStr == null)
            rawStr = “”;

        strLength = rawStr.length()%8;
        chrAdded = 8-strLength;

        if (strLength > 0){
            for (i=0; i<chrAdded; i++)
                rawStr = rawStr + “X”;
            rawStr = rawStr + “ADDCHAR” + String.valueOf(chrAdded);
        } else{
        if (rawStr.length()!=0)
            rawStr = rawStr + String.valueOf(strLength);
        else{
            rawStr = “DATANUL0”;
            }
        }
        return rawStr;
    }


    // Method cut string
    public String cutValidStr(String rawStr){
        String tempStr = “”;
        int i = Integer.valueOf(String.valueOf(rawStr.charAt((rawStr.length())-1))).intValue();

        if (rawStr.length()!=8)
            tempStr = rawStr.substring(0,rawStr.length()-(i+8));
        return tempStr;
    }


    // Method generate digital signature
    public byte[] genSignature(PrivateKey privKey, String[] plainText) {
        byte[] sig=null;
        byte[] cipherText;

        try {
            Signature genSig = Signature.getInstance(“SHA1withRSA”);
            genSig.initSign(privKey);
            try {
                for (int i=0; i<java.lang.reflect.Array.getLength(plainText); i++) {
                    cipherText = plainText[i].getBytes();
                    genSig.update(cipherText); //add msg to be sign
                }
            } catch(Exception e){
```

```
            System.out.println(“[Updating signature error] ” + e.toString());
            }
            sig = genSig.sign();
        } catch(Exception e){
        System.out.println(“[Generating signature error] ” + e.toString());
        }
        return sig;
    }


    // Method verify digital signature
    public void verifySignature(PublicKey pubKey, byte[] inSig, String[] plainText) {
        boolean verifies=false;
        byte[] inCipher;

        try {
            Signature verifySig = Signature.getInstance(“SHA1withRSA”);
            verifySig.initVerify(pubKey);
            for (int i=0; i<java.lang.reflect.Array.getLength(plainText); i++) {
                inCipher = plainText[i].getBytes();
                verifySig.update(inCipher); // add msg that need to be verified
            }
            verifies=verifySig.verify(inSig);
        } catch(Exception e){
        System.out.println(“[Verifying signature error] ” + e.toString());
        }
    }


    //Method polynomial calculation
    public double polynomial(int a, double b, int c, int x) {
        double result;

        result=(a * Math.pow(x,2)) + (b * x) + c;
        return result;
    }


    // Method Random Sequence 3-level obfuscation algorithm

    public double rs3(double budget,int randomNumber) {
        int branch;
        int newRnd = randomNumber;
        for (int i=0; i<MAXLEVEL; i++) {
            branch = newRnd % MAXLEVEL;
            Random selector = new Random(branch);
            newRnd = (int) (selector.nextDouble() * 1000000);
            budget = polynomial(branch, budget, (newRnd % randomNumber), randomNumber);
        }
        return budget;
    }

}
```

D.2 SecureFAS Slave Agent (SlaveAgent.java)

```
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.URL;
import java.io.*;
import java.util.*;
import java.security.*;
import javax.crypto.Cipher;
import iaik.security.provider.IAIK;
import java.lang.reflect.*;
import java.util.Random;


public class SlaveAgent extends Aglet {
    private AgletProxy Master_AgentProxy;
    private AgletInfo Slave_AgentInfo;
    private AgletProxy Slave_AgentProxy;
    Vector readOnly = new Vector();
    Vector executeOnly = new Vector();
    Vector collectOnly = new Vector();
    PublicKey MApubKey=null;
    byte[] digitalSig;
    AgletProxy proxyAirlineAgent;
    boolean verifies=false;
    String[] userPurchaseReq = new String[10];
    Vector RO_Con = new Vector();
    int xMatchCounter=0;
    int rqstFareCount=0;
    int recvFareCount=0;
    double totalFare=0;
    String classType=“”;
    String recvFlightNo=“”;
    int recvDeparture=0;
    int recvArrival=0;
    boolean searchFlight=true;
    int passanger=0,NoiseCount=0,SlaveCount=0;
    String[] exeState = new String[27];
    String[] colState = new String[14];
    Random RN;
    public static final int MAXLEVEL = 3;


    // Method onCreation
    public void onCreation(Object args) {
        Object init[] = (Object[])args;
        Master_AgentProxy = (AgletProxy)init[0];
        MApubKey = (PublicKey)init[1];
        readOnly = (Vector)init[2];
        digitalSig = (byte[])init[3];

        Slave_AgentInfo = getAgletInfo();
        Slave_AgentProxy = getAgletContext().getAgletProxy(Slave_AgentInfo.getAgletID());

        try{
            executeOnly.addElement(“ET ” + new Date(Slave_AgentInfo.getCreationTime()).toString());
            collectOnly.addElement(“ET ” + new Date(Slave_AgentInfo.getCreationTime()).toString());
            executeOnly.addElement(“OA ” + Slave_AgentInfo.getOrigin());
            executeOnly.addElement(“ID ” + Slave_AgentInfo.getAgletID());
            collectOnly.addElement(“ID ” + Slave_AgentInfo.getAgletID());
        } catch (Exception e) {;}

        addMobilityListener(
            new MobilityAdapter() {
```

```
            public void onArrival(MobilityEvent me) {
                Enumeration enum = getAgletContext().getAgletProxies();
                int count = 1; //List counter

                //Looking for AirlineAgent
                while(enum.hasMoreElements()) {
                    // get the proxy in this current context
                    AgletProxy proxy = (AgletProxy)enum.nextElement();

                    // checking for server AirlineAgent
                    try {
                        String agentName = proxy.getAgletClassName().toString();
                        if(agentName.equals(“AirlineAgent”)) {
                            // keeping Airline_Agent's proxy
                            proxyAirlineAgent = proxy;
                            AgletInfo AirlineAgentInfo = proxyAirlineAgent.getAgletInfo();
                            executeOnly.addElement(“EA ” + AirlineAgentInfo.getOrigin());
                            collectOnly.addElement(“EA ” + AirlineAgentInfo.getOrigin());
                        }
                    }catch(Exception e) {
                    System.out.println(e.getMessage());
                    }
                }
            }
        }
    );
}


// Method onDisposing
public void onDisposing() {
}


// Method run
public void run() {
    byte[][] data;
    String[] result;
    Date d = new Date();

    Security.insertProviderAt(new IAIK(), 2);

    //Sending readOnly container to Airline agent for decryption
    try {
        Message msgDECRYPT = new Message(“DECRYPT”);
        msgDECRYPT.setArg(“argSlaveProxy”, Slave_AgentProxy);
        msgDECRYPT.setArg(“argServerAddress”, Slave_AgentInfo.getOrigin().toString());
        msgDECRYPT.setArg(“argDigitalSig”, digitalSig);
        msgDECRYPT.setArg(“argReadOnly”, readOnly);
        proxyAirlineAgent.sendFutureMessage(msgDECRYPT);
    } catch (Exception e) {;}
}


// Method message handling
public boolean handleMessage(Message msg) {
    if (msg.sameKind(“RO_CONTAINER”)) {
        RO_Con = (Vector)msg.getArg(“argRO_CONTAINER”);
        Enumeration enumRO = RO_Con.elements();
        while (enumRO.hasMoreElements()) {
            userPurchaseReq = (String[])enumRO.nextElement();
            verifySignature(MApubKey, digitalSig, userPurchaseReq);
        }
        executeOnly.addElement(“SG ” + verifies);
        for(int i=0; i<Array.getLength(userPurchaseReq); i++) {
            executeOnly.addElement(“UD ” + userPurchaseReq[i]);
        }

        if (verifies) {
            // Find available flight
            try {
                Message msgFIND_FLIGHT = new Message(“FIND_FLIGHT”);
                msgFIND_FLIGHT.setArg(“argOrigin”, userPurchaseReq[0]);
                msgFIND_FLIGHT.setArg(“argDestination”, userPurchaseReq[1]);
                msgFIND_FLIGHT.setArg(“argDeparture”, userPurchaseReq[2]);
                proxyAirlineAgent.sendFutureMessage(msgFIND_FLIGHT);
            } catch (Exception e) {;}
        } else
        System.out.println(“Signature not verified”);
        return true;
    } else
    if (msg.sameKind(“DATE_XMATCH”)) {
        executeOnly.addElement(“XDT”);
        collectOnly.addElement(“XDT”);

        // send recorded state to Master Agent
        try {
            Message msgRecordedState = new Message(“Process Fail”);
            msgRecordedState.setArg(“argNoiseNo”, NoiseCount);
            msgRecordedState.setArg(“argES”, executeOnly);
            msgRecordedState.setArg(“argCS”, collectOnly);
            Master_AgentProxy.sendFutureMessage(msgRecordedState);
            Slave_AgentProxy.dispose();
        } catch (Exception e) {;}
        return true;
    } else
    if (msg.sameKind(“FLIGHT_XAVAILABLE”)) {
        executeOnly.addElement(“XFL”);
        collectOnly.addElement(“XFL”);

        // send recorded state to Master Agent
        try {
            Message msgRecordedState = new Message(“Process Fail”);
            msgRecordedState.setArg(“argNoiseNo”, NoiseCount);
            msgRecordedState.setArg(“argES”, executeOnly);
            msgRecordedState.setArg(“argCS”, collectOnly);
            Master_AgentProxy.sendFutureMessage(msgRecordedState);
            Slave_AgentProxy.dispose();
        } catch (Exception e) {;}
        return true;
    } else
    if (msg.sameKind(“FLIGHT_FOUND”)) {
        recvFlightNo = (String)msg.getArg(“argFlightNo”);
        recvDeparture = ((Integer)msg.getArg(“argDeparture”)).intValue();
        recvArrival = ((Integer)msg.getArg(“argArrival”)).intValue();
        int fstSeats = ((Integer)msg.getArg(“argSeats1”)).intValue();
        int bizSeats = ((Integer)msg.getArg(“argSeats2”)).intValue();
        int ecoSeats = ((Integer)msg.getArg(“argSeats3”)).intValue();
        passanger = (Integer.parseInt(userPurchaseReq[3]) + Integer.parseInt(userPurchaseReq[4]) +
                Integer.parseInt(userPurchaseReq[5]));

    if(fstSeats >= passanger) {
        classType = “Fst”;
        fstSeats = fstSeats - passanger;
    } else if(bizSeats >= passanger) {
        classType = “Biz”;
        bizSeats = bizSeats - passanger;
    } else if(ecoSeats >= passanger) {
        classType = “Eco”;
        ecoSeats = ecoSeats - passanger;
    }

    if(passanger > 0){
        try {
            rqstFareCount++;
            Message msgFLIGHT_FARE = new Message(“FLIGHT_FARE”);
            msgFLIGHT_FARE.setArg(“argFlightNo”, recvFlightNo);
            msgFLIGHT_FARE.setArg(“argSeatClass”, classType);
            msgFLIGHT_FARE.setArg(“argNoPassAdult”, userPurchaseReq[3]);
            msgFLIGHT_FARE.setArg(“argNoPassChild”, userPurchaseReq[4]);
            msgFLIGHT_FARE.setArg(“argNoPassInfant”, userPurchaseReq[5]);
            proxyAirlineAgent.sendFutureMessage(msgFLIGHT_FARE);
        } catch (Exception e) {;}
    }
    return true;
} else
if (msg.sameKind(“FARE_RESULT”)) {
    String statusSearch=“”;
    String resultFF=“”;
    double ObfuscateUserBudget=0;
    totalFare=0;

    recvFareCount++;
    String recvSeatClass = (String)msg.getArg(“argSeatClass”);
    String recvPassanger1 = (String)msg.getArg(“argPassanger1”);
    double recvFare1 = ((Double)msg.getArg(“argFare1”)).doubleValue();
    String recvPassanger2 = (String)msg.getArg(“argPassanger2”);
    double recvFare2 = ((Double)msg.getArg(“argFare2”)).doubleValue();
    String recvPassanger3 = (String)msg.getArg(“argPassanger3”);
    double recvFare3 = ((Double)msg.getArg(“argFare3”)).doubleValue();
    totalFare = recvFare1 + recvFare2 + recvFare3;

    int exeCount,colCount;
    int RandN=Integer.parseInt(userPurchaseReq[9]);
    double ObfuscateFare=rs3(totalFare,RandN);

    if(NoiseCount==0)
        ObfuscateUserBudget= Double.parseDouble(userPurchaseReq[6]);
    else
        ObfuscateUserBudget= Double.parseDouble(userPurchaseReq[6+NoiseCount]);
    if(ObfuscateFare <= ObfuscateUserBudget) {
        recvFareCount=0;
        rqstFareCount=0;
        executeOnly.addElement(“STA”);
        executeOnly.addElement(“FI_FN ” + recvFlightNo);
        collectOnly.addElement(“FI_FN ” + recvFlightNo);
        executeOnly.addElement(“FI_CC ” + recvSeatClass);
        collectOnly.addElement(“FI_CC ” + recvSeatClass);
        executeOnly.addElement(“FI_DP ” + recvDeparture);
executeOnly.addElement(“FI_AR ” + recvArrival);
collectOnly.addElement(“FI_DP ” + recvDeparture);
collectOnly.addElement(“FI_AR ” + recvArrival);
executeOnly.addElement(“FI_PA ” + userPurchaseReq[3] +“ ”+ recvPassanger1);
collectOnly.addElement(“FI_PA ” + userPurchaseReq[3] +“ ”+ recvPassanger1);
executeOnly.addElement(“FI_FA ” + recvFare1);
collectOnly.addElement(“FI_FA ” + recvFare1);
executeOnly.addElement(“FI_PC ” + userPurchaseReq[4] +“ ”+ recvPassanger2);
collectOnly.addElement(“FI_PC ” + userPurchaseReq[4] +“ ”+ recvPassanger2);
executeOnly.addElement(“FI_FC ” + recvFare2);
collectOnly.addElement(“FI_FC ” + recvFare2);
executeOnly.addElement(“FI_PI ” + userPurchaseReq[5] +“ ”+ recvPassanger3);
collectOnly.addElement(“FI_PI ” + userPurchaseReq[5] +“ ”+ recvPassanger3);
executeOnly.addElement(“FI_FI ” + recvFare3);
collectOnly.addElement(“FI_FI ” + recvFare3);
executeOnly.addElement(“TF ” + totalFare);
collectOnly.addElement(“TF ” + totalFare);

// send recorded state to Master Agent
try {
    Iterator EO=executeOnly.iterator();
    exeCount=0;
    while(EO.hasNext()) {
        try {
            exeState[exeCount] = EO.next().toString();
            exeCount++;
        }catch (Exception e) {;}
    }

    Iterator CO=collectOnly.iterator();
    colCount=0;
    while(CO.hasNext()) {
        try {
            colState[colCount] = CO.next().toString();
            colCount++;
        }catch (Exception e) {;}
    }

    // send ExecuteState and CollectState to be sign by AirlineAgent
    Message msgSignature = new Message(“Get Signature”);
    msgSignature.setArg(“argExecuteData”, exeState);
    msgSignature.setArg(“argCollectData”, colState);
    proxyAirlineAgent.sendFutureMessage(msgSignature);
    NoiseCount++;

    if(NoiseCount<3) {
        if(passanger > 0){
            try {
                rqstFareCount++;
                classType=“Fst”;
                Message msgFLIGHT_FARE = new Message(“FLIGHT_FARE”);
                msgFLIGHT_FARE.setArg(“argFlightNo”, recvFlightNo);
                msgFLIGHT_FARE.setArg(“argSeatClass”, classType);
                msgFLIGHT_FARE.setArg(“argNoPassAdult”, userPurchaseReq[3]);
                msgFLIGHT_FARE.setArg(“argNoPassChild”, userPurchaseReq[4]);
                msgFLIGHT_FARE.setArg(“argNoPassInfant”, userPurchaseReq[5]);
                proxyAirlineAgent.sendFutureMessage(msgFLIGHT_FARE);
            } catch (Exception e) {;}
        }
    }
    } catch (Exception e) {;}
}
else
if(recvSeatClass.equals(“Fst”)) {
    classType=“Biz”;
    statusSearch=“Again”;
}
else if(recvSeatClass.equals(“Biz”)) {
    classType=“Eco”;
    statusSearch=“Again”;
}
else
    statusSearch=“Fail”;

if (statusSearch==“Again”) {
    statusSearch=“”;
    rqstFareCount=0;
    recvFareCount=0;
    totalFare=0;

    if(passanger > 0){
        try {
            rqstFareCount++;
            Message msgFLIGHT_FARE = new Message(“FLIGHT_FARE”);
            msgFLIGHT_FARE.setArg(“argFlightNo”, recvFlightNo);
            msgFLIGHT_FARE.setArg(“argSeatClass”, classType);
            msgFLIGHT_FARE.setArg(“argNoPassAdult”, userPurchaseReq[3]);
            msgFLIGHT_FARE.setArg(“argNoPassChild”, userPurchaseReq[4]);
            msgFLIGHT_FARE.setArg(“argNoPassInfant”, userPurchaseReq[5]);
            proxyAirlineAgent.sendFutureMessage(msgFLIGHT_FARE);
        } catch (Exception e) {;}
    } else {
        NoiseCount++;
        executeOnly.addElement(“XF”);
        collectOnly.addElement(“XF”);

        // send recorded state to Master Agent
        try {
            Message msgRecordedState = new Message(“Process Fail”);
            msgRecordedState.setArg(“argNoiseNo”, NoiseCount);
            msgRecordedState.setArg(“argES”, executeOnly);
            msgRecordedState.setArg(“argCS”, collectOnly);
            Master_AgentProxy.sendFutureMessage(msgRecordedState);
        } catch (Exception e) {;}

        if(NoiseCount<3) {
            if(passanger > 0){
                try {
                    rqstFareCount++;
                    classType=“Fst”;
                    Message msgFLIGHT_FARE = new Message(“FLIGHT_FARE”);
                    msgFLIGHT_FARE.setArg(“argFlightNo”, recvFlightNo);
                    msgFLIGHT_FARE.setArg(“argSeatClass”, classType);
                    msgFLIGHT_FARE.setArg(“argNoPassAdult”, userPurchaseReq[3]);
                    msgFLIGHT_FARE.setArg(“argNoPassChild”, userPurchaseReq[4]);
                    msgFLIGHT_FARE.setArg(“argNoPassInfant”, userPurchaseReq[5]);
                    proxyAirlineAgent.sendFutureMessage(msgFLIGHT_FARE);
                } catch (Exception e) {;}
            }
        }
```

```
            }
        } else if (statusSearch==“Fail”) {
            NoiseCount++;
            executeOnly.addElement(“XF”);
            collectOnly.addElement(“XF”);

            // send recorded state to Master Agent
            try {
                Message msgRecordedState = new Message(“Process Fail”);
                msgRecordedState.setArg(“argNoiseNo”, NoiseCount);
                msgRecordedState.setArg(“argES”, executeOnly);
                msgRecordedState.setArg(“argCS”, collectOnly);
                Master_AgentProxy.sendFutureMessage(msgRecordedState);
            }catch (Exception e) {;}

            if(NoiseCount<3) {
                if(passanger > 0){
                    try {
                        rqstFareCount++;
                        classType=“Fst”;
                        Message msgFLIGHT_FARE = new Message(“FLIGHT_FARE”);
                        msgFLIGHT_FARE.setArg(“argFlightNo”, recvFlightNo);
                        msgFLIGHT_FARE.setArg(“argSeatClass”, classType);
                        msgFLIGHT_FARE.setArg(“argNoPassAdult”, userPurchaseReq[3]);
                        msgFLIGHT_FARE.setArg(“argNoPassChild”, userPurchaseReq[4]);
                        msgFLIGHT_FARE.setArg(“argNoPassInfant”, userPurchaseReq[5]);
                        proxyAirlineAgent.sendFutureMessage(msgFLIGHT_FARE):
                    } catch (Exception e) {;}
                }
            }
        }
        return true;
    } else
    if (msg.sameKind(“Signed State”)) {
        SlaveCount++;
        byte[] ESdigitalSig = (byte[])msg.getArg(“argES_Sig”);
        byte[] CSdigitalSig = (byte[])msg.getArg(“argCS_Sig”);

        // Send Recorded State Container to Master Agent
        try {
            Message msgRecordedState = new Message(“Recorded State”);
            msgRecordedState.setArg(“argNoiseNo”, NoiseCount);
            msgRecordedState.setArg(“argES”, exeState);
            msgRecordedState.setArg(“argES_Sig”, ESdigitalSig);
            msgRecordedState.setArg(“argCS”, colState);
            msgRecordedState.setArg(“argCS_Sig”, CSdigitalSig);
            Master_AgentProxy.sendFutureMessage(msgRecordedState);
        } catch (Exception e) {;}
        return true;
    } else
    return false;
}


// Method RSA encryption
public byte[][] RSAencrypt(PublicKey pubKey, String[] text) {
    byte[][] cipherText=new byte[java.lang.reflect.Array.getLength(text)][];
    byte[] textByte=null;
    String tempStr = “”;
```

```
        try {
            Cipher cipher = Cipher.getInstance(“RSA”,“IAIK”);
            cipher.init(Cipher.ENCRYPT_MODE,pubKey);
            for(int i=0; i<java.lang.reflect.Array.getLength(text); i++) {
                tempStr = addValidStr(text[i]);
                textByte = tempStr.getBytes();
                cipherText[i]=cipher.doFinal(textByte);
            }
        } catch(Exception e) {
        System.out.println(“[PKI encryption error] ” + e.toString());
        }
        return cipherText;
    }


    // Method RSA decryption
    public String[] RSAdecrypt(PrivateKey pvKey, byte[][] cipherText) {
        String[] plainText = new String[Array.getLength(cipherText)];
        byte[] cipherByte=null;
        String tempStr = “”;

        try {
            Cipher cipher = Cipher.getInstance(“RSA”,“IAIK”);
            cipher.init(Cipher.DECRYPT_MODE,pvKey);
            for (int i=0; i<Array.getLength(cipherText); i++) {
                cipherByte = cipher.doFinal(cipherText[i]);
                tempStr = new String(cipherByte);
                plainText[i] = cutValidStr(tempStr);
            }
        } catch(Exception e) {
        System.out.println(“[PKI decryption error] ” + e.toString());
        }
        return plainText;
    }


    // Method add string
    public String addValidStr(String rawStr){
        int chrAdded,i,strLength;

        if (rawStr == null)
            rawStr = “”;

        strLength = rawStr.length()%8;
        chrAdded = 8-strLength;

        if (strLength > 0){
            for (i=0; i<chrAdded; i++)
                rawStr = rawStr + “X”;
            rawStr = rawStr + “ADDCHAR” + String.valueOf(chrAdded);
        }else{
            if (rawStr.length()!=0)
                rawStr = rawStr + String.valueOf(strLength);
            else{
                rawStr = “DATANUL0”;
            }
        }
        return rawStr;
    }
```

```
// Method cut string
public String cutValidStr(String rawStr){
      String tempStr = “”;
      int i = Integer.valueOf(String.valueOf(rawStr.charAt((rawStr.length())-1))).intValue();

      if (rawStr.length()!=8)
            tempStr = rawStr.substring(0,rawStr.length()-(i+8));
      return tempStr;
}


// Method generate digital signature
public byte[] genSignature(PrivateKey pvKey, String[] plainText) {
      byte[] sig=null;
      byte[] cipherText;

      try {
            Signature genSig = Signature.getInstance(“SHA1withRSA”);
            genSig.initSign(pvKey);
            try {
                  for (int i=0; i<Array.getLength(plainText); i++) {
                        cipherText = plainText[i].getBytes();
                        genSig.update(cipherText); //add msg to be sign
                  }
            } catch(Exception e){
            System.out.println(“[Updating signature error] ” + e.toString());
            }
            sig = genSig.sign();
      } catch(Exception e){
      System.out.println(“[Generating signature error] ” + e.toString());
      }
      return sig;
}


// Method verifying digital signature
public void verifySignature(PublicKey pubKey, byte[] inSig, String[] plainText) {
      byte[] inCipher;

      try {
            Signature verifySig = Signature.getInstance(“SHA1withRSA”);
            verifySig.initVerify(pubKey);
            for (int i=0; i<Array.getLength(plainText); i++) {
                  inCipher = plainText[i].getBytes();
                  verifySig.update(inCipher); // add msg that need to be verified
            }
            verifies  verifySig.verify(inSig);
            System.out.println(“Verifies=” + verifies);
      } catch(Exception e){
      System.out.println(“[Verifying signature error] ” + e.toString());
      }
}


// Method polynomial calculation
public double polynomial(int a, double b, int c, int x) {
      double result;

      result=(a * Math.pow(x,2)) + (b * x) + c;
```

```
        return result;
    }


    // Method Random Sequnce 3-level obfuscation algorithm

    public double rs3(double budget,int randomNumber) {
        int branch;
        int newRnd = randomNumber;
        for (int i=0; i<MAXLEVEL; i++) {
            branch = newRnd % MAXLEVEL;
            Random selector = new Random(branch);
            newRnd = (int) (selector.nextDouble() * 1000000);
            budget = polynomial(branch, budget, (newRnd % randomNumber), randomNumber);
        }
        return budget;
    }

}
```

D.3 Airline System (AirlineAgent.java)

```
import com.ibm.aglet.*;
import java.net.URL;
import iaik.security.provider.IAIK;
import javax.crypto.KeyGenerator;
import java.security.*;
import java.lang.reflect.*;
import javax.crypto.Cipher;
import java.util.*;
import java.sql.*;


public class AirlineAgent extends Aglet {
    PrivateKey privKey=null;
    PublicKey pubKey=null;
    PublicKey slavePubKey=null;
    byte[] digitalSig;
    Vector readOnly = new Vector();
    Vector RO_Con = new Vector();
    boolean verifies=false;
    AgletProxy proxySlaveAgent;
    AgletProxy RequestAgentProxy;
    AgletProxy AirlineAgentProxy;
    String RqstOrigin, RqstDestination, RqstDeparture, RqstTripType, RqstReturn;
    double discount=1;
    AgletInfo AirlineAgentInfo;
    String slaveOriginServer;
    String Address;
    static final String DB = “jdbc:odbc:Airline_A”;
    static final String USER = “”;
    static final String PASSWORD = “”;
    Connection conSearch;
    Connection conUpdate;


    //Constructor AirlineAgent
    public AirlineAgent() {
        Security.insertProviderAt(new IAIK(), 2);
    }


    // Method onCreation
    public void onCreation(Object args) {
        generateKeys(); //generate public and private key
        AirlineAgentInfo = getAgletInfo();
        AirlineAgentProxy = getAgletContext().getAgletProxy(AirlineAgentInfo.getAgletID());
        Address = AirlineAgentInfo.getOrigin().toString();

        //Register server address and public key to CA server
        try {
            URL CAServer = new URL("atp://cs-research01.aston.ac.uk:4444/");
            Object arg[]=new Object[] {Address,pubKey};
            AgletProxy RegisterAgentProxy = getAgletContext().createAglet(getCodeBase(),"RegisterAgent", arg);
            RegisterAgentProxy.dispatch(CAServer);
        } catch(Exception e) {;}

        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            conSearch = DriverManager.getConnection(DB, USER, PASSWORD);
            conUpdate = DriverManager.getConnection(DB, USER, PASSWORD);
        }catch (Exception e) {
        e.printStackTrace();
        }
    }


    // Method onDisposing
    public void onDisposing() {
    }


    // Method run
    public void run() {
    }


    // Method message handling
    public boolean handleMessage(Message msg) {
        boolean notMatch=false;

        if (msg.sameKind("DECRYPT")) {
            proxySlaveAgent = (AgletProxy)msg.getArg("argSlaveProxy");
            slaveOriginServer = (String)msg.getArg("argServerAddress");
            digitalSig = (byte[])msg.getArg("argDigitalSig");
            readOnly = (Vector)msg.getArg("argReadOnly");

            //Request Slave Origin Server Public Key from CA server
            try {
                URL CAServer = new URL("atp://cs-research01.aston.ac.uk:4444/");
                Object argReq[]=new Object[] {slaveOriginServer,AirlineAgentProxy};
                AgletProxy RequestAgentProxy = getAgletContext().createAglet(getCodeBase(),"RequestAgent", argReq);
                RequestAgentProxy.dispatch(CAServer);
            } catch(Exception e) {;}
            return true;
        } else
        if (msg.sameKind("PublicKey")) {
            byte[][] data;
```

```
          String[] result;

          String serAdd = (String)msg.getArg(“argServerAddress”);
          slavePubKey = (PublicKey)msg.getArg(“argPublicKey”);
          Enumeration enumRO = readOnly.elements();
          while (enumRO.hasMoreElements()) {
                data = (byte[][])enumRO.nextElement();
                result = RSAdecrypt(slavePubKey,data);
                verifySignature(slavePubKey, digitalSig, result);
                RO_Con.addElement(result);
          }

          if (verifies) {
          //Sending readOnly container back to slave agent for further action
          try {
                Message msgRO_CONTAINER = new Message(“RO_CONTAINER”);
                msgRO_CONTAINER.setArg(“argRO_CONTAINER”, RO_Con);
                proxySlaveAgent.sendFutureMessage(msgRO_CONTAINER);
          } catch (Exception e) {;}
    }
    return true;
    } else
    if (msg.sameKind(“FIND_FLIGHT”)) {
          int FstSeats=0;
          int BizSeats=0;
          int EcoSeats=0;

          RqstOrigin = (String)msg.getArg(“argOrigin”);
          RqstDestination = (String)msg.getArg(“argDestination”);
          RqstDeparture = (String)msg.getArg(“argDeparture”);
          try {
                String queryFlightDB_Depart = “Select FlightNo, Departure, Arrival, ” +
                                              “Seats1, Seats2, Seats3 ” +
                                              “From FlightInfo ” +
                                              “Where Origin = ’” + RqstOrigin + “’” +
                                              “and Destination = ’” + RqstDestination + “’”;

                Statement stmt = conSearch.createStatement();
                ResultSet rs = stmt.executeQuery(queryFlightDB_Depart);

                if (rs.next()) {
                      String FlightNo = rs.getString(1);
                      int Departure = rs.getInt(2);
                      int Arrival = rs.getInt(3);
                      int Seats1 = rs.getInt(4);
                      int Seats2 = rs.getInt(5);
                      int Seats3 = rs.getInt(6);

                      if(Departure != Integer.parseInt(RqstDeparture)) {
                            notMatch=true;
                      }
                      else {
                      // check seats availability
                      String querySeats = “Select SUM(Seats1), SUM(Seats2), ” +
                                          “SUM(Seats3) From FlightReservation ” +
                                          “Where theFlightNo=’” + FlightNo + “’”;

                      Statement stmt1 = conSearch.createStatement();
                      ResultSet rs1 = stmt1.executeQuery(querySeats);
```

```
            rs1.next();
            int ResSeats1 = rs1.getInt(1);
            int ResSeats2 = rs1.getInt(2);
            int ResSeats3 = rs1.getInt(3);

            FstSeats=Seats1 - ResSeats1;
            BizSeats=Seats2 - ResSeats2;
            EcoSeats=Seats3 - ResSeats3;

            rs1.close();
            stmt1.close();

            try {
                Message msgFLIGHT_FOUND = new Message(“FLIGHT_FOUND”);
                msgFLIGHT_FOUND.setArg(“argFlightNo”,FlightNo);
                msgFLIGHT_FOUND.setArg(“argDeparture”,Departure);
                msgFLIGHT_FOUND.setArg(“argArrival”,Arrival);
                msgFLIGHT_FOUND.setArg(“argSeats1”,FstSeats);
                msgFLIGHT_FOUND.setArg(“argSeats2”,BizSeats);
                msgFLIGHT_FOUND.setArg(“argSeats3”,EcoSeats);
                proxySlaveAgent.sendFutureMessage(msgFLIGHT_FOUND);
            } catch (Exception e) {;}
            }
        } else {
            try {
                Message msgFLIGHT_XAVAILABLE = new Message(“FLIGHT_XAVAILABLE”);
                proxySlaveAgent.sendFutureMessage(msgFLIGHT_XAVAILABLE);
            } catch (Exception e) {;}
        }

        rs.close();
        stmt.close();

        if(notMatch) {
            try {
                Message msgDATE_XMATCH = new Message(“DATE_XMATCH”);
                proxySlaveAgent.sendFutureMessage(msgDATE_XMATCH);
            } catch (Exception e) {;}
        }
    }catch (Exception e) {
    e.printStackTrace();
    }
    return true;
} else
if (msg.sameKind(“FLIGHT_FARE”)) {
    int flightFare=0;
    double AdultFare=0;
    double ChildFare=0;
    double InfantFare=0;

    String reqstFlightNo = (String)msg.getArg(“argFlightNo”);
    String reqstSeatClass = (String)msg.getArg(“argSeatClass”);
    String reqstNoPassAdult = (String)msg.getArg(“argNoPassAdult”);
    String reqstNoPassChild = (String)msg.getArg(“argNoPassChild”);
    String reqstNoPassInfant = (String)msg.getArg(“argNoPassInfant”);

    try {
        String queryFare = “Select theFare From FlightFare ” +
                           “Where theFlightNo = '”+ reqstFlightNo + “'”;
        Statement stmt = conSearch.createStatement();
        ResultSet rs = stmt.executeQuery(queryFare);

        rs.next();
        flightFare = rs.getInt(1);

        if(reqstSeatClass==“Fst”) {
            if(Integer.parseInt(reqstNoPassAdult) > 0) {
                AdultFare = discount * (flightFare * Integer.parseInt(reqstNoPassAdult));
            }
            if(Integer.parseInt(reqstNoPassChild) > 0) {
                ChildFare = discount * ((flightFare * 0.75) * Integer.parseInt(reqstNoPassChild));
            }
            if(Integer.parseInt(reqstNoPassInfant) > 0) {
                InfantFare = discount * ((flightFare * 0.10) * Integer.parseInt(reqstNoPassInfant));
            }
        } else if(reqstSeatClass==“Biz”) {
            if(Integer.parseInt(reqstNoPassAdult) > 0) {
                AdultFare = discount * ((flightFare * 0.65) * Integer.parseInt(reqstNoPassAdult));
            }
            if(Integer.parseInt(reqstNoPassChild) > 0) {
                ChildFare = discount * (((flightFare * 0.65) * 0.75) * Integer.parseInt(reqstNoPassChild));
            }
            if(Integer.parseInt(reqstNoPassInfant) > 0) {
                InfantFare = discount * (((flightFare * 0.65) * 0.10) * Integer.parseInt(reqstNoPassInfant));
            }
        } else if(reqstSeatClass==“Eco”) {
            if(Integer.parseInt(reqstNoPassAdult) > 0) {
                AdultFare = discount * (((flightFare * 0.2) * Integer.parseInt(reqstNoPassAdult)));
            }
            if(Integer.parseInt(reqstNoPassChild) > 0) {
                ChildFare = discount * ((((flightFare * 0.20) * 0.75) * Integer.parseInt(reqstNoPassChild)));
            }
            if(Integer.parseInt(reqstNoPassInfant) > 0) {
                InfantFare = discount * ((((flightFare * 0.20) * 0.10) * Integer.parseInt(reqstNoPassInfant)));
            }
        }
        rs.close();
        stmt.close();
    } catch (Exception e) {;}

    try {
        Message msgFARE_RESULT = new Message(“FARE_RESULT”);
        msgFARE_RESULT.setArg(“argSeatClass”, reqstSeatClass);
        msgFARE_RESULT.setArg(“argPassanger1”, “Adult”);
        msgFARE_RESULT.setArg(“argFare1”, AdultFare);
        msgFARE_RESULT.setArg(“argPassanger2”, “Child”);
        msgFARE_RESULT.setArg(“argFare2”, ChildFare);
        msgFARE_RESULT.setArg(“argPassanger3”, “Infant”);
        msgFARE_RESULT.setArg(“argFare3”, InfantFare);
        proxySlaveAgent.sendMessage(msgFARE_RESULT);
    } catch (Exception e) {;}
    return true;
} else
if (msg.sameKind(“Get Signature”)) {
    String[] exeState = (String[])msg.getArg(“argExecuteData”);
    String[] colState = (String[])msg.getArg(“argCollectData”);

    byte[] ESdigitalSig = genSignature(privKey, exeState);
    byte[] CSdigitalSig = genSignature(privKey, colState);
```

```
            // send signed state to slave agent
            try {
                Message msgSignedState = new Message(“Signed State”);
                msgSignedState.setArg(“argES_Sig”, ESdigitalSig);
                msgSignedState.setArg(“argCS_Sig”, CSdigitalSig);
                proxySlaveAgent.sendMessage(msgSignedState);
            } catch (Exception e) {;}
            return true;
        } else
        return false;
}


// Method generate public and private key
public void generateKeys() {
    try {
        KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance(“RSA”,“IAIK”);

        keyPairGen.initialize(1024);

        KeyPair pair = keyPairGen.generateKeyPair();
        privKey = pair.getPrivate();
        pubKey = pair.getPublic();
    } catch (Exception e) {
    System.out.println(“[Generating RSA-PKI keypair error]”);
    e.printStackTrace();
    }
}


// Method RSA encryption
public byte[][] RSAencrypt(PrivateKey pvKey, String[] text) {
    byte[][] cipherText=new byte[java.lang.reflect.Array.getLength(text)][];
    byte[] textByte=null;
    String tempStr = “”;

    try {
        Cipher cipher = Cipher.getInstance(“RSA”,“IAIK”);
        cipher.init(Cipher.ENCRYPT_MODE,pvKey);
        for(int i=0; i<java.lang.reflect.Array.getLength(text); i++) {
            tempStr = addValidStr(text[i]);
            textByte = tempStr.getBytes();
            cipherText[i]=cipher.doFinal(textByte);
        }
    } catch(Exception e) {
    System.out.println(“[PKI encryption error] ” + e.toString());
    }
    return cipherText;
}


// Method RSA decryption
public String[] RSAdecrypt(PublicKey pbKey, byte[][] cipherText) {
    String[] plainText = new String[java.lang.reflect.Array.getLength(cipherText)];
    byte[] cipherByte=null;
    String tempStr = “”;

    try {
        Cipher cipher = Cipher.getInstance(“RSA”,“IAIK”);
        cipher.init(Cipher.DECRYPT_MODE,pbKey);
```

```
            for (int i=0; i<java.lang.reflect.Array.getLength(cipherText); i++) {
                cipherByte = cipher.doFinal(cipherText[i]);
                tempStr = new String(cipherByte);
                plainText[i] = cutValidStr(tempStr);
            }
        } catch(Exception e) {
        System.out.println(“[PKI decryption error] ” + e.toString());
        }
        return plainText;
    }


    //Method add string
    public String addValidStr(String rawStr){
        int chrAdded,i,strLength;

        if (rawStr == null)
            rawStr = “”;

        strLength = rawStr.length()%8;
        chrAdded = 8-strLength;

        if (strLength > 0){
            for (i=0; i<chrAdded; i++)
                rawStr = rawStr + “X”;
            rawStr = rawStr + “ADDCHAR” + String.valueOf(chrAdded);
        } else{
        if (rawStr.length()!=0)
            rawStr = rawStr + String.valueOf(strLength);
        else{
            rawStr = “DATANUL0”;
            }
        }
        return rawStr;
    }


    // Method cut string
    public String cutValidStr(String rawStr){
        String tempStr = “”;
        int i = Integer.valueOf(String.valueOf(rawStr.charAt((rawStr.length())-1))).intValue();

        if (rawStr.length()!=8)
            tempStr = rawStr.substring(0,rawStr.length()-(i+8));
        return tempStr;
    }


    //Method generate digital signature
    public byte[] genSignature(PrivateKey privKey, String[] plainText) {
        byte[] sig=null;
        byte[] cipherText;

        try {
            Signature genSig = Signature.getInstance(“SHA1withRSA”);
            genSig.initSign(privKey);
            try {
                for (int i=0; i<java.lang.reflect.Array.getLength(plainText); i++) {
                    cipherText = plainText[i].getBytes();
                    genSig.update(cipherText); //add msg to be sign
                }
            } catch(Exception e){
            System.out.println(“[Updating signature error] ” + e.toString());
            }
            sig = genSig.sign();
        } catch(Exception e){
        System.out.println(“[Generating signature error] ” + e.toString());
        }
        return sig;
    }

    //Method verify digital signature
    public void verifySignature(PublicKey pubKey, byte[] inSig, String[] plainText) {
        byte[] inCipher;

        try {
            Signature verifySig = Signature.getInstance(“SHA1withRSA”);
            verifySig.initVerify(pubKey);
            for (int i=0; i<java.lang.reflect.Array.getLength(plainText); i++) {
                inCipher = plainText[i].getBytes();
                verifySig.update(inCipher); // add msg that need to be verified
            }
            verifies=verifySig.verify(inSig);
        } catch(Exception e){
        System.out.println(“[Verifying signature error] ” + e.toString());
        }
    }
}
```

D.4 Certificate Authority System (CAAgent.java)

```
import com.ibm.aglet.*;
import java.net.URL;
import java.security.*;
import java.util.Vector;
import java.util.*;
import iaik.security.provider.IAIK;

public class CAAgent extends Aglet {
    AgletProxy RequestServerProxy;
    Vector ServerInfo = new Vector();
    String serverAdd;
    PublicKey pubKey;

    //Constructor CAAgent
    public CAAgent() {
        Security.insertProviderAt(new IAIK(), 2);
    }

    //Method onCreation
    public void onCreation(Object args) {
    }

    //Method onDisposing
    public void onDisposing() {
    {

    //Method run
```

```
public void run() {
{

//Method message handling
public boolean handleMessage(Message msg) {
      if (msg.sameKind("Register")) {
            String recvServerAdd = (String)msg.getArg("argServerAddress");
            PublicKey recvPublicKey = (PublicKey)msg.getArg("argPublicKey");
            Iterator srvInfo = ServerInfo.iterator();
            while(srvInfo.hasNext()){
                  serverAdd = srvInfo.next().toString();
                  if(serverAdd .equals(recvServerAdd)) {
                  srvInfo.remove();
                  srvInfo.next();
                  srvInfo.remove();
            }
      }
      ServerInfo.addElement(recvServerAdd);
      ServerInfo.addElement(recvPublicKey);
      return true;
      } else
      if (msg.sameKind("Request PublicKey")) {
            boolean result=false;
            pubKey=null;
            String recvServertemp = (String)msg.getArg("argServerAddress");
            RequestServerProxy = (AgletProxy)msg.getArg("argAgentProxy");
            String recvServerAdd = recvServertemp.toString();
            Iterator SI = ServerInfo.iterator();
            while(SI.hasNext()) {
                  serverAdd = SI.next().toString();
                  if(serverAdd .equals(recvServerAdd)) {
                        result=true;
                        pubKey=(PublicKey)SI.next();
                  }
            }

            if(result) {
                  try {
                        Message msgREQSTPK = new Message("RequestedPK");
                        msgREQSTPK.setArg("argPubKey", pubKey);
                        RequestServerProxy.sendFutureMessage(msgREQSTPK);
                  } catch (Exception e) {;}
            }
            return true;
      } else
      return false;
      }
   }
```

D.5 Register Agent (RegisterAgent.java)

```
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.URL;
import java.security.*;
import java.util.*;
import javax.crypto.Cipher;
```

```
public class RegisterAgent extends Aglet {
    AgletProxy proxyCAAgent;
    String serverAdd;
    PublicKey pubKey;

    //Method onCreation
    public void onCreation(Object args) {
        Object init[] = (Object[]) args;
        serverAdd = (String)init[0];
        pubKey = (PublicKey)init[1];

        addMobilityListener(
            new MobilityAdapter(){
                public void onArrival(MobilityEvent me) {
                    Enumeration enum = getAgletContext().getAgletProxies();
                    int count = 1; //List counter
                    while(enum.hasMoreElements()){ //Looking for CAAgent
                        AgletProxy proxy = (AgletProxy)enum.nextElement();// get the proxy in this current context
                        // checking for server CAAgent
                        try {
                            String agentName = proxy.getAgletClassName().toString();
                            if(agentName.equals(“CAAgent”)) {
                                // keeping CAAgent proxy
                                proxyCAAgent = proxy;
                            }
                        }catch(Exception e) {
                        System.out.println(e.getMessage());
                        }
                    }
                }
            }
        );
    }

    //Method OnDisposing
    public void onDisposing(){
    }

    //Method run
    public void run(){
        //send message to CAAgent for registration
        try{
            Message msgRegister = new Message(“Register”);
            msgRegister.setArg(“argServerAddress”, serverAdd);
            msgRegister.setArg(“argPublicKey”, pubKey);
            proxyCAAgent.sendFutureMessage(msgRegister);
        } catch(Exception e) {;}
        dispose();
    }

    //Method message handling
    public boolean handleMessage(Message msg) {
        return false;
    }
}
```

D.6 Request Agent(RequestAgent.java)

```
import com.ibm.aglet.*;
import com.ibm.aglet.event.*;
import java.net.URL;
import java.security.*;
import java.util.*;
import javax.crypto.Cipher;

public class RequestAgent extends Aglet {
    private AgletProxy proxyCAAgent;
    private AgletProxy proxyRequestAgent;
    private AgletProxy proxyRequestedServer;
    AgletInfo RequestAgentInfo;
    String serverAdd;
    PublicKey pubKey;

    //Method onCreation
    public void onCreation(Object args) {
        Object init[] = (Object[]) args;
        serverAdd = (String)init[0];
        proxyRequestedServer = (AgletProxy)init[1];

        addMobilityListener(
            new MobilityAdapter(){
                public void onArrival(MobilityEvent me) {
                    Enumeration enum = getAgletContext().getAgletProxies();
                    int count = 1; //List counter

                    //Looking for CAAgent
                    while(enum.hasMoreElements()) {
                        // get the proxy in this current context
                        AgletProxy proxy = (AgletProxy)enum.nextElement();

                        // checking for server CAAgent
                        try {
                            String agentName = proxy.getAgletClassName().toString();
                            if(agentName.equals(“CAAgent”)) {
                                // keeping CAAgent proxy
                                proxyCAAgent = proxy;
                            }
                        }catch(Exception e) {
                        System.out.println(e.getMessage());
                        }
                    }
                }
            }
        );
    }

    //Method OnDisposing
    public void onDisposing() {
    }

    //Method run
    public void run() {
        RequestAgentInfo = getAgletInfo();
        proxyRequestAgent = getAgletContext().getAgletProxy(RequestAgentInfo.getAgletID());
        //Request PK from CAAgent
        try{
            Message msgRequestPK = new Message(“Request PublicKey”);
```

```
                msgRequestPK.setArg(“argServerAddress”, serverAdd);
                msgRequestPK.setArg(“argAgentProxy”, proxyRequestAgent);
                proxyCAAgent.sendFutureMessage(msgRequestPK);
            } catch(Exception e) { ;}
        }

        //Method message handling
        public boolean handleMessage(Message msg) {
            if (msg.sameKind(“RequestedPK”)) {
                pubKey = (PublicKey)msg.getArg(“argPubKey”);
                //send PK to requested agent
                try{
                    Message msgReqPK = new Message(“PublicKey”);
                    msgReqPK.setArg(“argServerAddress”, serverAdd);
                    msgReqPK.setArg(“argPublicKey”, pubKey);
                    proxyRequestedServer.sendFutureMessage(msgReqPK);
                } catch(Exception e) {;}
                dispose();
                return true;
            } else
            return false;
        }
}
```

D.7 Evaluation Agent (EvaluationAgent.java)

```
import com.ibm.aglet.*;
import java.net.URL;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import java.sql.*;
import iaik.security.provider.IAIK;
import java.security.*;
import javax.crypto.Cipher;


public class EvaluationAgent extends Aglet{
        private AgletProxy MasterAgentProxy;
        private AgletInfo EvaluationAgentInfo;
        private AgletProxy EvaluationAgentProxy;
        String[] recvES;
        byte[] recvES_Sig;
        String[] recvCS;
        byte[] recvCS_Sig;
        Vector recvExecuteState = new Vector();
        Vector recvCollectState = new Vector();
        List recvSAdetails = new ArrayList();
        List recvPubKeyList = new ArrayList();
        String[] recvMAdata;
        String remoteAddress;
        String rAddress;
        String bestOffer,
        String slaveID;
        PrivateKey MAprivKey;
        PublicKey remoteServPK;
        boolean verifies=false;
        double UserBudget;
        static final String DB = “jdbc:odbc:BlacklistAddress”;
static final String USER = “”;
static final String PASSWORD = “”;
Connection conSel;
Connection conIns;
static final String DB_Result = “jdbc:odbc:Result”;
static final String USER_Result = “”;
static final String PASSWORD_Result = “”;
Connection conResult_Sel;
Connection conResult_Ins;


//Constructor EvaluationAgent
public EvaluationAgent() {
      Security.insertProviderAt(new IAIK(), 2);
}


// Method onCreation
public void onCreation(Object args) {
      Object init[] = (Object[])args;
      MasterAgentProxy = (AgletProxy)init[0];
      recvES = (String[])init[1];
      recvES_Sig = (byte[])init[2];
      recvCS = (String[])init[3];
      recvCS_Sig = (byte[])init[4];
      recvMAdata = (String[]) init[5];
      recvSAdetails = (List) init[6];
      MAprivKey = (PrivateKey)init[7];
      recvPubKeyList = (List)init[8];
      String UB = (String)init[9];

      UserBudget = Double.parseDouble(UB);
      try{
            Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
            conSel = DriverManager.getConnection(DB, USER, PASSWORD);
            conIns = DriverManager.getConnection(DB, USER, PASSWORD);
      } catch (Exception e) {
      e.printStackTrace();
      }

      try{
            Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
            conResult_Sel = DriverManager.getConnection(DB_Result, USER_Result, PASSWORD_Result);
            conResult_Ins = DriverManager.getConnection(DB_Result, USER_Result, PASSWORD_Result);
      } catch (Exception e) {
      e.printStackTrace();
      }

      for(int z=0; z¡java.lang.reflect.Array.getLength(recvES); z++) {
            recvExecuteState.addElement(recvES[z]);
      }

      for(int i=0; i¡java.lang.reflect.Array.getLength(recvCS); i++) {
            recvCollectState.addElement(recvCS[i]);
      }

      Iterator PKlist = recvPubKeyList.iterator();

      Enumeration esElement = recvExecuteState.elements();
      while (esElement.hasMoreElements()) {
        String data=esElement.nextElement().toString();
        String indexData=data.substring(0,2);
        String rdata=data.substring(3,data.length());

        if (“EA”.equals(indexData)) {
            rAddress=rdata;
            while (PKlist.hasNext()) {
                try {
                    String Address = (String)PKlist.next();
                    if(Address.equals(rAddress)) {
                        remoteServPK = (PublicKey)PKlist.next();
                        break;
                    }
                } catch (Exception e) { ;}
            }
        }
    }

    verifySignature(remoteServPK, recvES_Sig, recvES);

    if(verifies) {
        verifies=false;
        verifySignature(remoteServPK, recvCS_Sig, recvCS);
    } else
    if(!verifies) {
        try {
            String querySel = “Select servAdd From serverAddress ” +
                              “Where servAdd=’” + remoteAddress + “’”;

            String queryIns = “Insert Into serverAddress (servAdd) ” +
                              “Values(’” + remoteAddress + “’) ”;

            Statement stmtSel = conSel.createStatement();
            ResultSet rsSel = stmtSel.executeQuery(querySel);

            if (!rsSel.next()) {
                Statement stmtIns = conIns.createStatement();
                stmtIns.executeUpdate(queryIns);
                conIns.close();
                stmtIns.close();
            }

            conSel.close();
            stmtSel.close();
        } catch (SQLException e) {
        e.printStackTrace();
        }
        dispose();
    }
    EvaluationAgentInfo = getAgletInfo();
    EvaluationAgentProxy = getAgletContext().getAgletProxy(EvaluationAgentInfo.getAgletID());
}


// Method onDisposing
public void onDisposing() {
    try {
        Message msgFinish = new Message(“Finished”);
        MasterAgentProxy.sendFutureMessage(msgFinish);
    } catch (Exception e) { ;}
```

```
}


// Method run
public void run() {
    String rawData;
    String realData;
    String index;
    String spcIndex;
    String statusSearch=“Y”;
    int ackRecv=0;
    boolean proceed=false;

    Enumeration eS = recvExecuteState.elements();
    while (eS.hasMoreElements()) {
        rawData=eS.nextElement().toString();
        index=rawData.substring(0,2);
        spcIndex=rawData.substring(0,3);
        realData=rawData.substring(3,rawData.length());

        //check user data
        if(“UD” .equals(index)) {
            for(int i=0; i¡java.lang.reflect.Array.getLength(recvMAdata); i++) {
                if(recvMAdata[i].equals(realData)) {
                    ackRecv++;
                    break;
                }
            }
        } else if(“SG” .equals(index)) {
            if(“true”.equals(realData)) {
                ackRecv++;
            }
        } else if ((“OA”.equals(index)) || (“ID”.equals(index)) || (“EA”.equals(index))) {
            if (“EA”.equals(index))
                remoteAddress=realData;
            if (“ID”.equals(index))
                slaveID=realData;
            Iterator SAdetails = recvSAdetails.iterator();
            while(SAdetails.hasNext()) {
                if(SAdetails.next().equals(realData)) {
                    ackRecv++;
                    break;
                }
            }
        } else if(“ET”.equals(index)) {
            ackRecv++;
        } else if((“XFL” .equals(spcIndex)) || (“XDT” .equals(spcIndex))) {
            statusSearch=“N”;
        }
    }

    if((ackRecv == 15) && (statusSearch==“Y”)) {
        String rawElement;
        String indexElement;
        String headerInfo;
        String bodyInfo;

        ackRecv=0;
        Enumeration cS = recvCollectState.elements();
        Enumeration eST = recvExecuteState.elements();
```

```
while (cS.hasMoreElements()) {
    rawElement=cS.nextElement().toString();
    indexElement=rawElement.substring(0,2);
    headerInfo=rawElement.substring(3,rawElement.length());
    bodyInfo=rawElement.substring(5,rawElement.length());
    if(“ET” .equals(indexElement)) {
        ackRecv++;
    } else if(“ID” .equals(indexElement)) {
        while (eST.hasMoreElements()) {
            rawData=eST.nextElement().toString();
            realData=rawData.substring(3,rawData.length());
            if(headerInfo .equals(realData)) {
                ackRecv++;
                break;
            }
        }
    } else if(“EA” .equals(indexElement)) {
        while (eST.hasMoreElements()) {
            rawData=eST.nextElement().toString();
            realData=rawData.substring(3,rawData.length());
            if(headerInfo .equals(realData)) {
                ackRecv++;
                break;
            }
        }
    } else if(“TF” .equals(indexElement)) {
        while (eST.hasMoreElements()) {
            rawData=eST.nextElement().toString();
            realData=rawData.substring(3,rawData.length());
            if(headerInfo .equals(realData)) {
                bestOffer=realData;
                if(Double.parseDouble(bestOffer) < UserBudget)
                    ackRecv++;
                break;
            }
        }
    } else if(“FI” .equals(indexElement)) {
        while (eST.hasMoreElements()) {
            rawData=eST.nextElement().toString();
            index=rawData.substring(0,2);
            if(“FI” .equals(index)) {
                realData=rawData.substring(5,rawData.length());
                if(bodyInfo .equals(realData)) {
                    ackRecv++;
                    break;
                }
            }
        }
    }
}
if(ackRecv==14)
    proceed=true;
}
else if (ackRecv != 15) {
    try {
        String querySel = “Select servAdd From serverAddress ” +
                                    “Where servAdd=’” + remoteAddress + “’”;

        String queryIns = “Insert Into serverAddress (servAdd) ” +
                                    “Values(’” + remoteAddress + “’) ”;
                Statement stmtSel = conSel.createStatement();
                ResultSet rsSel = stmtSel.executeQuery(querySel);

                if (!rsSel.next()) {
                    Statement stmtIns = conIns.createStatement();
                    stmtIns.executeUpdate(queryIns);
                    conIns.close();
                    stmtIns.close();
                }

                conSel.close();
                stmtSel.close();
            } catch (SQLException e) {
            e.printStackTrace();
            }
            dispose();
        }

        if(proceed) {
            try {
                String queryResult_Sel = “Select slaveID From ResultData ” +
                                          “Where slaveID=’” + slaveID + “’”;

                String queryResult_Ins = “Insert Into ResultData (slaveID,fromServerAddress, Offer) ” +
                                          “Values(’”+slaveID+“’,’”+remoteAddress+“’,’”+bestOffer+“’)”;

                Statement stmtResult_Sel = conResult_Sel.createStatement();
                ResultSet rsResult_Sel = stmtResult_Sel.executeQuery(queryResult_Sel);
                if(!rsResult_Sel.next()) {
                    Statement stmtResult_Ins = conResult_Ins.createStatement();
                    stmtResult_Ins.executeUpdate(queryResult_Ins);
                    conResult_Ins.close();
                    stmtResult_Ins.close();
                }

                conResult_Sel.close();
                stmtResult_Sel.close();
            } catch (SQLException e) {
            e.printStackTrace();
            }
            dispose();
        } else
        dispose();
}


//Method RSA decryption
public String[] RSAdecrypt(PrivateKey pvKey, byte[][] cipherText) {
    String[] plainText = new String[java.lang.reflect.Array.getLength(cipherText)];
    byte[] cipherByte=null;
    String tempStr = “”;

    try {
        Cipher cipher = Cipher.getInstance(“RSA”,“IAIK”);
        cipher.init(Cipher.DECRYPT_MODE,pvKey);
        for (int i=0; i¡java.lang.reflect.Array.getLength(cipherText); i++) {
            cipherByte = cipher.doFinal(cipherText[i]);
            tempStr = new String(cipherByte);
            plainText[i] = cutValidStr(tempStr);
```

```
                System.out.println("PlainText :" + plainText[i]);
            }
        } catch(Exception e) {
        System.out.println("[PKI decryption error] " + e.toString());
        }
        return plainText;
    }

    //Method cut string
    public String cutValidStr(String rawStr){
        String tempStr = "";
        int i = Integer.valueOf(String.valueOf(rawStr.charAt((rawStr.length())-1))).intValue();

        if (rawStr.length()!=8)
            tempStr = rawStr.substring(0,rawStr.length()-(i+8));
        return tempStr;
    }


    //Method verify digital signature
    public void verifySignature(PublicKey pubKey, byte[] inSig, String[] plainText) {
        byte[] inCipher;

        try {
            Signature verifySig = Signature.getInstance("SHA1withRSA");
            verifySig.initVerify(pubKey);
            for (int i=0; i¡java.lang.reflect.Array.getLength(plainText); i++) {
                inCipher = plainText[i].getBytes();
                verifySig.update(inCipher); // add msg that need to be verified
            }
            verifies=verifySig.verify(inSig);
        } catch(Exception e){
        System.out.println("[Verifying signature error] " + e.toString());
        }
    }
}
```