2-1-2001

Security in mobile agent systems: an approach to protect mobile agents from malicious host attacks

Dalia Fakhry

Security in Mobile Agent Systems:

A Model to Protect Mobile Agents from Malicious Host Attacks

By:
Dalia Fakhry

Thesis Supervisor:
Dr. Ahmed Sameh

A Thesis Submitted to:
The Department of Computer Science

At

T h e A m e r i c a n U n i v e r s i t y i n C a i r o

In partial fulfillment of the requirements for
The Degree of Master of Science

December 2000

Abstract

Mobile agents are autonomous programs that roam the Internet from machine to machine under their own control on behalf of their users to perform specific pre-defined tasks. In addition to that, a mobile agent can suspend its execution at any point; transfer itself to another machine then resume execution at the new machine without any loss of state.

Such a mobile model can perform many possible types of operations, and might carry critical data that has to be protected from possible attacks. The issue of agent security and specially agent protection from host attacks has been a hot topic and no fully comprehensive solution has been found so far. In this thesis, we examine the possible security attacks that hosts and agents suffer from. These attacks can take one of four possible forms: Attacks from host to host, from agents to hosts, from agents to agents (peer to peer) and finally from hosts to agents. Our main concern in this thesis is these attacks from a malicious host on an agent. These attacks can take many forms including rerouting, spying out code, spying out data, spying out control flow, manipulation of code, manipulation of data, manipulation of control flow, incorrect execution of code, masquerading and denial of execution.

In an attempt to solve the problem of malicious host attacks on agents, many partial solutions were proposed. These solutions ranged across simple legal protection, hardware solutions, partitioning, replication and voting, components, self-authentication, and migration history. Other solutions also included using audit logs, read-only state, append only logs, encrypted algorithms, digital signatures, partial result authentication codes, and code mess-up, limited life time of code and data as well as time limited black box security.

In this thesis, we present a three-tier solution. This solution is a combination of code mess up, encryption and time out. Choosing code mess-up as part of the solution was due to the several strengths of this method that is based on obfuscating the features of the code so that any attacker will find it very difficult to understand the original code. A new algorithm was developed in this thesis to implement code mess-up that uses the concept of variable disguising by altering the values of strings and numerical values.

Several encryption algorithms were studied to choose the best algorithm to use in the development of the proposed solution. The algorithms studied included DES, LUCIFER, MADRYGA, NEWDES, FEAL, REDOC, LOKI, KHUFU & KHAFRE, IDEA and finally MMB. The algorithm used was the DES algorithm due to several important factors including its key length.

Not any language can be used to implement mobile agents. Candidate languages should possess the portability characteristic and should be safe and secure enough to guarantee a protection for the mobile agent. In addition to that the language should be efficient in order to minimize the implementation overhead and the overhead of providing safety and security. Languages used to implement mobile agents include Java, Limbo, Telescript, and Safe TCL. The Java language was chosen as the programming language for this thesis due to its high security, platform independence, and multithreading. This is in addition to several powerful features that characterize the Java language as will be mentioned later on.

Implementing a mobile agent requires the assistance of a mobile agent system that helps in launching the agent from one host to another. There are many existing agent launching systems like Telescript, Aglets, Tacoma, Agent TCL and Concordia. Concordia was chosen to be the implementation tool used to launch our mobile agent. It is a software framework for developing, running and administering mobile agents, and it proved to be very efficient, and effective.

The results of our proposed solutions showed the strength of the proposed model in terms of fully protecting the mobile agent from possible malicious host attacks. The model could have several points of enhancements. These enhancements include changing the code mess-up algorithm to a more powerful one, using a different encryption technique, and implementing an agent re-charge mechanism to recharge the agent after it is timeout.

Acknowledgments

Many thanks to my advisor, Dr. Ahmad Sameh for his guidance, constructive criticism and gentle nature; to Dr. Sherif El kassas for his willingness to help and assist at any time; For Dr. Mikhail for accepting to be one of my internal examiners and for his help; to the entire computer science department for setting my career path, and for all the engineers who helped making the experiments possible.

Special thanks to my mother, father and Adel for their endless support and for being there for me at every moment during my thesis; and many thanks to engineer Ahmad Hameed for his guidance during my implementation.

Table of Contents

List of Figures

List of Tables

Chapter 1 – Introduction

1.1 Motivation

Nowadays we are living in the information era that requires the existence of information anywhere, anytime and with minimal overhead. That is why mobile agents, because of their distinguished features that resemble distributed systems, are becoming a common tool to transport information all over the net.

The spread of information all over the globe is becoming more and more risky due to the existence of malicious attacks that threats the security and privacy of associations and corporations whose data are wide spread through the Internet. That is why the emergence of protection mechanisms to protect exchanging information through the Internet is becoming a necessity for any organization in order to be able to face competitors and to prevent malicious attacks on its data.

Threats of malicious attacks come in many forms. Organizations can fear the threat of malicious agents attacks that can spy their data and can even corrupt it. They can also fear the attack in their own agents who roam the Internet searching for information that might be critical. Serious attempts are done to find a comprehensive and trustworthy protection mechanism to protect data from malicious attacks. Protection mechanisms were found for some areas of data protection like protecting the hosts from malicious agent attacks or protecting hosts from other hosts attacks, but until now, no fully comprehensive solution was found to solve the problem of protecting agents from malicious host attacks. Unfortunately all existing solutions that range from legal protection, to hardware solutions, partitioning, replication and voting, components, self-authentication, migration history. Solutions also included using audit logs, read-only state, append only logs, encrypted algorithms, digital signatures, partial result authentication codes, code mess-up, limited life time of code and data as well as time limited blackbox security, are only partial solutions that do not provide a guaranteed protection for the agent. That is why, until now the issue of agent protection against malicious host attacks remains an open question with no comprehensive solution.

What makes the problem more critical is the difficulty in assessing security in general. Everyday malicious attackers discover more advanced ways to steal data and access critical information illegally. There are two ways of assessing security. A practical one that involves experimentations and results, and a formal one that involves theoretical analysis.

1.2 Statement of the problem

This thesis extends and uses previous research conducted on mobile agents security with the focus on agent protection to solve the problem of malicious host attacks. This is achieved by proposing a three-tier solution. It is based on combining three security models which are: code mess-up, encryption and timing. This has involved implementing a new mess-up algorithm based on the idea of altering the string and numerical values existing in the agent's code in addition to inserting dummy code to obfuscate the original agent's code.

Each of the existing partial solutions by its own does not provide a full protection for the agent, this is in addition to the fact that not all the existing methods are fully implemented. For instance, the code mess-up algorithm proposed in **[Hoh97]** is not implemented until now.

For testing purposes, the thesis developed a flower-buying mobile agent. The two approaches of security assessment were tested in this thesis. The practical approach is used in the form of experimentations to test the code mess-up and timing techniques, and a theoretical assessment was presented in chapter seven to assess the encryption technique.

1.3 Thesis Structure

This thesis is divided into eleven chapters. Chapter one is a general introduction. Chapter two introduces the concept of agents to the reader, then it proceeds by focusing on mobile agents and their advantages, applications and some of their features. Then, chapter three talks about the issue of security in mobile agent systems, explaining the possible types of security threats that can face both agents and hosts, and focusing on possible host attacks on agents.

Chapters four and five talk about the partial solutions that currently exist to protect agents from host attacks. Chapter four contains a survey of some of the most powerful existing encryption algorithm, and chapter five continues to talk about the rest of the partial solutions.

Then, chapter six starts talking about the proposed solutions, and chapter seven proceeds by an analysis of testing and security assessment. Chapter eight shows a brief survey of existing languages for mobile code and chapter nine proceeds by a survey of the Concordia mobile agent system.

Finally, chapter ten shows the results and analysis and chapter eleven concludes by talking about problems faced while developing this thesis and possible future work.

Chapter 2 – What are Agents?

2.1 Introduction

There is little conflict among researchers about exactly what they consider an agent to be. One can define agents to be computer programs that simulate human relationship, by doing something that another person could otherwise do for you. In other words, we can say that an agent is anybody who acts on behalf or in the interest of somebody else **[Ans95].** Agents help users perform tasks, possibly by maintaining persistent state and communicating with their owners, with other agents or with their environment in general. They also relieve users from the routine work and facilitate complex tasks.

2.2 What are mobile agents?

Mobility is a common characteristic of many agents. Mobile agents constitute a new approach to the architecture and implementation of distributed systems that is promising to structure problems of such systems. Mobile agents are programs, typically written in script languages, that can move through a network under their own control, migrating from host to host and interacting with other agents and resources on each **[Dal98].** Each mobile agent consists of code, data and the current Execution State that is transported with the agent.

An agent can suspend execution at any point; transport its code and state to another machine, then resume execution on the new machine. Agents that migrate have the ability to access resources locally, eliminating the need to transfer data across the network. Thus the agent can access the resource efficiently even if the network conditions are poor or the resource has a low-level interface.

Almost any distributed application is a potential mobile-agent application. Other applications include active documents as well as electronic commerce. Mobile agent technology is regarded as an advance compared to a rigid client/server model. It tends to move us towards a model in which programs communicate as peers and act as either clients or servers depending on their current needs. Furthermore, In a networking context, the mobile agent visits several servers containing scattered information and provides the results in a package.

Security is a significant concern with mobile agent-based computing, as a server receiving a mobile agent for execution may require strong assurances about the agent's intentions.

A mobile agent should be able to execute on any machine within a network, regardless of the processor type or operating system. Also, the agent should not have to be installed on every machine that the agent could potentially visit; rather, it should move with the agent's data automatically. Hence, mobile agents have the potential to provide a convenient, efficient and robust programming paradigm for distributed applications specially when partially connected computers are involved. A mobile agent can for example, migrate off a laptop and roam the Internet to gather information for its user. It can access the needed resources efficiently since it moves to their network location rather than transforming all the data and requests across the laptop connection **[Dal98]**. In addition to that, it is worth mentioning here that since the agent is not always in direct contact with the laptop, it will not be affected by unexpected or sudden loss of connection, and can continue its task even if the user powers down or disconnects from the network. And if any disconnection occurred, when the user reconnects, the agent returns to the laptop with the results of its trip.

The literature on agents defines two different approaches to agent mobility: weak mobility and strong mobility. The weak mobility is restricted. It only allows the agent to migrate once to another agent platform. When it finishes its task, the result of its execution is either sent directly to the agent owner in form of a message or the agent itself is returned to its owner who extracts the results from the agent. This approach is considered to be much simpler. This is because the agent platform does not have to provide the current execution state, and because this approach has a simpler trust model since any damage that occurs by the agent on the agent platform or vice versa can be easily detected and attributed to the other agent at once. On the other hand, strong mobility is not that restricted. It allows agents to visit as many agent platforms as possible to accomplish the desired task. The problem with this approach lies in the trust issue. If damage occurs, it will be difficult to attribute this damage to a particular party **[Dal98]**.

2.3 Why Mobile Agents?

The current competitive business environment requires the availability of information anytime, anywhere, and independent of specific computing or communication devices. Mobile agent applications perform data access, decision execution, and notification. They provide the user with information that answers their need, whenever and wherever they need them.

Actually, mobile agents are very suitable for implementing distributed applications. They help minimizing network traffic, by locating the program with its needed data on one machine, and thus saving the bandwidth **[Dal98]**. Mobile agents are also very useful for balancing network load and in network management applications.

2.4 Advantages of Mobile Agent Architecture

Mobile agent architectures have several advantages over client-server systems and distributed object systems. These advantages include:

1. The mobile agent performs much of the processing required at the server where the local bandwidth is high, and thus, it reduces the amount of network bandwidth consumed and thereby increases overall performance.
2. The mobile agent help overcoming network latency **[Lan99]**. For instance, in critical real-time systems, there is a need for real-time response to changes in their environments. Controlling such systems through a “factory network” of a large

size involves significant latencies that is not acceptable. Mobile agents can be dispatched from a central controller and execute the controller's directions locally.

3. The mobile agent operates independently of the application from which the agent was invoked. The client application does not need to wait for the results because mobile agents operate asynchronously.
4. Using mobile agents allows injecting new functionality to the system at run time.
5. Mobile agents allow for easy development, testing and deployment of distributed applications since they hide the communication channels.
6. Mobile agents also eliminate the need to detect and handle network failure except during migration.
7. They eliminate the need for pre-installing application-specific software at each site.
8. They can dynamically distribute and re-distribute themselves throughout the network hosts to obtain the optimal configuration for solving a particular problem **[Lan99]**.
9. They increase efficiency because they can help to reduce intermediate data transfer by moving across the network to the location where the resource resides.
10. Once the agent is launched, it is no longer dependent on the system that launched it and therefore it is not affected by any failure that might occur to this system **[Dal98]**. And hence, mobile agents are persistent.

2.5 Issues Against Mobile Agents

In spite of the advantages of mobile agents mentioned above, some people argue that there are severe problems with mobile agents. These arguments include:

Authentication: How can an agent be assured that a given mobile agent is who it claims to be and represents whom it claims to represent?

Security: How can remote sites avoid being exploited by mobile agents and prevent their integrity being violated **[Bau97]**. This may include executing illegal instructions, accessing and modifying unauthorized information, and monopolizing resources. Generally, all mobile agents roaming the net in pursuit of information are exposed to a number of security threats.

Secrecy: How to maintain and preserve the internal representation of a mobile agent while it is in transit and while it is executing on remote sites? In this two cases, the contents of the mobile agents can be inspected and copied, which can be a serious problem specially if the data was of a private nature.

Payment for services: How are mobile agents to pay for the services that they require? **[Dal98]** This includes making sure that the agent can pay fully the amount and that the service paid for is satisfactory for the payee.

2.6 Applications of Mobile Agents

Mobile agents can be useful in many applications. A very common application for mobile agents is **information retrieval**. For instance, information retrieval on the network can be supported much more efficiently if an agent representing a query can move to the place where the data are actually stored, rather than having to move all the data across the network. Hence, a mobile agent reads the information locally and may even do some filtering or pre-processing, and then it moves back to its home place and returns the results to its owner, and thus increasing system's efficiency **[Ans95]**. Techniques such as *semantic routing,* dispatching a query according to where it is most likely to be answered, rather than according to predetermined addressing information, can facilitate a system's usage.

Another area of application for mobile agents is **network management.** In big networks containing hundreds of connected computers, both operation monitoring and fault detection is very difficult. Therefore it is feasible to use mobile agents to keep tabs on the system, discover possible trouble spots or performance bottlenecks and bring them to the attention of the maintainers **[Ans95]**. Also, current computer networks are based on the exchange of passive data packets. In active networks, each data packet is replaced by an active mobile code packet that carries data but also the instructions telling a network node how to process this packet. This increases the flexibility of the network **[Sat97].**

A third application of mobile agents is **electronic commerce.** Business through the Internet is becoming a widespread phenomenon. Mobile agents can help locate the cheapest offerings, negotiate deals or even conclude business transactions on behalf of its owners. Mobile agents can be a way for customers to stay on top of the commercial developments. A typical application of mobile agents in the domain of electronic commerce is an agent that roams the Internet searching for some good or service on behalf of its owner. The owner of such an agent normally has to configure it with all the relevant information about the desired good or service. Also, the owner has to clarify the constraints that define under which conditions an offer from a specific provider might be accepted, and a list of some suggested providers to whom the agent can visit in its journey of search. The agent’s itinerary is the path that defines the agent’s journey between the providers **[Val96]**.

Mobile computing is another possible application of mobile agents. Portable computers become smaller and more powerful, but wireless access to a fixed information infrastructure is likely to stay slow due to restrictions on radio transmissions **[Ans95]**. Also, to minimize power consumption and transmission costs, users will not want to remain on-line while some complicated query is handled on their behalf by the fixed computing resources. Here comes the role of mobile agents. Users simply submit mobile agents that embody their queries and log off, waiting for the agents to deposit their results ready to be picked up at a later time.

Actually, none of the above applications absolutely requires mobile agents, because most of them can be handled by stationary programs and remote procedure calls (RPC's) **[Dal98]**. However, using these old ways will increase the system and network load and may reduce user convenience. Whereas using mobile agents can reduce communication costs, allow for better support of asynchronous interactions, and provide enhanced flexibility in the process of software distribution. Hence, the agent architecture provides a flexible alternative to client/server and distributed object architectures.

2.7 Alternatives to Mobile Agents

Actually, there are several alternatives to mobile agents that can perform the same tasks mobile agents perform, but this will be done at a price of increased system and network load and possibly at the inconvenience of users. These alternatives include messaging, simple datagrams, sockets, remote procedure call, and conversations. These alternatives are either synchronous like RPC, or asynchronous like messaging. Mobile agents employ messaging frameworks for transport and thus are asynchronous. In both synchronous and asynchronous alternatives, the client and server exchange data that is to be processed by specific procedures at the remote CPU. Neither party specifies how data are to be processed; each has implicit knowledge of the capabilities of the remote procedures. This contrasts with mobile agents that communicate both data and their own procedures and that exploit procedures resident at the client or server.

2.8 Summary

As shown in chapter two, agents are autonomous, goal-oriented software programs that perform tasks on behalf of their creators. A great advantage of agents is that they can communicate with other agents and with the environment they are headed to. Some agents are mobile. This type of agents can roam the Internet from host to host, performing a specific task on behalf of their users. This mobility privilege has a great impact on the network, as well as the agent's launchers. For instance, moving an agent from one host to another in order to perform calculations or to get data will increase network efficiency. This is due to the fact that mobile agents help reducing data transfer from one host to another by moving to the location where the data and resources needed physically exist.

Due to the flexibility of mobile agents, and their wide spread usage, they are becoming very useful in numerous applications including information retrieval, network management, electronic commerce, and mobile computing.

Chapter three is going to tackle the issue of security in mobile agent systems and how it is a very crucial issue in all mobile agent applications.

Chapter 3 - Security in Mobile Agent Systems

3.1 Introduction

In spite of the numerous advantages of mobile agents, they produce a lot of additional overhead in terms of performance and security. For instance, mobile agents have to be protected from mobile agent hosts and vice versa. Beside, there are several security issues to be considered in mobile agent-based computing. These issues include:

1. Authentication of the user, who is the sender of the mobile agent, by the server and authentication of the server or agent execution environment **[Dav95]**. The server may wish to be able to authenticate the user uniquely or it may be satisfied to know simply that the user belongs to a group of authorized users. Some servers may not require any authentication at all if they do not have any protected information or functions. The agent itself may convey the authentication information or it might be transmitted separately. The outcome of the authentication process is that the user/agent knows the identity of the server/agent execution environment and the server/agent execution environment knows the identity of the user/agent. This authentication is based only on header information transmitted with the agent; the server still has no idea what the agent wants to do.
2. Another security issue is the determination of whether the user has authorization to execute agents at the server and which functions may be used. Also, determining whether the agent will attempt to infect the server, deny service to other agents or otherwise attempt to do harm to the server or other agents is a very important issue. The server's agent execution environment will re-constitute the agent into an executable. But before the server dispatches this executable, it may wish to examine the agent's code to see what resources it will access **[Dav95]**. Following successful completion of this test, the agent execution environment will then permit agent access to server resources, depending on the privileges of the user.

3. A third and very **important** security issue is the determination of the agent's ability or willingness to pay for services provided by the server, unless they are free. Since the agent is consuming at least computational resources at the server, and may be performing transactions for goods, the user also requires considerable assurance that his or her liability is limited. A method to solve this issue is using electronic currency called *Teleclicks.* During the execution of the agent by the server, the server is entitled to transfer currency units from the agent to the agent execution environment as a form of payment. The user's liability is limited to the quantity of currency, which the agent was issued by the client. In some environments, the agent which exhausts its currency is killed. But on the other hand, for security reasons, the user also requires assurance that the agent execution environment cannot fake the quantity of currency transferred and that the server is truly providing the contracted services.

3.2 Security in Existing Mobile Agent Systems

Mobile agent systems are the environments that support agents. There are many mobile agent systems currently existing and functioning. This section covers only some of these systems including: Telescript, Aglets, Agent TCL and Tacoma. This does not mean that these are the only available mobile agent systems; rather, there are many other available systems, some of them are for commercial purposes while others are only for research.

3.2.1 Telescript

Telescript is a commercial product developed by General Magic Incorporated to support mobile agents for electronic marketplaces **[Val96]**. It consists of three major components: the architecture, the language and the development environment. Major security concerns arose in the Telescript environment like:

- *User Authentication:* i.e. who is responsible for a specific script and who is accountable for any charges it runs up.

- *User Authorization:* i.e. can a particular user executes a particular script. And which operations on a particular object can a user invoke?
- *Trojan Horses:* Since Telescript does inter-process communication by passing objects around, and the only way to use an object is to invoke an operation on it, there must be a way to know whether or not an arbitrary class contains malicious code. The threat of malicious code might include communicating private information, modifying data, impersonating another user, and denial of service attacks.

In Telescript, when an agent wishes to migrate to a remote destination, it issues a "*go*" command with a ticket argument indicating the required destination and the time of completion. When the agent successfully reaches its destination, the agent managing the destination must authenticate the mobile agent by determining its authority. If access is granted, then the agent enters their chosen place and can interact with the resources and other agents it finds there. If access is not granted, the agent will move to a purgatory where they cannot access any resources. This is clearly shown in figure 3.1. The figure shows that if the engine accepted the mobile agent, it will be granted access to the destination's resources, otherwise it will go to a purgatory. For further security restrictions, an agent is granted a permit when it travels between places or regions. This permit details the capabilities of the agent and the agent has to agree to the restrictions imposed by the permit before it can enter a given place. Also, this permit can deny an agent access to particular resources within a place, and can also limit the amount of usage an agent can make of a resource through the allocation of an allowance that can be expressed in terms of time, size and computation. When an agent exceeds its allowance for a resource, it is destroyed.

The Telescript language is an object oriented wrapping language. The interfaces to the stationary applications are written in C or C++, and the communication interface is written in the Telescript language. The language itself is compiled into byte-code for portability and is executed by engines, which contain the interpreter and represent the runtime environment **[Dal98]**. A further discussion of mobile agent languages will be presented in chapter eight.

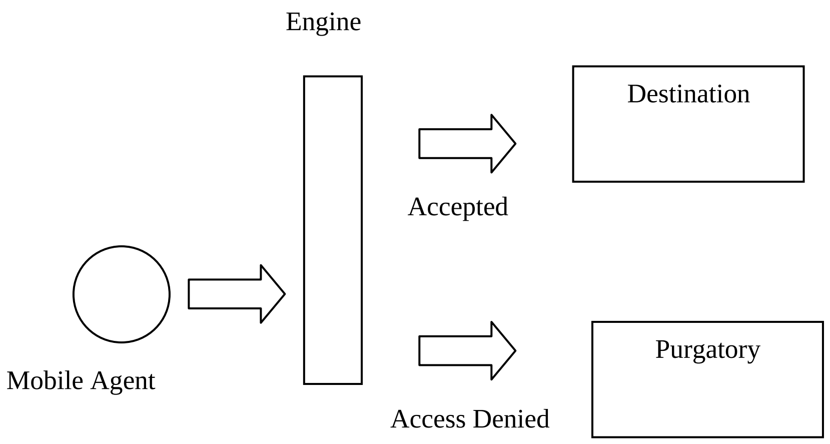


Fig. 3.1 - The Telescript Authentication Process

3.3.2 Aglets

Aglets are Java objects that can move from one host on the Internet to another.

An aglet is a Java-based autonomous software agent. It can halt itself, ship itself to another computer on the network, and continue execution at the new computer, and it continues execution where it left off. When an aglet migrates, it carries its code and data with it. An aglet is both autonomous and reactive; Autonomous because it runs in its own thread of execution after arriving at a host, and reactive because of its ability to respond to its incoming messages.

Like an applet, the class files for an aglet can migrate across a network. An aglet interacts with its environment through an AgletContext object **[Dal98]**. To interact with each other, aglets do not invoke each other's methods directly. Instead, they go through AgletProxy objects that serve as aglet representatives. The AgletProxy class allows Aglets to request other Aglets to take actions. The Aglet that has been requested to take action can comply, refuse to comply, or decide to comply later. The Agletproxy class contains methods that allow aglets to request other aglets to take actions, such as dispatch(), clone(), deactivate(), and dispose(). The aglet that has been requested to take action has the freedom to either comply, refuse to comply, or decide to comply later.

In order to interact with other aglets, an aglet must go through a proxy object. Even if both interacting aglets reside in the same aglet host, they are not allowed to interact directly with each other to prevent access to the aglet's callback and initialization methods, which are public **[Ven97]**.

3.2.3 Tacoma

The Tacoma project is concerned mainly with providing operating system support for agents. Agents are considered to be the computational unit of the system and have three storage mechanisms:

Folders: they are the essential unit of data that is accessible by an agent.

Filing Cabinets: are permanent data repositories that can contain folders. Agents can leave information at various sites by placing the data in a folder and then depositing this folder within a filing cabinet. They are stationary.

Briefcases: Are containers that agents carry with them and can hold system folders. They are not stationary.

TACOMA allows agents to charge for the use of their services through adopting electronic cash (ECU). Once an agent has spent all of its units of electronic cash, it cannot access any more resources. It uses the replication and voting mechanism (see below) to protect its agents from possible attacks by malicious hosts. Figure 3.2 shows a diagram of the Tacoma charging of services system. If the agent has enough cash, it will be able to access resources, otherwise the agent will no longer be granted any system resources.

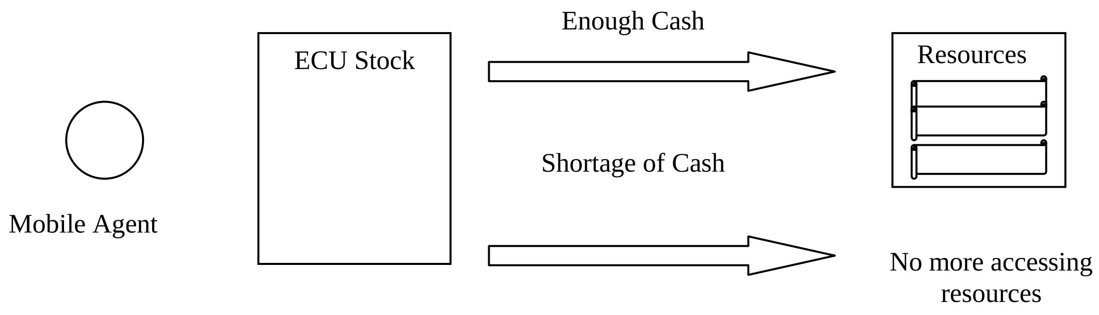


Fig. 3.2 - Tacoma-Charging of services

3.2.4 Agent TCL

Agent TCL is a system for supporting mobile agents developed at Dartmouth College, USA **[Dal98]**. Agents provide all services available within the system. The architecture of Agent TCL is based upon the server model advocated by Telescript. At each TCL site, a server resides and handles the management of local agents and incoming mobile agents. The server also enforces security, and allows agents to address each other locally by providing a hierarchical name space in which agents can be referenced.

Agents move between sites by issuing the command *agent_jump*, which transfers the program context of the agent to a destination site where the server restarts it at the instruction after the *agent_jump* command. The method in which the agent is transported is determined by the transport system advocated by the local site server.

Agent’s execution is handled by an interpreter appropriate to the source language of the mobile agent. In agent TCL, the interpreter of TCL was extended to support three extra modules:

- A security module to prevent agents performing malicious actions.
- A server API module that allows communication with the server to facilitate migration, communication and check-pointing.
- A state capture module to capture and restore the state component of an agent when it wishes to migrate to a new location.

There are two interpreters provided for the language by safe TCL:

- A trusted interpreter in which all of the TCL commands are made available to the agent **[Dal98].**
- An untrusted interpreter in which the more dangerous commands like opening files, creating network connections are removed.

The *agent-meet* command is another important command that is used to initiate a communication with another agent. The *agent_accept* command completes and synchronizes the two agents. The *agent_send, and agent_receive* commands allow the agent to communicate asynchronously.

3.3 Types of Security Threats in the Mobile Agents' World

Neither the agent nor the machines are trustworthy in an open network environment. A malicious agent might try to access or destroy privileged information or consume more than its share of some resource. Also, a malicious host might try to pull sensitive information out of the agent or change the behavior of the agent by removing, modifying or adding to its data and code. That is why, many security considerations have to be put in mind before sending an agent to another host.

3.4 Types of Security Concerns

3.4.1 Protecting the machine

The machine should authenticate the agent's owner, assign resource limits based on this authentication, and prevent any violation of the resource limits. It should also prevent both the theft and damage of sensitive information and denial of service attacks. **[Gra98]**

Authentication: verifying the identity of the agent's owner. Agent TCL server distinguishes between two kinds of agents: owned and anonymous. An owned agent is an agent whose owner could be authenticated and is on the server's list of authorized users. An anonymous agent is an agent whose owner could not be authenticated if it is not on the server's list of authorized users. A server can accept or reject anonymous agents, and if it accepts them it gives them an extremely restrictive set of resource limits.

RSA public-key cryptography is used to authenticate an agent's owner. Each owner and machine in Agent TCL has a public-private key pair **[Gra98]**. The server can authenticate the owner if the agent is digitally signed with the owner's public key or the agent is digitally signed with the sending machine's key, and the server trusts the sending machine and the sending machine was able to authenticate the owner itself.

Authorization and enforcement: Authorization involves assigning access restrictions to the agent; whereas enforcement ensures that the agent does not violate these restrictions.

3.4.2 Protecting other agents

The security of agents against other malicious agents is another security issue in mobile agent systems. Normally, what is required is that an agent should not be able to interfere with another agent or steal that agent's resources. The possible agent-agent attacks include manipulating code and data by accessing the code and data areas of the attacked agent as well as cheating and denial of service. This area deals with security issues that were already dealt with in traditional distributed systems, that is why there are already mechanisms that are able to overcome such attacks, these methods include:

- Using an agent language that prevents other agents from having physical access to private data like Java, or using isolated address spaces.
- Using digital signatures to authenticate the agents **[Hoh97]**.

3.4.3 Protecting a group of machines

An agent might consume excessive resources in the network as a whole even if it consumes few resources at each machine. E.g. an agent that roams forever, or one that creates two child agents on different machines and each create two other children and so on.

If the machines were under the same administrative control, protecting them is straightforward, by assigning an agent a maximum resource allowance when it first enters the machine group. The allowance and the amount that the agent has used so far is propagated along with the agent as it migrates. If the agent exceeds the group allowance it is terminated.

If the machines were not under the same administrative control, this becomes more complex. In this case, the solution is to standardize on a maximum number of migrations and children per agent, and each agent carries along a count of migrations and children so far, and each agent server increments this count when appropriate. Or make the agents pay for their resource usage with cryptographically protected electronic cash **[Gra98]**. And so each engine is given finite cash from its owner's currency supply. When run out of currency the agent and its children come back to the owner for a refill of supplies or termination.

3.4.4 Security between agents and hosts

Both the agent and the environment it is running on impose a mutual threat on each other. The agent may seek to obtain data from the site it is running on or damage the site in some way. In the same time, the execution environment may have been designed to steal data or corrupt agents that migrate to it. The two major security threats between agents and hosts can be summarized into the two following points:

3.4.4.1 Protecting the host from malicious agents

The attacks of agents are usually the same whether they are oriented to other agents or to hosts, and this include the physical manipulation of data, cheating, masking and denial-of-service. The only difference here is the greater effect of the hosts in controlling the agent's execution. Therefore attacking a host might be much more critical and harmful than attacking a single agent. The great control of hosts on the agents that run in it can reach to the level of stopping suspicious agents at any time.

Again, this attack has mechanisms to overcome it. These mechanisms include:

- Using secure languages or isolated address spaces.
- Using digital signatures or other cryptographic techniques to authenticate the agent.
- Using resource control mechanisms like limited resource “accounts” and runtime restrictions **[Hoh97]**.

3.4.4.2 Protecting the agent from malicious hosts

This problem is the main concern of this thesis. Normally, any host that executes an agent is allowed to see the agent’s internal workings and has full control of the agent. Thus it is so easy for an attacker who has time, to analyze the inner workings of any agent and to change its code and steal its information. But this can easily occur only if the attacker has enough time to do that. Normally, the cost of mounting the hostile host attacks is very high. This is because modifying an agent system at runtime is not easy and requires time and effort for realizing how to attack the agent’ s code.

Protecting agents against malicious host attacks is the most difficult security problem. And apart from hardware and organizational solutions, there are no technical approaches to solve this problem so far **[Dal98]**.

Our main issue here is that the host should not be able to tamper with an agent or pull sensitive information out of the agent without the agent’s cooperation. But this is difficult in the absence of trusted and tamper-resistant hardware support. We must detect the tampering as soon as an agent migrates from a malicious machine to an honest one and then terminate or fix the agent if tampering has occurred.

Yee presented a simple example of how such an attack can be harmful **[Yee97].** Lets take as an example the standard airfare agent scenario. A person needs to travel to Washington D.C. to attend a meeting, so he sends an agent to visit servers at all the airlines to query their databases to get the cheaper airfare from the specified source to the specified destination given the trip timing, seat preferences and routing constraints. Then, assume that one of the airlines, called Fly-By-Night airlines, runs a server, www.flybynight.com. And assume that the agent’s code was recognized and brainwashed such that all the information gathered about all other airlines and their prices were modified so that it ends up recommending a specific flight by the Fly-By-Night airlines. This is a possible host attack that can brainwash the agent's data and changes its results.

In order to summarize the possible security threats in mobile agent systems, Fritz Hohl suggested the following diagram: **[Hoh97]**

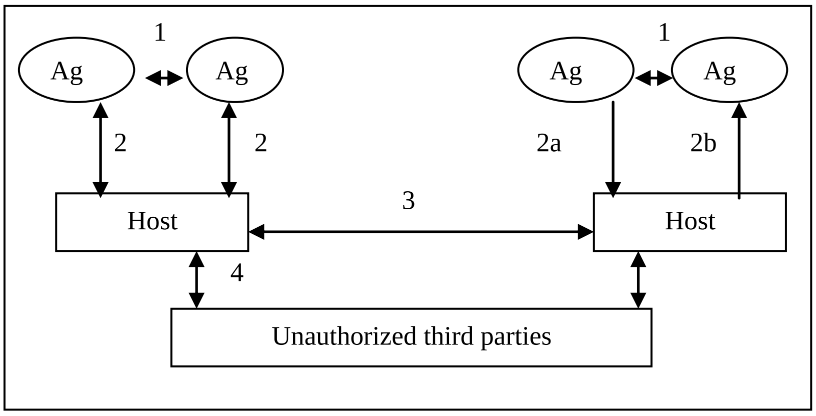


Fig. 3.3 Security areas of mobile agent systems

The figure defines four main security areas that differ in the parties involved (agents and hosts). These four areas represent the possible attacks between the parties.

1. Security between two agents
2. Security between agents and hosts
3. Security between hosts
4. Security between hosts and unauthorized third parties.

3.4.5 Possible Host Attacks

3.4.5.1 Normal Routing

The host holds the agent longer than necessary which harms a time-critical agent, charges the agent extra money, or modifies the agent's code or state which causes the agent to perform some work on behalf of the malicious host.

3.4.5.2 Rerouting

Rerouting the agent to a different destination machine. Or preventing the agent from migrating at all.

3.4.5.3 Spying out code

Most of the hosts see all of the code of the agent as they execute it. And it might happen that the code can be seen even before the execution time. Also, if the agent's code was constructed using standard building blocks and libraries in order to reduce code migration cost, the details of these blocks will be available for the host. This knowledge of code allows the host to know more about the execution strategy of the agent, the exact physical structure of code and data in the host's memory and may be the knowledge of parts of the agent's data **[San97]**.

3.4.5.4 Spying out data

When this threat takes place, it does not leave any trace or indication of whether the host read the data or not. This is very critical for agents carrying secret data that requires extreme privacy.

3.4.5.5 Spying out control flow

The next execution step in the agent's code can be simply recognized by the host if it knows the entire code of the agent and its data. And even if the data was protected, the control flow cannot be fully protected. And thus, the host can easily deduce more information about the state of the agent.

3.4.5.6 Manipulation of code

If the host can read the code and access the agent's code memory, it will be easy for it to modify the agent's program. This modification may be by inserting any virus, which last permanently, or by altering the behavior of the agent. Permanent code manipulation is impossible if one can cryptographically sign the code of the agent. If the code does not change over the execution time, this will be great since the owner can sign the agent at start time. If the owner managed to protect the code so that the host does not know that each line of code does exactly and where this line is stored, temporary code manipulation attacks would be impossible.

3.4.5.7 Manipulation of data

When the host knows the physical location of data in the memory it can modify it specially if it knew the semantics of the data elements. But if the data elements will not be modified by correct code during the agent's visit to the host, this data can be encrypted and thus protected from the host.

3.4.5.8 Manipulation of control flow

Manipulating the control flow allows the host to control the behavior of the agent even if the host does not have access to the agent's data.

3.4.5.9 Incorrect execution of code

The host can alter the way it executes the agent's code even without changing the code itself.

3.4.5.10 Masquerading

Normally, the host sends the agent to a receiver host to make sure of the agent's identity. This receiving host can be malicious and pretends to be the correct receiver host, and can thus proceeds by attacking the agent's code, data or flow control.

3.4.5.11 Denial of execution

The host can refuse to execute the agent. Or deny its execution. But constructing a protocol to force the host to proof the execution of each agent can solve this problem **[Hoh98]**. For instance marking hosts that are unwilling to execute agents as bad hosts, and instructing agents not to visit any bad hosts can do this.

3.5 Summary

Chapter three stressed three major issues that have to be considered in mobile computing. These issues include user authentication, user authorization, and agent's payment method.

These three issues are tackled differently in various mobile agent systems. The table below summarizes the methods that major mobile agent systems use to handle these three issues.

	Authentication	**Authorization**	**Payment**
Telescript	• Agent managing destination authenticates mobile agent depending on its authority • If access is granted, agent enters, otherwise agents goes to purgatory.	• Permits to detail the agent’s capabilities. • Agent has to agree to the restrictions imposed by the permit before it can enter a given place.	• Agents are allocated allowances that can be expressed in terms of time, size and computation. When an agent exceeds its allowance for a resource, it is destroyed.
Aglets	• Aglets can invoke each other’s methods using an Agletproxy object that serve as aglet representatives.	• An aglet is not authorized to invoke other aglet’s functions except through a proxy object.	• Depends from one aglet to another
Tacoma	• Replication and voting.		• Agents pay for their services through Electronic cash. Once an agent has spent all of it ECU it cannot access any resources.

Agent TCL	• At each TCL site a server resides to manage local and incoming agents. • Agents are allowed to address each other locally through a hierarchical name space in which agents can be referenced.	• An agent-meet command initiates a communication between one agent to another. • Agent_accept command completes and synchronized two agents Agent_send & agent_receive for asynchronous communication between agents.	

Table 3.1 – Comparing Mobile Agent Systems

In addition to that, different types of possible host attacks were discussed in chapter three. These attacks are summarized in the following table:

Threat	**Description**
Normal Routing	Host can hold agent longer than necessary, which harms time-critical agents. Host can charge agent extra money, or modify agent's code or state.
Rerouting	Host can reroute the agent to a different unintended destination.
Spying out code	Host might spy the agent's execution strategy as well as its exact physical structure of code.

Spying out data	Host might attempt to read the agent’s data. This is very critical for agents carrying secret data.
Spying out control flow	Host can easily recognize the next execution step of the agent, and hence, can deduce the next state of the agent.
Manipulation of code	Host can modify the agent’s program by altering its behavior or inserting a virus.
Manipulation of data	When the host knows the physical location of data in the memory and its semantics, it can modify it.
Manipulation of control flow	The host can control the behavior of the agent even if it does not have access to the agent’s data.
Incorrect execution of code	Host can alter the way it executes the agent.
Masquerading	A host can pretend to be the receiver host of a particular agent and attacks its code and data.
Denial of execution	The host can refuse the execution of an agent or deny its execution.

Table 3.2 – Possible Host Attacks

Chapter 4 - Encryption/Decryption

4.1 Introduction

There exist a common claim that mobile code in general can never be protected without using special hardware, simply because the code has to be executed by the hosting system. Actually, this makes sense because any cleartext data can be read and changed, cleartext programs can be manipulated, and cleartext messages can be faked. That is why the need of cryptographic solutions and encryption arose **[Sat97]**.

Encryption is the process of disguising a message to hide its substance. An encrypted message is called cyphertext. Decryption is the process of turning cyphertext back into plaintext. Cryptanalysis is the science of recovering the plaintext of a message without access to the key. If the cryptanalysis was intended then it will be considered as an attack. Cryptanalysis assumes that the secrecy resides mainly in the key. Figure 4.1 shows the process of encrypting a plaintext to be cyphertext and of decrypting a cyphertext to obtain the original plaintext.

There are four main types of cryptanalytic attacks. Each of them assumes that the cryptanalyst knows the encryption algorithm used:

1. Cyphertext-only attack: The cryptanalyst job is to recover the plaintext of as many messages as possible or deduce the key used to encrypt the messages in order to decrypt other messages encrypted with the same keys.
2. Known-plaintext attack: the cryptanalyst has access to both the cyphertext and plaintext of messages.
3. Chosen-plaintext attack: the cryptanalyst has access to cyphertext, plaintext and he chooses the plaintext that gets encrypted as well.
4. Adaptive-chosen-plaintext attack: this is a special type of the chosen-plaintext attack. Here the cryptanalyst can modify his choice of plaintext that is encrypted based on the results of previous encryption **[Sch96]**.

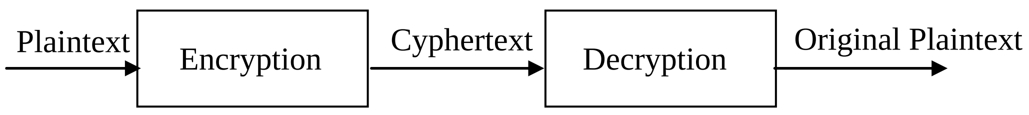


Fig. 4.1 The Encryption / Decryption process

Actually, cryptography is only a part of security; it involves the mathematics of making a system secure. Our challenge in studying cryptography is to find answers to the following problems:

- Can a mobile agent protect itself against tampering by a malicious host? (code and execution integrity).
- Can a mobile agent remotely sign a document without disclosing the user's private key? (computing with secrets in public)
- Can a mobile agent conceal the program it wants to have executed? (code privacy) **[Sat97].**

Performing cryptographic protocols enforces the need of protecting critical elements and processing them inside a secure execution environment. It is very important to keep the key used for generating signatures secret. Normally, a secure execution environment includes something called the computing base that provides the physical storage and the execution support. Maintaining the trustworthiness of the computing base is very important.

4.2 How to Choose an Algorithm?

Normally, the designer of a secure system has to think about every possible means of attack to his/her system and tries to protect his/her system against those possible attacks. But in many cases, cryptanalyts and attackers do find some holes through which they can attack the designed system. That is why, not all people who want to protect their systems design their own algorithms, rather, when it comes to choosing and evaluating algorithms, people have several alternatives including:

1. Choosing a published algorithm, believing that published algorithms have been well tested and no body have broken them. And this is a reasonable choice.
2. Trusting a specific manufacturer, believing that this manufacturer will never risk his name and reputation by selling equipment or software with insecure algorithms. But this is not a reliable option because may be the manufacturer thinks that his equipment and software are secured but actually they can be attacked.
3. Trusting private consultants or the government believing that they would never betray any of their customers.
4. Building their own security algorithms. But this is not a reliable solution at all because unless the security algorithm is heavily tested and published, it can never be fully reliable **[Sch96]**.

An encryption scheme is said to be semantically secure if, given a public key $\mathbf{P_k}$, a cyphertext **c,** and two possible plaintexts x1, x2, it is infeasible to determine if **c** is an encryption of x1 or an encryption of x2 even if extra help in the form of the ability to ask for the decryption of some chosen cyphertexts is provided. Also, it should be infeasible to derive any partial information on the encrypted plaintext given its cyphertext **[Hal99]**.

4.2.1 What are block ciphers?

They are symmetric algorithms that operate on the plaintext in groups of bits called blocks. A typical block size is 64 bits, which is large enough to preclude analysis and small enough to be workable. A symmetric algorithm is an algorithm where the encryption key can be calculated from the decryption key and vice versa. They require that the sender and receiver agree on the same key before they can communicate securely. Hence, the security of block ciphers rests on the key, and as long as the communication remains secret, the key remains secret **[Sch96]**.

4.2.2 Cryptographic Protocols

In order for a communication to take place between any two parties, a protocol must be established. A protocol is a set of standards that the communicating parties agree upon before starting their communication. Cryptographic protocols are divided into four major categories: Basic, intermediate, advanced and esoteric. In this thesis, we will be concerned with the basic types of protocols.

4.2.2.1 Basic Cryptographic Protocols

A. *Key Exchange*

This method assumes that any two users on the network share a secret key with the key distribution center (which acts as a trusted arbitrator). These keys should be in place before the start of the protocol that does not take care of how to distribute these secret keys. For example, user A calls the key distribution center and requests a session key to communicate with user B. The key distribution center then generates a random session key and encrypts two copies of it, one in the first user's key and the other in the second user's key. Next, user A decrypts its copy of the session key and sends a copy to user B who decrypts it and then both users use this copy to communicate securely. This method depends primarily on the absolute security of the key distribution center. If this center was attacked by any means then this forms a great threat. Also, this center can become a bottleneck since it is involved in every key exchange, and so if it fails it will interrupt the whole system.

1. *<u>Key Exchange with Public-key cryptography:</u>* The two communicating parties use public-key cryptography to agree on a session key, and use that session key to encrypt the data. These signed public keys will be available on a database that makes the key-exchange protocol easier even if the receiver does not know the sender before hand.

2. *<u>Key Exchange with Digital Signatures:</u>* In this method, the key distribution center signs the public keys of both the sending and receiving parties. The signed keys include a signed certification of ownership. When the two communicating parties receive the keys, they verify the distribution center's signature, and then the key exchange protocol can proceed. This method is less risky than the previous one. If somebody managed to attack the key distribution center, all he can get is the private key. This key enables the attacker only to sign new keys, but it does not allow him to decrypt or read any message.

3. *Key and message Transmission:* In this method, the sender and receiver do not need to complete the key-exchange protocol before they exchange messages. This occurs using a database. For instance, the sender generates a random session key, K, and encrypts the message using K. Then it gets the receiver's public key from the database. The sender then encrypts the session key K with the receiver's public key and sends him both the encrypted message and the encrypted session key. Finally, the receiver decrypts the sender's session key using his private key and then decrypts the message using the decrypted session key.

4. *Key and Message Broadcast:* Using this method, the sender can send the encrypted message to several people. The sender generates a random session key K using which it encrypts the message. Then it gets the receiver's public keys from the database and encrypts the session k with each public key of each and every receiver. Then the sender broadcasts the encrypted message and all the encrypted keys to any body who cares to receive it. The receivers can decrypt the key K using their private keys, and they can decrypt the sender's message using K.

B. ***Authentication***

When a user logs into any type of terminals, it has to prove that it is who it claims to be to the host. This was done previously by passwords. Both the user and the host know this type of information that is requested each time the user needs to log onto the host.

1. *Authentication using one-way functions:* This method depends on the fact that the host does not need to know the passwords, rather it has to be able to differentiate valid from invalid passwords. Thus, instead of storing passwords, hosts store one-way functions

of the passwords. That is when the user sends his/her password to the host it performs a one-way function on this password and the host then compares the result of the one-way function to the value it previously stored. The host in this method does not need to store a table of everybody's valid password, and so it will no longer be threatened that someone would break into the host and steal its password list. The drawback of this method is that an attacker can compile a list of the most commonly used passwords and then operate on them with the one-way function and stores the results. Then it can steal an encrypted password file and compares it with his file of possible encrypted passwords and sees the matching.

An example authentication program that relies on a one-way function for its security is **SKEY**. In this program, the user enters a random number R. The computer then computes f (R), f(f (R)), f(f(f (R))), and so on, for about a hundred times. These numbers are called x1, x2…, x100. The computer then prints out this list of numbers for the user to keep. The computer then stores x101 in a login database's clear next to the sender's name. Then, when the user tries to login, he/she types the name and x100, the computer then calculates f(x100) and compares it with x101. If they match, the user is authenticated and the computer replaces x101 with x100 in the database. Thus, every time the user logs in, he/she crosses the last number from their list, and this way an attacker cannot track any sequence because each number is used only once and the function is one-way. And when the user is run out of numbers, he/she has to reinitialize the system again.

2. <u>Authentication using public-key Cryptography:</u> This method solves the drawback of the one-way function method. A file of every user's public key is kept by the host, and all users keep their private keys. First, the host sends the user a random string, then the user encrypts the string with his/her private key and sends it back to the host along with the user's name. Then the host looks up the user's name in its database and decrypts the message using that public key. If the decrypted string matches what the host sent the user in the first place then the host allows the user to use the system. In this method, no one else has access to the user's private key because the user never sends his/her private key over the transmission line.

C. ***Authentication & Key exchange:***

This method combines both authentication with key exchange. This solves the problem of how can a sender and a receiver exchange a secret key and at the same time they make sure that they are talking to one another not to a third malicious party.

Under this come several protocols including:

1. *Wide-Mouth Frog:* This protocol is very simple and uses a trusted server. Both the sender and receiver share a secret key with the trusted server. These keys are used for key distribution and not for encrypting the actual messages. The sender concatenates a timestamp, the receiver's name and a random session key and encrypts the whole message with the key it shares with the trusted server, and then it sends it to the server with its name. The server then decrypts the message and concatenates a new timestamp, the sender's name and the random session key, and encrypts the whole message with the key it shares with the receiver.
2. *Yahalom:* In this protocol also, both the sender and receiver share a secret key with the trusted server. The sender concatenates her name and a random number and sends it to the receiver, who concatenates the sender's name, its random number and its own random number and encrypts it with the key it shares with the trusted server. Then it sends this combination to the trusted server along with its name. The server generates two messages, one contains the receiver's name, a random session key, and the sender's and receiver's random number, and encrypts it with the key shared with the sender; and the other one consists of the sender's name, the random session key encrypted with the key shared with the receiver. The server then sends both messages to the sender. Finally, the sender decrypts the first message and extracts the key and confirm that the sender's random number has the same value as it did. And it sends the receiver the two messages. The

receiver decrypts the message encrypted with its key, extracts the key and confirms that its random number has the same value.

3. *Needham-Schroeder:* In this protocol, the sender sends a message to the trusted server consisting of its name, the receiver's name and a random number. The server generates a random session key and encrypts a message that consists of a random session key and the sender's name with the key it shares with the receiver. Then it encrypts the sender's random value, the receiver name, the key, and the encrypted message with the secret key it shares with the sender and sends the sender the encrypted message. The sender then decrypts the message to extract the key and sends the receiver the message. The receiver decrypts the message and extracts the key and generates another random value and encrypts the message with the key and sends it back to the sender who decrypts it and generates a random variable which consists of the receiver's random variable – 1 and encrypts the message again with the key and sends it back to the receiver. Finally the receiver decrypts the message and verifies that it is the random variable – 1. The problem with this protocol is that old session keys are valuable, and if an attacker gets access to an old key, it can launch a successful attack by recording the sender's message and once it gets the key it pretends to be the sender.

4. *Otway-Rees:* In this protocol the sender generates a message consisting of an index number, his/her name, the receiver's name and a random number and encrypts it with the key he/she shares with the trusted server. Then this method is sent to the receiver along with the index number, the sender and the receiver's names. The receiver then generates a message that consists of a new random number, the index number, and the sender and receiver's name and encrypts it with the key it shares with the server. The receiver then sends the message to the server with the sender's message. Next, the server generates a random session key then creates two messages that it sends along with the index to the receiver. One is the sender's random number and the session key, encrypted in the key it shares with the sender, and the other is the receiver's random number and the session key

encrypted in the key it shares with the sender. Finally, the receiver sends the sender the message encrypted with the sender's key and the index number, and the sender decrypts the message to recover his/her key and random number and makes sure that they have not changed in the protocol. This protocol makes the two communicating parties convinced of each other's identity.

5. *Kerberos:* In this protocol, it is assumed that everyone's clocks are synchronized with the trusted server's clock. Both the sender and the receiver share keys with the trusted server. When the sender wants to generate a session key to talk to the receiver he/she sends a message to the trusted server with his/her identity and the receivers identity. The server then generates a message with a timestamp, a lifetime, and a random session's key along with the sender's identity. The server encrypts all of this using the key it shares with the receiver, then it takes the timestamp, the lifetime, the session key and the receiver's identity and encrypts them using the key it shares with the sender. The sender then generates a message with his/her identity and the timestamp and encrypts it before sending it to the receiver who in turn creates a message consisting of the timestamp plus one, and encrypts it using the key and sends it to the sender.

6. *Neuman-Stubblebine:* In practice, clocks can be unsynchronized, and thus there is a possible attack against most of the protocols that rely on synchronized clocks. This attack is called **suppress-replay**. In this attack, if the sender's clock is ahead of the receiver's clock, the attacker can take the sender's message and replay it later when the timestamp becomes current at the receiver's site. The Neuman-Stubblebine protocol tries to solve the problem of the suppress-replay. In this protocol, the sender concatenates his/her name and a random number and sends it to the receiver who concatenates the sender's name, and random number and a timestamp and encrypts all with the key he/she shares with the trusted server. The receiver then sends this message to the trusted server along with his/her own name and a new random number. The trusted server then generates a random session key and creates two messages. The first one consists of the receiver's name, the sender's

random number, a random session key and the timestamp all encrypted with the key shared with the sender. The second message consists of the sender's name, the session key, the timestamp encrypted with the key shared with the receiver. Both two messages are sent to the sender in addition to the receiver's random number. Finally, the sender decrypts the message, extracts the key and confirms that the random variable did not change. Then the sender sends the receiver two messages, the first one is the one it received from the server encrypted with the receiver's key, and the second one is the receiver's random variable encrypted with the sender's key. The receiver then decrypts the message, extracts the key and confirms that the new random number and his own random number have the same value.

Hence, as we can see, in this protocol, synchronized clocks are not required because the timestamp is only relative to the receiver's clock and the receiver only checks the timestamp he/she generated.

There are many other protocols that come under Authentication and key exchange, but the above ones are the major ones.

D. ***Multiple-Key Public-Key Cryptography***

Normally, public-key cryptography uses two keys, one of them is public and the other is private. The message encrypted with one key can be decrypted with the other. With the multiple-key concept however, up to n keys can be used. For instance, with three keys, the sender can encrypt a message with the key K_A so that 3 receivers can decrypt the message with another two keys K_B and K_C.

E. ***Secret Splitting***

This protocol involves dividing a message into pieces. Each piece alone is meaningless, but when all the pieces are put together they make the message. As an example, lets' split a message between two people. The trusted server generates a random-bit string that has the same length as the original message. The server then XORs the message with the string to generate a string S and then it gives the random-bit string to one party, and the generated string S to the other party. To reconstruct the message, the two parties XOR their pieces together. In this technique, each piece of the message is worthless, which guarantees extreme security. The problem with this method that if any piece is lost and the trusted server did not interfere, the whole message will be lost.

F. ***Secret Sharing***

In this protocol, the trusted server can divide the message among n parties such that an agreed number of them can put their pieces (or shadows) together to reconstruct the message. For instance, if the message was divided among 4 parties such that any three of them can put their pieces together and reconstruct the message, if one of the parties is absent, the other three can reconstruct the message, but if more than one party is absent, the message can not be reconstructed. This protocol is secure but there are cases where it can be cheated. For instance, if a party intentionally entered a wrong shadow during the process of reassembling the message, the message will never be reassembled and no body will discover who is responsible for the problem.

4.2.2.2 Password Mechanisms and Their Security

There are several existing mechanisms for password authentication, we are presenting here two of these mechanisms **[Hal99]**.

A. ***Password transmission***

Transmitting a password in the clear from the user to the server is the simplest form of password transmission. For password validation, the server stores a file containing either the plain passwords attached to the user name or an image of the passwords under a one-way function. The problem of this mechanism is that an eavesdropper from the network can easily read the password.

Challenge response: This mechanism is a more secure form of password authentication. The password is never transmitted in the clear, rather, it is used to compute a secret function on a challenge selected by the authentication server with each new authentication instance. Although this method provides freshness for the authentication, it leaves the password open to password-guessing attacks. In such attacks, the attacker has access to a relatively small dictionary containing many common passwords. After recording an authentication session including the challenge and the corresponding response from the user, the attacker tries a set of possible passwords on the challenge to see whether the same response is obtained. If so, the password is found.

B. ***One-time passwords***

In one-time password authentication the user uses a different password every time it tries to authenticate itself. If these one-time passwords are derived from a human password, it will still be vulnerable to password-guessing attacks. Providing the user with a list of one-time passwords written on paper and that should be kept secret, can avoid this problem. But again, this method is not fully secure as it allows for attacks including stolen passwords and man-in-the-middle attacks that will be explained below.

4.2.2.3 Security of Password Authentication

Halevi and Krawczyk were the ones who introduced the notion of security for password authentication **[Hal99]**. First, we will introduce the possible attacks that a password-based protocol needs to guard against as presented by Halevi and Krawczyk and by Neuenhofen and Thompson in their paper **[Neu98]**.

- *Eavesdropping:* In this attack, the attacker listens on the line and tries to learn some useful information from ongoing communication between two or more parties. The parties exchanging the information are usually not aware that they are being eavesdropped on. Eavesdroppers might learn confidential information or pick up credentials for use in later authentication procedures. They might as well be able to steal electronic money.

- *Replay:* The attacker records messages sent in past communications and resend them at a later time.
- *Man-in-the-middle:* The attacker in this attack catches the messages between the parties and replaces them with its own messages. The following scenario explains this attack clearly. A sender Alice wants to send a receiver Bob some sensitive information. She starts by sending Bob her public key. Mallory, a malicious attacker catches the message and substitutes one of his public keys for Alice's. He then passes the message on to Bob who accepts Mallory's key as Alice's key and replies with his own public key. Mallory substitutes the key with another public key of his own. Now, when Alice and Bob exchange messages they encrypt them with Mallory's public keys and this allows the attacker to easily catch all of their messages. For the attacker to be able to perform this attack successfully, it has to have access to the communicators' channels of communication.
- *Password-guessing attacks:* Here the attacker has access to a small dictionary that contains common choices of passwords. The attacker can use the dictionary in an off-line attack in which it records past communication and then goes over the dictionary looking for passwords consistent with recorded communication. Or it can use it in an on-line attack in which the attacker repeatedly picks a password from the dictionary and tries to use it in order to impersonate the user. When the password fails, the attacker removes it from the dictionary and tries again with a different password.
- *Exposure of secrets:* An attacker might get access to sensitive data that should have been kept secret. If this happens, the effect that the compromise of the server or the user of any key or file has on the entire system should be minimized.

Halevi and Krawczyk defined an attacker to be:

> An intruder **I** that is allowed to watch regular runs of the protocol between the user **U** and the server **S**, and can also actively communicate with the user and the server in replay, impersonation, and man-in-the-middle attacks. The intruder **I** can

prompt the parties to initiate new authentication sessions. In each session **I** can see all the messages sent between **U** and **S** and can intercept these messages, change them to any value of its choice, or drop them altogether.

For a protocol to be secure in the presence of such an intruder, it is required that at the end of each session between the user **U** and the server **S**, the server should output the pair (U, sid) and the user should output the pair (S, sid), where sid is a session identifier unique to that session and is the same for both the user and the server. The intruder is said to be breaking the authentication protocol if **S** outputs the same pair (U, sid) twice or if it outputs a pair (U, sid), and user **U** never outputs a matching pair (S, sid) **[Hal99]**.

4.2.2.4 Importance of Key Length

Actually, the length of a key contributes heavily in the security of a symmetric cryptosystem. The longer the key length, the larger the number of the attempts that are needed to successfully discover the correct key. For example, if the key is 8-bits length, then 2^8 possible attempts are needed to discover the correct key among the 256 possible keys, and there is a 50% chance of finding the key after half of the attempts. But if the key length is 56 bits long then it will take 2^{56} attempts to discover the key that requires a huge amount of time and processing capabilities.

4.2.2.5 An Analysis of Brute-Force Attack

Brute force attack is a known-plaintext attack where the cryptanalyst has access to both the cyphertext and plaintext of messages. The number of keys to be tested and the speed of each test are the two parameters that determine the speed of a brute-force attack. This type of attacks is best suited for parallel processors where each processor tests a subset of the keys, and they only report success or failure. The problem of attacking messages of long keys is irritating attackers all over the world. This is because it can take years and years to be able to figure out the correct key of an algorithm. Some attackers tried hardware attacks, software attacks and even viruses. For example, in order to get millions of computers to work on a brute-force attack, there will be no possible existing laboratory that can support this attack. S. R. White **[Whi90]** suggests that the attacker can create a non-malicious virus that neither reformat the hard disk nor delete the files on the computer. The virus only purpose is to work on the brute-force cryptanalysis problem whenever the receiving computer is idle. If a machine was able to find the correct key, the virus then can reproduce another virus that deletes the original virus and return back with the needed information, or, the virus could launch a warning message to the computer user saying:

There is a serious bug in this computer.
Please call 0101234567 and read the following 64-bit number to the operator:

xxxx xxxx xxxx xxxx

There is a $100 reward for the first person to report the bug

Fig. 4.2 A Sample Virus Warning Message

Actually, this is a very smart way of getting back the key without harming the users' computers.

Generally speaking, a longer key provides more security, but it is worth remembering that it costs more in terms of computation time as the keys get longer. So, we want a key long enough to provide the needed security and short enough to be computationally usable and cost effective. But until now, there is no single answer to the question of how long should a key be because this depends on the situation. It also depends on how much the data to be discovered is worth, and how long does it need to be secure.

4.3 Key Management

As was mentioned before, the security of an algorithm lies in the key. If the key generation algorithm was weak, then the whole system will be weak, and the key generation algorithm itself can be easily attacked.

4.3.1 Choosing keys

Poor key choice significantly affects the security of the algorithm. And many people do choose obvious keys, which makes the job of a smart brute-force attack easier. It only has to try the obvious keys first and then try all possible keys in numerical order. This is called a dictionary attack because the attacker uses a dictionary of common keys. The most important thing is to use a good random-key generator and to use the appropriate encryption algorithm and key management procedures.

There are many guidelines to choose a key. The most important thing is that the key should not be easy to discover. Using long keys may be a solution but it is not usually the best. Sometimes it is not easy to remember very long keys for a long time. Therefore, ideally, a key should be something easy to remember but difficult to guess. As an example, we can use word pairs separated by punctuation character or use strings of letters that are an acronym of a longer sentence. Another suggested solution is to use a technique called *pass phrases.* In this technique, an entire phrase is used instead of a word and is then converted into a key using a one-way hash function to transform a text string to a pseudo-random bit string. These techniques help in obscuring the key, which provides security, but obscure keys do not substitute for true randomness that makes keys difficult to remember.

The entropy of a cryptosystem (the amount of real information in the plaintext) is a measure of the size of the keyspace, K. The greater the entropy, the harder it is to break the cryptosystem, because it will have a greater key size. This is a directly proportional relationship.

4.3.2 Transferring keys

If the two communicating parties need to use the same key, there should be a secure means of transferring the key from the generator party to the other party. An appropriate method is to encrypt the key that is to be sent using Key-Encryption keys that are created to encrypt other keys for distribution. Another solution for the problem of transferring keys is to split the key into several different parts and send each of those parts over a different channel.

4.3.3 Verifying keys

When the receiver receives a key, he/she should verify that it truly comes from the sender not someone else pretending to be the sender. This is simply achieved on a trust basis. That is, if the sender uses a key-encryption key to send the key, the receiver has to trust this mechanism. If the sender uses a digital signature protocol to sign the key, the receiver has to trust the public key database when verifying the signature. This trust-based mechanism relies on the fact that it is practically very difficult for an attacker to successfully discover the key without very powerful resources that the attacker has to have full access over. Another method that would be helpful beside the trust-based system is voice recognition. The receiver can verify the sender's authentication using the phone. If the key was a public key then it can be easily recited in public because with public keys, the encryption key is totally different from the decryption key and so reciting the encryption key in public will be of no harm at all. Sometimes it is not as important to verify who is the owner of the public key as verifying that it is the same person that the key belonged last year.

Unfortunately, sometimes errors do occur during key transmission, and so the receiver will not accept the key. This problem can be overcome by providing error correction and detection bits with the key itself. For instance, there is a method that involves encrypting a constant value with the key and sending the first 2 to 4 bytes of the cyphertext with the key, and doing the same thing at the receiving end. If the encrypted constants match, then this proves that the key was transmitted without errors.

4.3.4 Updating keys

Sometimes, changing keys daily will be required in an application. And thus, it will be painful to distribute a new key everyday. Generating a new key from the older one, or key updating is an easier solution for this problem. This method involves using a one-way function that the sender and receiver operate on. If the two parties share the same key, using the one-way function, they will get the same result, and then can take the bits they need from the results and use them to create the new key. The newly generated key will have the same level of security as the old one.

4.3.5 Backing Up Keys

It is very important to backup keys to avoid the risk of forgetting or loosing the source of the key for any reason. A simple method to backup keys is Key escrow. This method involves writing the keys on paper and then locking them in a secret place or encrypting them all with a master key. But again, the security officer who keeps the master key should be trustworthy not to mess up with all the keys he/she keeps. That is why, another solution is proposed. This solution is the secret-sharing protocol. The key owner divides the key into pieces and sends each piece to a different security officer. To regenerate the key, someone has to collect all the pieces together. This method protects the owner against any one malicious person.

4.3.6 Key Lifetime & Destruction

Every encryption key expires automatically after a specific time period. This adds to the security of a key, because the longer the key is used, the greater the chance that it will be discovered, and the larger the temptation for any person to attack it because that person will have enough time to do so. Also, if the same key is used for multiple purposes, its loss will have greater damaging effects.

The length of the key is determined by the type of application it is used in. For instance, systems on dedicated communications channels should have relatively short lifetimes depending on the value and amount of data encrypted during a given period. On the other hand, key-encryption keys do not have to be frequently changed because they are occasionally used for key exchange therefore they can have longer lifetime. Same for encryption keys that are used to encrypt data files for storage, they do not need to be changed often.

After being used, old keys are replaced and then destroyed. They have to be destroyed because an attacker can use them to read old messages encrypted with those keys.

4.4 Public-key Cryptography vs. Symmetric Cryptography

Public-key cryptography and symmetric cryptography solve different sorts of problems. For instance, Symmetric cryptography is best for encrypting data because it is faster and is not threatened by cyphertext attacks. On the other hand, public-key cryptography can do things that symmetric cryptography cannot. For instance, it is best for key management.

4.5 Where Can Encryption Take Place?

Generally speaking, encryption can take place either at the lowest layers where everything going through a particular data link is encrypted, and is called link-by-link encryption, or at the highest layers where the data are encrypted selectively and stay encrypted until they are decrypted by the intended final recipient, and is called end-to-end encryption.

Link-by-link encryption occurs at the physical layer. It is easy to connect hardware devices at this point of the OSI. These devices encrypt all the data passing through them, and any nodes between the sender and the receiver, whether switching or storing nodes need to decrypt the data stream before processing it. The cryptanalyst cannot get any information about the structure of the information itself, like who is talking to whom, the length and time of the messages, etc…. As for the keys, only the two endpoints of the line need a common key and they can change their key independently from the rest of the network **[Sch96]**. The problem with encryption at this layer is that each physical link in the network needs to be encrypted otherwise the security of the network will be threatened. Also, security between the nodes is not guaranteed since the data processed at each node is unencrypted.

End-to-End encryption involves placing equipment between the network layer and the transport layer. The encryption device should encrypt only the transport data units which are recombined with the routing information and sent to be transmitted by the lower layers. This method allows the data to remain encrypted until it reaches its final destination, which solves the problem of the link-by-link approach. But on the other hand, the primary problem here is that the routing information for the data is not encrypted.

Combining the two above methods together is the best way of securing a network, although it might be much more expensive. We encrypt each physical link as well as applying end-to-end encryption which reduces the threat of unencrypted data at the different network nodes **[Sch96]**.

4.5.1 Hardware Vs. Software Encryption

Until a near time, encryption was performed using hardware devices. Those devices are encryption/decryption boxes that are plugged into a communication line and encrypt all data going across the line. Hardware encryption is much faster and cheaper since it does not involve all the complicated operations that are built in encryption algorithms. Also, hardware encryption is more efficient in terms of processor usage. Tying up a whole processor to execute an encryption algorithm reduces efficiency, whereas moving the entire encryption process to an external chip speeds up the whole system and increases efficiency.

Most important is the security issue. Hardware encryption is much more secure since the devices can be encapsulated, thus preventing any possible attacks or modifications of the encryption algorithm. On the other hand, software encryption algorithms are more flexible and portable. They can be easily and cheaply copied and installed on any machine. But the problem of software encryption is that it is slower and more expensive in terms of processor usage. The most important thing is to ensure that the key management scheme is secure enough **[Sch96].**

4.6 Criteria for a Standard Cryptographic Algorithm

For a cryptographic algorithm to be defined as standard, it has to satisfy several criteria including:

The algorithm must provide a high security level.

The algorithm must be completely specified and easy to understand.

The security of the algorithm must reside in the key.

The security should not depend on the secrecy of the algorithm.

The algorithm must be available to all users.

The algorithm must be adaptable for use in diverse applications.

The algorithm must be economically implementable in electronic devices.

The algorithm must be efficient to use.

The algorithm must be able to be validated.

The algorithm must be exportable.

4.7 The Data Encryption Standard (DES)

The data encryption standard (DES) is a worldwide standard for 20 years. It was approved to be a private sector standard by the American National Standards Institute (ANSI). DES is a block cipher, and it encrypts data in 64-bit blocks. It is a symmetric algorithm that uses the same algorithm and key for both encryption and decryption. DES uses a key of length 56 bits that can be changed at any time. All the security relies within the key. The DES was proven to be secure after debates on the possibility of its weaknesses in areas like key lengths and internal design.

The DES algorithm relies on an idea called confusion and diffusion. These two basic encryption techniques help in obscuring the redundancies that might exist in a plaintext message. Confusion aims to obscure the relationship between the plaintext and the cyphertext. This makes the attempts to study the cyphertext looking for redundancies and patterns difficult. Diffusion spreads out the redundancy of the plaintext over the cyphertext. A cryptanalyst searching for these redundancies will face difficulties in finding them. The fundamental building block of DES is a combination of the confusion/Diffusion techniques. It applies them on the plaintext 16 times (called rounds), based on the key **[Sch96]**.

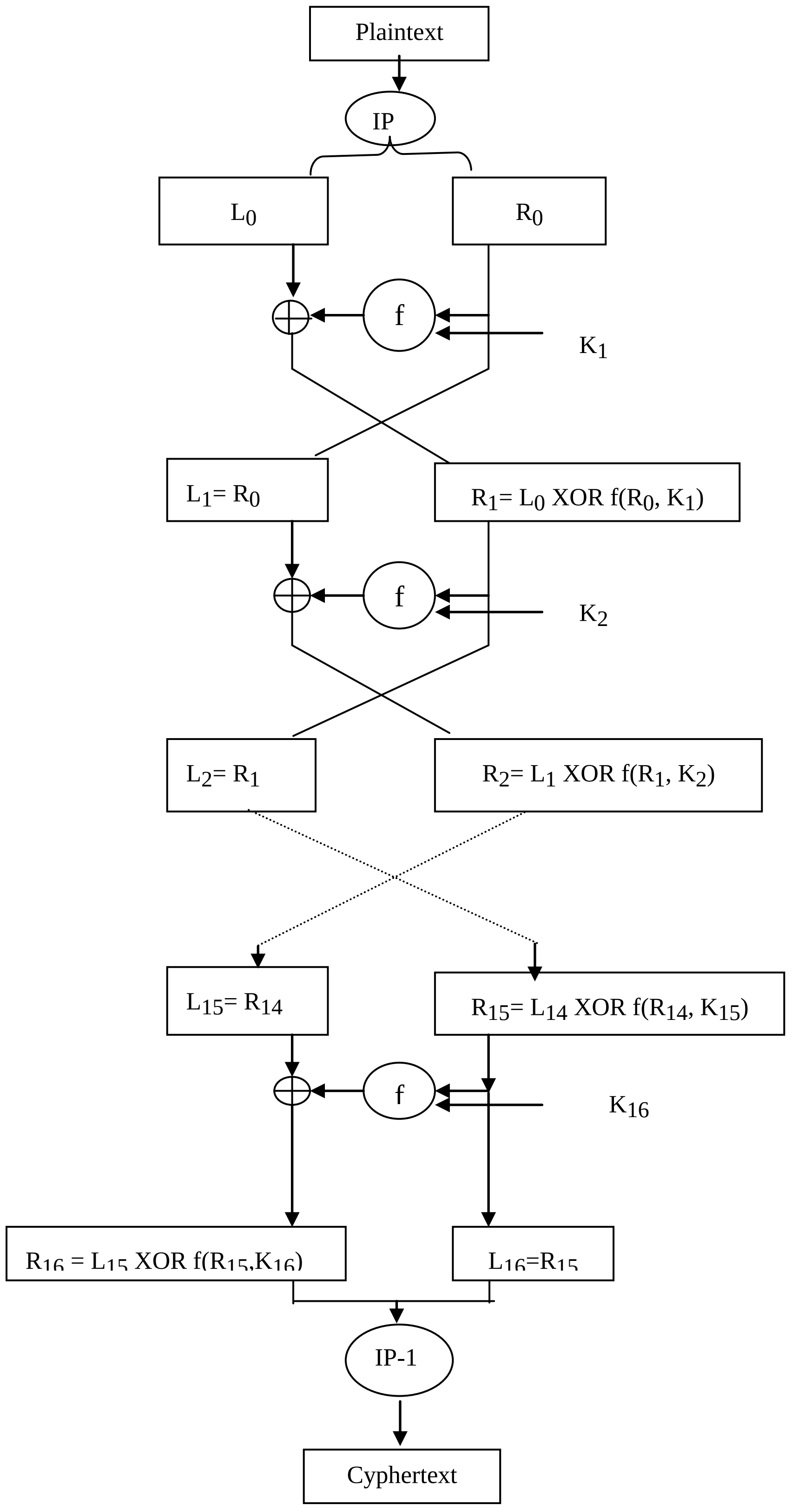
The DES algorithm operates on a 64-bit block of plaintext. A shown in figure 4.3, the block is broken into 2-32-bit blocks after performing an initial permutation (IP). These two blocks are the left half and the right half in the figure denoted by L0-L15 and R0-R15 consecutively. Then, 16 rounds of operations are performed upon these 2 blocks. In these rounds, data are combined with the key. After the 16 rounds finish, the 2 halves are recombined again and a final inverse permutation is performed to finish the algorithm.

In each of the 16 rounds, the key bits are shifted and then 48 bits are selected from the 56-bits key. Next, the right half is expanded to 48 bits from 32 using a dedicated expansion permutation, and is XORed, as shown in the figure, with the other 48 bits of a shifted and permuted key. Ki is the 48-bit key for round i. This whole process forms a function f whose output is XORed with the output of the left half. The result of these operations become repeatedly the new right half, and the old right half becomes the new left half for 16 times, making up the 16 rounds of the DES algorithm.

If Bi is the result of the ith iteration, Li and Ri are the left and right halves of Bi, Ki is the 48-bit key for round I, and f is the function that does all the substituting, and permuting and XORing with the key, then a round looks like this:

$Li=R_{i-1}$

$Ri = L_{i-1}$ XOR $f(R_{i-1}, Ki)$



Plaintext
IP
L_0
R_0
f
K_1
L_1= R_0
R_1= L_0 XOR f(R_0, K_1)
f
K_2
L_2= R_1
R_2= L_1 XOR f(R_1, K_2)
L_{15}= R_{14}
R_{15}= L_{14} XOR f(R_{14}, K_{15})
f
K_{16}
R_{16} = L_{15} XOR f(R_{15},K_{16})
L_{16}=R_{15}
IP-1
Cyphertext

Fig. 4.3 The DES Algorithm Structure

4.8 Existing Block Ciphers

4.8.1 LUCIFER

It is a block cipher with building blocks similar to DES and has 16 rounds as well. But unlike DES, in Lucifer, there is no swapping between rounds, and no block halves are used. Some people feel that Lucifer is more secure than DES because of the longer key length and lack of published results, which is not always the case.

4.8.2 MADRYGA

Madryga is a block algorithm that has no irritating permutations and all its operations work on bytes. The design objectives of W. E. Madryga, the algorithm designer, included:

1. "The plaintext cannot be derived from the cyphertext without using the key", which assures the algorithm's security.
2. "The number of operations required to determine the key from a sample of plaintext and cyphertext should be statistically equal to the product of the operations in an encryption times the number of possible keys."
3. "Knowledge of the algorithm should not defeat the strength of the cipher", that is all the security should rest in the key.
4. Redundant bit groups in the plaintext should be totally obscured in the cyphertext.
5. The length of the cyphertext should be the same as the plaintext.
6. There should be no weak keys.
7. The length of the key and the text should be adjustable to meet varying security requirements.

Madryga solved the security problem that arose from the fixed-length key in DES. A variable-length key can surely silence those who thought that 56 bits was a short key length. But Madryga cannot be considered terribly secure. The algorithm consists only of linear operations that are slightly modified depending on the data, and it is considered weaker than DES in terms of its inner structure.

4.8.3 NewDES

This algorithm was designed as a possible replacement of DES. It operates on 64-bit blocks of plaintext, but it has a 120-bit key. It has no initial or final permutations like the old DES. All operations are performed on entire bytes. It was proven that NewDES is weaker than DES because it requires less work for a brute-force attack to figure out the key than in DES.

4.8.4 FEAL

FEAL was based on the idea of creating a DES-like algorithm with a stronger round function. It uses a 64-bit block and a 64-bit key. It was first initiated with 4 rounds, but then, it was easily attacked. Its designers then introduced FEAL-8 with 8 rounds, but again it was proven to be weak as it was attacked by a chosen-plaintext attack. This forced its designers to introduce FEAL-N with a variable number of rounds greater than 8. But this did not prevent attackers from creating Brute-force attacks to attack FEAL-N.

4.8.5 REDOC

REDOC II is another block algorithm designed with a 20-byte key and an 80-bit block. It performs all of its manipulations on bytes. It has 10 rounds. REDOC II is proven to be very secure against brute force attacks.

REDOC III is a modified version of REDOC II. It operates on an 80-bit block and a variable key-length. The algorithm is easy and fast. But REDOC III is not secure, it can be subject to differential cryptanalysis, and is easier to be attacked by a brute force attack.

4.8.6 LOKI

This algorithm uses a 64-bit block and a 64-bit key. It was proven that with less than 14 rounds, LOKI can be attacked by differential cryptanalysis. That is why, its designers created LOKI91 with slight differences in the internal design. LOKI91 has 16 rounds. LOKI91 is secure against differential cryptanalysis. But it can be attacked by a chosen brute force attack.

4.8.7 KHUFU AND KHAFRE

These two algorithms were proposed in 1990 by Ralph Merkle **[Mer91]**. Their basic design principle included:

1. Increasing the key size more than the DES's 56-bit key size, specially that the cost of increasing the key size is negligible.
2. Reducing the number of permutations in DES because they are not suitable to be implemented in software, as they hinder the algorithm speed.
3. Discarding the initial and final permutations in DES.

3.8.7.1 Khufu

Khufu is a 64-bit block cipher. It calls for a total key size of 512 bits. The number of rounds for the algorithm is left open but it is recommended to use more than 16 rounds, and a number that is a multiple of 8. Khufu was proven to be resistant to differential cryptanalysis. It was proven to be impressively secure especially with a 512-bit key or more.

3.8.7.2 Khafre

It is similar in design to Khufu but it was designed for applications without pre-computation time. The designer stated that key sizes of 64 – 128 bits would be used for Khafre and that more rounds of encryption would be required for Khafre than for Khufu. Since each round of Khafre is more complex than for Khufu, Khafre is considered slower, but it will encrypt smaller amounts of data more quickly since it does not require any pre-computations.

4.8.8 IDEA

It is claimed to be the best and most secure block algorithm currently available to the public. IDEA is a block cipher that operates on 64-bit plaintext blocks, with a 128 bits key which is twice as long as the DES key. The same algorithm is used for both encryption and decryption. IDEA uses both confusion and diffusion. It is based on the idea of "mixing operations from different algebraic groups". These algebraic groups include XOR, Addition modulo 2^{16}, multiplication modulo $2^{16} + 1$. All of these operations operate on 16-bit sub-blocks and the algorithm is more suitable to be used on 16-bit processors. Current software implementations of IDEA are about two times faster than DES. Since the key length is 128 bits, it would take an efficient brute-force attack 2^{128} encryptions to recover the key which is a terribly long time. It was also proven that IDEA is immune against differential cryptanalysis after only 4 of its 8 rounds. Until now, there is no public attack to IDEA that was proven to be successful.

4.8.9 MMB

This algorithm is based on the same theory as IDEA with a 128-bit key and a 128-bit block size. It operates on 32-bit sub-blocks of text and 32-bit sub-blocks of key that makes it more suitable for implementation on modern 32-bit processors. This algorithm was designed not to have any weak keys like IDEA has. But on the other hand, it was not designed to face linear cryptanalysis.

4.9 Summary

Chapter four presented an overview of the encryption and decryption issues. It presented the six basic cryptographic protocols: Key exchange, authentication, authentication and key exchange, multiple-key public-key cryptography, secret splitting, and secret sharing. In addition to that, the key management issue was discussed in this chapter in terms of how to choose keys, how to transfer, verify, update and back up the keys, and the consequences of different key lengths.

The table below summarizes the different encryption algorithms mentioned in this chapter. Of course the field of cryptography is a fast growing field and new algorithms are frequently introduces, that is why attempts to prove the weaknesses of existing algorithms normally occur.

Algorithm	Characteristics
DES	• Relies on the idea of confusion/diffusion which help in obscuring the redundancies that might exist in a plaintext message. • Operates on a 64-bit block of plaintext. • 16 rounds of operations operate on these two blocks. • In these rounds data are combined with the key. • Swapping between rounds occur. • These two halves are recombined again and a final permutation is performed on them. • Has a 56-bit key length.
LUCIFER	• Has building blocks similar to DES • Has 16 rounds • No swapping between rounds. • No block halves are used • Longer key length than DES. • Has a 128-bit key length.
MADRYGA	• Does not have permutations • Plaintext cannot be derived from cyphertext without using the key. • Has a variable key length • The length of the cyphertext should be the same as the plaintext. • Algorithm consists only of linear operations which are slightly modified depending on the data.
NEWDES	• A possible replacement of DES. • Operates on a 64-bit blocks of plaintext. • Has 12-bit key length. • All operations are performed on entire bytes. • Weaker than DES.
FEAL	• Uses a 64-bit block • Uses a 64-bit key Started with 4 rounds then was improved until it reached a variable number of rounds greater than 8.
REDOC	• Has a 20-byte key • Performs all of its manipulations on bytes. • Has 10 rounds • REDOC III is a modified version of REDOC • REDOC III operates on an 80-bit block and a variable key length.
LOKI	• Uses a 64-bit block • Uses a 64-bit key • Has 16 rounds

KHUFU	• Uses a 64-bit block. • Has a 512 bits key size. • Number of rounds is left open bit is recommended to be more than 16 and a multiple of 8.
KHAFRE	• Uses key sizes of 64-128 • Uses more rounds of encryption than KHUFU • More complex rounds • Slower than KHUFU • Encrypts small amounts of data faster since it does not require any Precomputations
IDEA	• Operates on 64-bit plaintext blocks. • Has a 128 bits key. • Uses confusion and diffusion. • Based on the idea of mixing operations from different algebraic groups including XOR, addition modulo 2^{16}, and multiplication modulo 2^{16+1}. • More suitable to be used on 16-bit processors.
MMB	• Based on the same theory as IDEA • Has 128-bit key • Has 128-bit block size • Operates on 32-bit sub-blocks of text and 32-bit sub-blocks of key. • More suitable for implementation on modern 23-bit processors.

Table 4.1 – Comparing Encryption Algorithms

A will be seen later in the implementation chapter, the algorithm used in the implementation of this thesis is the DES algorithm. This algorithm was chosen because the JAVA language used in the implementation supports security classes that use DES for encryption.

Chapter 5 – Protecting Agents from Malicious Host Attacks

5.1 - Introduction

Since, the area of mobile agents is relatively new, not much work has been done in the field of their security. There is no complete solution for this problem but there are some partial solutions. Existing mobile agent systems are still investigating efficient methods to protect their agents from malicious host attacks. For instance, Agent TCL make their agents anonymous as soon as they migrate through an untrusted machine, other systems use hardware solutions, others deal only with trusted hosts. The following are some solutions that were proposed in an attempt to overcome the threat of malicious hosts.

5.2 Solutions

5.2.1 Legal protection

An approach to the agent security problem is via legal/contractual means. That is, operators of the servers on which the agents run, promise via contractual agreements and guarantees that they will keep their servers secure from any malicious external attack, and that they will not violate the privacy or attack the code or data of any agents running on them **[Yee97]**. This method requires no complex cryptographic protocols, nor any run-time overhead. But in the mean time, this approach is not satisfactory because server operators cannot always be trusted. Also, for that detection to be meaningful, tamperproof logs should be available to serve as non-repudiable evidence of the breach of contract.

5.2.2 Hardware Solutions

These involve binding the execution of a program to the existence of a special medium or piece of hardware (dongle). This will allow the software to try to find periodically whether it can verify the existence of the dongle. Normally each user is provided with one dongle per program and the software does not start without it, and this protects the program, although it might allow copying the program, it does not allow the program to be used without the dongle. This approach does not provide full protection. Although it prevents the unauthorized execution of the program, it does not protect the program from manipulation by the user, and thus the part of the code that verifies the existence of the dongle can be simply removed by a hacker and thus the program is cracked.

5.2.3 Trusted machines and non-critical agents

The problem of malicious hosts can be reduced to an "inner-institutional problem" by letting trustworthy parties maintain the hosts. And hence, some agents do not need any protection at all, either because they are performing some non-critical task, or because they operate entirely on trusted machines. But the problem of this solution is that the mobile agent will not be considered open any more. It will be confined to those trusted machines and any foreign service provider will find it difficult to connect a new host as part of the platform to the mobile agent system. This is the approach General Magic used for its agent system application e.g. PersonaLink, that was operated by AT&T **[Gra98]**.

5.2.4 Partitioning

An agent can migrate through trusted machines only under the control of a trusted Internet Service Provider **[Gra98]**. Then it either interacts with untrusted resources from across the network using standard RPC, or sends out child agents that contain no sensitive data and will not migrate again, and that will only return their results since they contain no sensitive data. Thus the sensitive portion of the agent can always be left on the home machine that increases protection.

5.2.5 Replication and voting

With replication, if a task requires a single agent to visit n services in sequence, the application sends out several agents instead of one. Each of this agents visits distinct but equivalent copies of the n services. The agents exchange results after each stage, and each agent keeps the majority result **[Gra98]**. But the drawback of this system is that multiple copies of each service must exist which imposes a great overhead. Also, if the user spends money to access a service, it will have to spend more money than it would have spent if a single agent migrates through a single copy of the service. In addition to that there will be an increased cryptographic overhead to protect all those sent agents. This method is considered the only method that handles services providing incorrect information, that is why it is currently used in Tacoma.

5.2.6 Components

This method involves dividing each agent into components which can be added to the agent as it migrates and each component can be encrypted and signed with different keys **[Gra98]**. For example, the static code of the agent and its fixed variables can form a component signed with the owner's key before it leaves its home host. If a malicious machine modifies the code or variables, the digital signature becomes invalid and the next machine in the migration sequence will immediately detect the modification. If the agent carries sensitive information, it can place it in its own component which is then signed with the machine's key and even be encrypted with the trusted machine's key to prevent tampering and examination by other machines. For example, an agent can encrypt its electronic cash with a proxy site's key, so that it migrates through untrusted machines without worrying about any possible thefts, and the agent can return to the proxy site at any time it needs to spend the cash. Components make it easier for the agent to use the partitioning approach above by leaving particular component behind on a trusted machine, or creating and sending out a child agent that carries only certain components.

5.2.7 Self-Authentication

Almost every agent has certain parts that change as the agent migrates from host to host, like the variable values and control information. To detect any malicious attempts to change this state information, an authentication routine is constructed to examine this information and check for any inconsistencies **[Gra98]**. This routine can also examine the current set of components. It can be placed in its own component and then is digitally signed with the owner's key. If any inconsistencies are found while executing this routine, the agent server terminates the agent and notifies the home machine. Such an authentication routine allows a machine to trust an agent that has migrated through untrusted machines.

5.2.8 Migration History

Allows the detection of some rerouting attacks. It involves embedding a migration history inside the moving agent **[Gra98]**. This allows the "detection of some re-routing attacks" especially if the agent usually follows a repeated path. This movement history can be examined inside the authentication routine above.

5.2.9 Audit logs

In this method, machines keep logs of important agent events so that "an aggrieved agent or owner can request an audit from an authorized third-party" **[Gra98]**. The auditor normally seeks to identify the machine responsible for a theft or modification and assigns a punishment to that machine. But this method does not provide a forward protection. It can provide protection only after an attack occurs and a host is known to be malicious**.**

5.2.10 Read-Only State

The Ajanta mobile agent system employed this mechanism to prevent tampering of agent's critical information like agent's credentials or data that are carried by agents and that should not be tampered with like emails **[Kar99]**. This mechanism involves declaring parts of the agent as read-only. i.e. constant during the agent's travels. And any tampering with the read-only objects can be detected. But again, this method does not prevent the attackers from reading sensitive information from the agent, and it is only suitable for the parts in the agent that remain constant throughout the agent's travel.

5.2.11 Append Only Logs

Again, the Ajanta mobile agent system employed this mechanism to prevent any modifications of the data that an agent collects while it travels. It can also be called write-once data **[Kar99]**. The data can be modified any number of times by the agent until it decides that it should not be modifiable any more. And so, no party can modify the data carried by the agent until it delivers it back safely to its host. Agents may have the append only logs as part of their state. Entries in the log cannot be deleted or modified. If secrecy is also needed so that no body can read the data, the data in the append only log can be encrypted with the agent's public key before it is stored in the log.

5.2.12 Encrypted Algorithms

Encrypting the agent's code is a proposed solution for the problem of malicious hosts **[Gra98]**. The encryption algorithm should make sure that the program encrypted is directly executable, performs the same task as the original program, and can be decrypted using an existing decryption algorithm created by the program encryptor.

5.2.13 Digital Signatures

Digital signatures are a major cryptographic technique that can be used to protect agents and their codes. The application of digital signatures to mobile agents involves using the signatures to preserve partial results from computation done while executing in a server **[Yee97]**. For example, a server can sign a message using the server's key; this prevents any malicious agent from completely brainwashing the agent carrying the message. The digital signature approach is not fully general; it only applies to certain classes of functions where there are intermediate results that can be compressed.

5.2.14 Partial Result Authentication Codes(PRAC)

The idea here is based on authenticating an intermediate agent state or partial result that resulted from running on a server, not only authenticating the origin of the agent. PRACs are cheaper than digital signatures to compute **[Yee97]**. They ensure perfect forward integrity. That is, if a mobile agent visits a sequence of servers S = s1, s2, …, sn, and the first malicious server is Sc, then none of the partial results generated at servers Si, where i < c can be falsified. This is unlike digital signatures, where if an attacker attacks the generating host where the signature key is stored, the authenticity of all the messages signed with that particular key will be subject to suspicion.

5.2.15 Code Mess Up

An attacker needs a certain amount of time to read the data, understand the code and then manipulate both of them in a meaningful way. The idea here is not to give the attacker enough time to do this. This can be achieved by code mess up and limited lifetime of code and data. Code mess up involves creating a different code that does the same task but with a different inner structure. Also, the data elements are mixed up and are distributed to a list of new data elements. There are several possible mechanisms to implement code mess up. For instance**,** Fritz Hohl **[Hoh97]** proposed a mechanism for code mess up that is based on the idea of violating the well-known software engineering directives of writing code**.** As an example, he suggested using “variable recomposition” in which the program variables are mixed up and new variables are created that contain some bits of data from some of the original variables and adapts the corresponding variable accessing in the program code. Another approach for code mess up was proposed by Takanori et al [**Tak97]**. They suggested the insertion of dummy code in the original agent’s code. This dummy code has to be nullified e.g. the use of a pair of add and sub. The code mess-up approach trades both execution time and communication bandwidth Vs. the security it provides to the agent while carrying security sensitive work without the danger of being exploited by a malicious host.

Hohl presented some agent mess-up algorithms in **[Hoh97].** Generally, the designer of a mess-up algorithm should put into consideration two key issues: the modifiable attributes of an agent and the abilities and characteristics of the attacker. *Modifiable agent attributes* include statements and data. A statement has a *type* and a *location* in a program. We can hide the type of a statement until it is executed by creating it dynamically at runtime. This is possible by using self-modifiable code for example. Not only the type of a statement can be hidden, its location can be hidden as well, either implicitly by using dynamic code creation or explicitly by hiding a certain statement into other statements. Data include variables and constants. A piece of data consist of a type, a value and a location. The type of a date element can be hidden until the data is needed. The value can be hidden as well by replacing element accesses by accesses on subelements and to translate operations on the data elements by operations on the subelements. This will result in an execution where the value of a data element never occurs as a whole. Also, the location of a data element can be hidden statically by splitting up the element and distributing the parts, or dynamically by allowing the element to move around in the data area.

The abilities of the attacker have to be taken into consideration as well. There are two cases that have to be considered here. The first case is when the attacker does not know the original version of the agent in advance. Therefore the attacker has to analyze the agent in order to build a mental model. This is a slow process. In the second case, the attacker does not know the exact specification of the agent in advance. If the type of the agent was identified, then the exact specification of the agent will be easier to identify, and thus the attack can be automated by generating a program that tries to compute only a few attributes of the agent. **[Hoh98]**

Good mess-up algorithms should not allow the complete analysis to be done statically, but also require the agent to be run at least partially. Examples of mess-up algorithms include: variable recomposition, conversion of control flow elements into value-dependent jumps, and deposited keys.

1. *Variable Recomposition:* in this algorithm, each variable in the set of program variables is cut into segments and new variables are created that contain a recomposition of the original segments. For example, the figure below shows a shopping agent example and the agent's original variable access. Figure 5.2 shows the recomposition into two new variables: v23 and v19 from the contents of three of the original variables.

A Shopping Agent example

Entries in the data area

Address **home** = “PDA, sweet PDA”

Money **wallet** = 20 $

float **maximumprice** = 20.00 $

good **flowers** = 10 red roses

Address **shoplist[]** = empty list

int **shoplistindex** = 0

float **bestprice** = 20.00 $

Address **bestshop** = empty

Original variable access

```
5    buy (bestshop, flowers, wallet)

6    go(home)
```

Fig. 5.1 – A shopping agent example

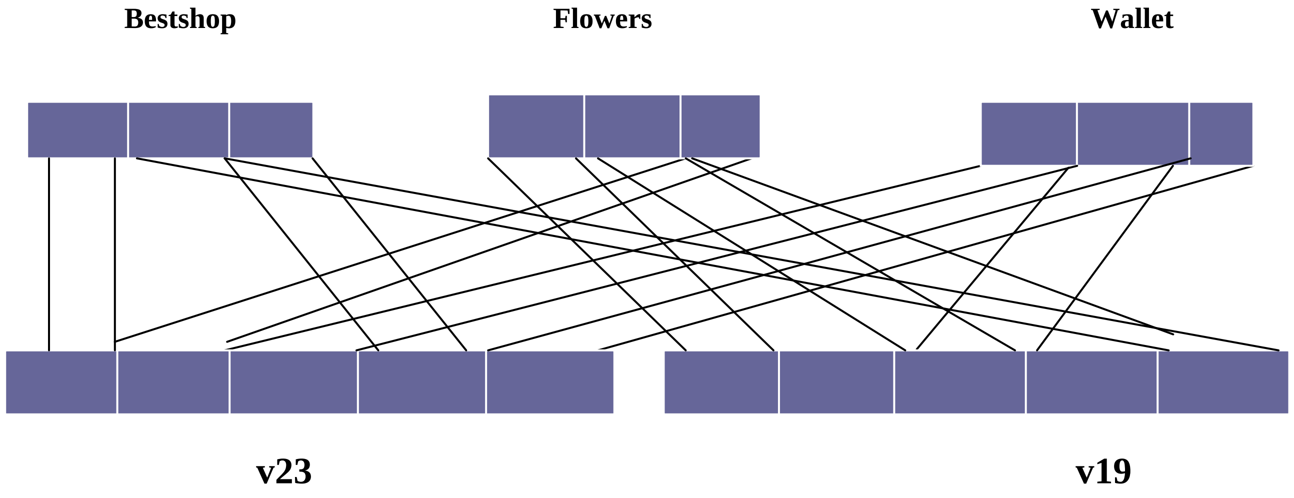


Fig. 5.2 – Variable Recomposition

In our example, the new variable access code for the new variables that is shown in second part of figure 5.3, is automatically created by using the conversion functions that create the original values from the new variables as shown in the upper part of the same figure. Thus, we are removing any direct relationship between the variables and the processing model elements, and the variable names are now meaningless and the data representation is becoming more complicated.

Conversion functions

```
Public Address c7(Bitstring b)
public Good c4(Bitsring b)
public Money c3(Bitstring b)
public Address c34(Bitstring b)
```

New variable access

```
7   buy (c7(v23[0]+v19[4]+v23[3]),
        c4(v19[0]+v19[3]+v23[1]),
        c3(v23[2]+v19[1]+v23[4]))
        go (c34(v21[4]+v19[2]+v21[2]))
```

Fig. 5.3 Conversion functions and New Variable Access

Attacking the above algorithm can be achieved by two ways. The first can involve guessing the variable layout by analyzing the access to a byte statistically over the operations known to access certain original variables. The second way tries to read the original variables by reading the parameters of the calls of the known procedures. For example, as soon as the *buy* procedure is called, the attacker sees the values of *bestshop, flowers, and wallet.*

2. *Conversion of Control Flow Elements into Value-Dependent Jumps:* a second code-mess up mechanism involves converting control flow elements. Control flow elements like if and while statements allow the programmer to imagine the flow of control even at compile time. Therefore, if we convert these elements into a form dependent on the variable contents, the control flow will not be easily determined. This can be achieved by using jumps that are bound to variable contents like switch statements. **[Hoh98]**
3. *Deposited Keys:* The idea here is to encrypt protection information included in the agent and that are subject to be attacked. Then the agent requests the keys of these parts from the trusted server by indicating the state of the agent.

5.2.16 Limited lifetime of code and data

An attacker can crack an agent only if it has enough time. So, we can make some parts of the agent invalid after a specified interval, this can be done using digital signatures **[Hoh97]**. That is, we can attach expiration dates to the data elements that are used for interaction with the outside world. There are four circumstances under which time limitedness affects processing **[Hoh98]**. These four cases are:

1. *When there is no communication with a third party.* In this case, neither the agent nor the host communicates with a third party. Thus, the time limited protection here is of no importance since there will be no threat from a third party.
2. *When there is communication only with trusted servers.* In this case, the agent only communicates with a trusted third party, that is one that never attacks. These two partners can establish a secure communication channel to prevent attacks by the host. Here, the agent time limitedness is important since the communication partner has to know whether it can still trust the agent or not. Normally, attacks can only take place after the protection interval; and so, the trusted server has to know the expiration date associated with the agent before it starts

communication. This can be done using an extended key certificate like the one shown in the figure below.

Version
Serial Number
Certification Algorithm
Certification parameters
Name of communicating agent.
Velidity of certificate
Name of the agent
Expiration Date
Public key of Agent

Signature of communicating agent

Fig. 5.4 - An extended Key Certificate

3. *When there is communication with untrusted servers.* In this case, the agent communicates with either an untrusted third party or with the host. There are two kinds of data that can be communicated. The first type is called **Token Data**. It includes self-contained documents that depend on the identity of the issuer. That is why they often have digital signatures. This type includes electronic money coins, secret keys and capabilities. Tokens have to bear expiration dates to prevent trading them by an attacker. When a party receives a token, it has to check its expiration date putting in mind that the receiving party should have the correct global time, this is because if a party accepts an outdated token, no other party will accept it in return. The problem here is that a token cannot be protected from malicious attacks after it expires. Thus, an agent must not transport tokens that need a larger protection interval.

   The second type of data is called **non-token data.** This type includes simple values that do not need to be protected and values that are security sensitive like the maximum price for example. Again, an agent must not transport non-token data that can be used to attack the owner of the agent and whose protection interval has to be larger than the lifetime of the agent.

4. *When there is agent migration.* In this case, the receiving host has to ensure that the arriving agent is still valid, that is, its expiration date has not passed already.

5.2.17 Time Limited Blackbox Security Approach

The idea behind this approach is to create a blackbox out of an original agent. A blackbox is simply an agent that performs the same work as the original agent, but is of a different structure **[Hohf98]**. This difference allows assuming a certain agent protection time interval, during which it is impossible for an attacker to discover relevant data or to manipulate the execution of the agent. The agent and some associated data get invalid after that time interval and the agent will not be able to migrate or interact anymore. This will prevent the exploitation of attacks after the protection time interval. In other words, what is required is creating an executable agent from a given agent specification. In **[Hohf98],** an agent is defined to be a time limited blackbox if and only if:

- For a certain known time interval, the code and data of the agent specification cannot be read,
- The code and data of the agent specification cannot be modified,
- And attacks after the protection interval are possible but these attacks do not have effects.

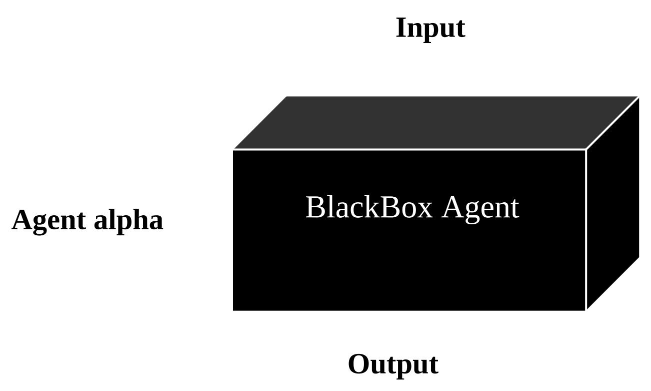


Fig. 5.5 - The Time Limited Blackbox Property

This way, only the input and the output of the blackbox can be observed as shown in figure 5.5. During a certain time interval, the agent will act as a black box whose code and data can never be read. When an agent fulfills the blackbox property, it is considered to be autonomous in the sense that if a host executes that agent, the host cannot interfere with this execution in a directed way.

As shown in figure 5.6, the conversion mechanism that generates an agent with the blackbox property uses parameters called configuration parameters that allow creating different blackboxes out of the same specification. These parameters prevent what is called dictionary attacks. In such attacks, the attributes of the blackbox are guessed by converting a number of agent specifications and compare the created blackboxes with the attacked one.

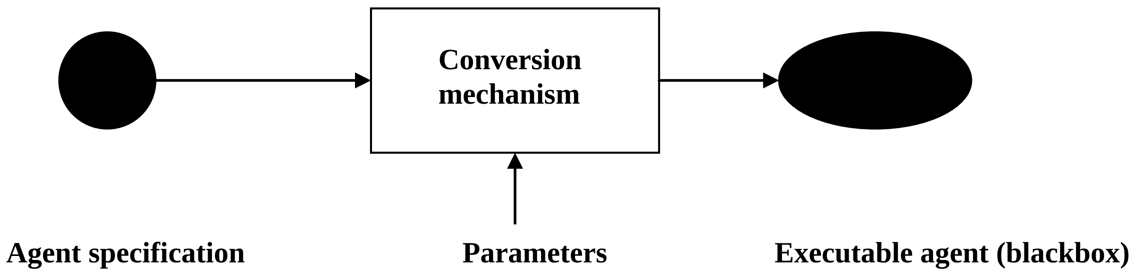


Fig. 5.6 - The Blackbox Approach

The blackbox protection does not hold forever. It only holds for a limited, known minimal time interval that is known in advance.

The central idea behind the time limited blackbox protection is not to allow an attacker to build a mental model of the agent in advance and to make the process of building this model a time-consuming task. In other words, we do not want to allow the attacker to think about the agent's code so that it figures out how it works and finds certain points in the code or values that are interesting to it **[Hoh98]**. In order to prevent the attacker from building a mental model in advance, a new form of the agent is created dynamically in an unpredictable manner at the start of the protection interval. And in order to make the process of building the model time consuming, conversion algorithms are used to produce new forms of the agent that are hard to analyze by the attacker. In other words, it should take the attacker as much time as possible to analyze the code. Here comes the role of the mess-up algorithms mentioned above.

5.3 Summary

Chapter five presented multiple partial solutions for the problem of protecting an agent against possible malicious host attacks. These solutions include simple ones like legal protection, hardware solutions and trusted machines, beside more complex solutions. The complex solutions included: partitioning, replication and voting, components, self authentication, migration history, audit logs, read-only state, append only logs, encryption, PRAC, code mess up, Limited life time of code and data and finally, the time limited blackbox security approach.

Most of these solutions are either partial solutions, that is they do not provide full protection of the agent, or unimplemented ones. In the proposed solution of this thesis, we provided a powerful combination of some of these methods that ensures a full protection of these methods.

Our proposed solution is based on a combination of a new code mess up algorithm presented, plus encryption, and time out. This in addition to another proposed solution that utilizes the advantages of the append only logs and read-only states. These methods, if used alone, will not provide a trusted protection for the agent. For instance, code mess up alone can be tracked and discovered if there is plenty of time. This is the same for encryption, and many encryption algorithms were proven to be attackable. That is why, combining those methods together will present a powerful, full protection against malicious host attacks.

Chapter 6 - Programming Languages for Mobile Code

6.1 Introduction

Mobile code represents any program that travels on a heterogeneous network and that crosses protection domains and automatically executes upon arrival at the destination. Mobile code supports a flexible form of distributed systems where the desired non-local computations need not be known in advance at the execution site. This increases network efficiency and increases simplicity and flexibility of the network maintenance. Furthermore, loading code on demand rather than having all programs duplicated on all sites can significantly reduce the total storage time.

6.2 How to choose a programming language for a mobile program?

There are a number of common needs that have to exist in a programming language in order that it becomes a good candidate for use in writing mobile agent programs.

- The first need is the need of *portability*. The idea of mobile code is that it moves between numerous heterogeneous execution sites. Thus, it is totally unfeasible to have a specific version of the code for every possible architecture. That is why portability is needed.
- The second need is the need for *safety*. For instance, a bug in a certain application should not affect the execution of other independent parts of the environment. Also, special protection should be provided for code arriving from un-trusted or unknown sources.
- The third need is the need for *security*. As the mobile code crosses protection domains, special care must be taken to protect against security threats presented by mobile applications. Security and safety are not the same issue. Safety is mostly concerned with the behavior of systems in the presence of bugs, therefore safety is one of the prerequisites of security.

- The last need is the need for *efficiency*. What is needed here is minimal overhead for the measures taken to ensure the above three needs: portability, safety and security.

  Safety and security at the programming-language level is becoming one of the most important issues. In order to obtain the required safety, we use programs written in a safe programming language. Most of the programming languages available nowadays provide protection against low-level errors through mechanisms such as typing, restricted pointers, automatic memory management and array bounds checking.

  There are numerous languages that can be suitable for mobile code. These languages can be either general-purpose languages intended for general application development, like Java, Limbo, Objective Caml, and Obliq; or they can be special-purpose languages like Safe-Tcl and Telescript. This chapter covers some of these languages in brief, presenting some of their basic security features.

6.2.1 Java

Java is one of the class-based object-oriented languages that emphasizes portability and security. The applet model was created by Javasoft as an example of how to use Java for mobile code. Applets are simply small applications that are automatically downloaded and executed upon visiting a Web page containing them.

The Java language is a simplified variant of C++. It does not include all the unsafe and complicated C++ language features like the pointer arithmetic, unrestricted casts, unions, unstructured goto statements, operator overloading, and multiple inheritance. The Java language instead includes automatic memory management to guarantee against possible pointer errors that can result from manual memory management. In addition to that, array and string types in Java are built in with range check of all accesses. Exception handling was also added to Java and threads and serialization were added to support concurrency.

Since the Java language is a modern safe language, it guarantees that type and access rules are always respected. This allows the expression of a low-level security policy within the language itself. The classes and attributes in Java have several visibility rules. They can be private, protected or public, etc. These rules play an important role in the security issue. That is, the interface to local resources is provided by libraries and is protected by the visibility rules. The SecurityManager controls most of the resources that require dynamic access control like the file system and network access, which is a centralized security monitor. All the security-related methods are declared *final* so that applications and applets are forced to use the appropriate code. When a class is declared final it indicates that it prevents the derivation of subclasses of itself. This protection prevents malicious applets from redefining the method in a subclass. When a class is defined as final, it will enjoy a stronger protection and no body can define any new methods with access to protected attributes in this class.

An object of the class ClassLoader does the loading of the Java classes over the network. For further security, this object is created at startup and cannot be replaced by applets afterwards. The ClassLoader also maintains a unique name space for each network source that is separate from the name space for classes coming from the local file system. The problem is that class files loaded through the network can be tampered with, that is why the bytecode is passed through a bytecode verifier that checks that the object code respects the Java semantics. This is verified by ensuring that the bytecode is in a valid format, that the pointers are not forged, that access rules are enforced and that the parameters passed have the expected types.

Any distributed and portable application can take advantage of the Java language due to its increased security and flexibility. Applications of Java include: **[Tho97]**

- *JavaOS*- a complete operating system written in Java that offers portability and extensibility.
- *Jeeves*- a framework for extendible network servers.
- *Java Management API*- a framework for management of heterogeneous networks.

- *Java Electronic Commerce Framework*- a software point-of-sale terminal accessible by any Java-enabled browser.
- *Java Beans-a* component architecture for reusable software components.
- *Java Database Access API*- a uniform interface to relational databases.
- *Java RMI*- an API for implementing remote method invocation. This eases the creation of client-server applications and permits the creation of more traditional distributed systems.

Since Java is considered to be the “jet fuel” of mobile agents, it possesses numerous properties that make it a successful candidate language for implementing mobile agents. These properties include: **[Lan99]**

1. *Platform-independence*: One major characteristic of Java is that it is designed to operate in heterogeneous networks. Java applications can execute anywhere on the network due to the privilege of the byte code generation offered by the compiler. This byte code can be executed on any machine provided that the Java runtime system is present on that machine. Therefore, there are no platform-dependent aspects of the Java language, which allows us to create mobile agents with no required knowledge about the types of computers that they will run on.
2. *Secure Execution:* Since the primary goal of Java is to be used on the Internet and Intranets, security was a major issue to be considered while creating the language. For instance, Java supports a pointer model that prevents the overwriting of memory and data corruption. Also, Java bans illegal type casting and pointer arithmetic. In addition to that, most activities of viruses are stopped by preventing programs from accessing private data in objects that they do not have access to. And even if someone managed to tamper with the Java generated byte code, the available Java runtime system ensures that the code will not be able to violate the basic semantics of Java. Hence, the Java security architecture

makes it safe to receive untrusted or malicious agents because they will not be able to tamper with the host or access any private information without its approval.

3. *Dynamic class loading:* This feature allows the Java virtual machine to define classes during runtime and enables classes to be loaded via the network. It allows each agent to execute independently and safely from other agents by providing it with a protective name space.
4. *Multithread programming:* Java allows multithread programming in which each agent is allowed to execute in its lightweight process or thread of execution. This allows agents to behave autonomously. In addition to that, Java supports agent interaction by providing a set of synchronization primitives that are built into the language.
5. *Object serialization:* Java provides a built-in serialization mechanism that can represent the state of an object in a serialized manner that is detailed enough to reconstruct the object later. Object serialization is a key feature of mobile agents, and the serialized form of the object must be able to identify the Java class from which the object's state was saved and to restore the state in a new instance.
6. *Reflection:* Java is able to use, within security restrictions, the reflected fields, methods and constructors to operate on their underlying counterparts in objects. Reflection serves the need for agents to be smart about themselves and other agents.

Examples of mobile agents frameworks that were written using the Java language include Gypsy **[JAZ00]**, Concordia **[Con99]**, and Aglets **[Dal98]**. The Aglets framework was surveyed in chapter 3 and the Concordia framework will be surveyed in depth in the next chapter.

6.2.2 Limbo

Limbo is a safe imperative language inspired mainly by the C programming language. But in addition to what is in C, it includes declarations as in Pascal, as well as abstract data types, first-class modules, first-class channels, automatic memory management and preemptive scheduled threads. And like Java, it does not support pointer arithmetic and casts. Limbo does not support inheritance for abstract data types (ADTs). In Limbo, the declaration of a module identifies the type of exported functions and contains the declarations of the ADTs, simple type declarations and constants. A module has to be instantiated before loading it. Loading an implementation of the module at run-time using a function called *load()* does this. This function takes a module type and a path to the module implementation and returns the instantiated module. This allows the program to choose among several implementations of a given module at runtime. The channels of Limbo allow the communication of any value of its declared type. A channel can be directly connected to another process, or through a library call to a named destination. The Limbo channels are the only built-in primitives for interprocess communication.

Security in Limbo comes with its restricted pointers and automatic memory management. Pointers can point to any object but cannot point to inside arrays, and there is no pointer arithmetic or arbitrary pointer conversion.

6.2.3 Telescript – the Language

The Telescript language is an object oriented, class-based wrapping language. The interfaces to the stationary applications are written in C or C++, and the communication interface is written in the Telescript language. The language itself is compiled into byte-code for portability and is executed by engines, which contain the interpreter and represent the runtime environment **[Dal98]**. In addition to that, Telescript is designed mainly for network programming. It is a specialized language for communication, thus it is not a general-purpose language. A Telescript program is basically a collection of hierarchically organized classes **[Val96]**. The agent is the central concept in Telescript. It travels autonomously on the Telescript network of engines called *Telesphere* doing business on behalf of its owner. The engine of the language is an interpreter with a collection of built-in classes and an engine *place*. A *place* is a stationary process that can accept incoming traveling agents. Users can create their own places nested within other existing places.

Telescript includes runt-time typing. It supports class inheritance. Classes can inherit from a single superclass and any number of mix-ins which are abstract classes that can not be instantiated. The use of classes in Telescript can be restricted in two ways. First, a sealed class cannot be specialized, and abstract class cannot be instantiated. Furthermore, as in other object-oriented, class-based languages, attributes of objects can be either private or public.

The Telescript agents are processes with several properties including:

a. The *Telename*: a pair of an authority and a process identity that together names a process. The authority identifies the Telescript user.

b. The *Owner:* It is the process that will own future created objects. It is usually the current process. Objects that are not owned by any process are garbage collected.

c. The *Sponsor*: It is the process whose authority will be attached to and charged for new created objects.

d. The *Permit*: It specifies the capabilities of the current process.

In telescript, agents are sent by invoking their go operation with a ticket. The ticket specifies the destination place and the route to be followed to reach this address. If the agent's authority and permit were accepted by the destination, the agent will be sent with its objects to the place and resumes execution within the new place.

Security is a major concern that affects most of the Telescript language. And the permission model adopted by Telescript aims to ensure security and moderate resource consumption as well. Also, mobile processes in Telescript are run in a separate domain and can only interact with the engine in which they run and that mediates all the interprocess access. Telescript is one of the languages that try to deal with denial of service attacks. The agents of Telescript are considered more powerful than Java applets in that they have their own initiative to travel. But in the mean time they can be more dangerous because once they are launched they can be harder to control.

6.2.4 Safe-Tcl – the Language of Agent TCL

Safe-Tcl is based on Tcl, a procedural script language that is characterized by being simple, portable, easily embedded and powerful. Every value in Tcl is represented as a string, even the programs themselves. This is what gives Tcl the simplicity character. Furthermore, Tcl scope rules are very simple. Only two scope levels exist: local and global (with respect to a function).

Since Tcl does not have any pointers, casts or unchecked array accesses, it is considered to be a safe language. Safe-Tcl was aimed primarily to be a safe and secure programming language. That is why every language construct and primitive that existed in Tcl was carefully examined and dangerous primitives were removed and replaced by more specific ones. For instance, the file system access functions were considered to be dangerous therefore they were replaced by an isolated global configuration space that is accessed through two functions: *SafeTcl_setconfigdata* which associates a string to a key, and *SafeTcl_getconfigdata* which returns the string associated with a key.

Two interpreters are used by Safe-Tcl. One only for Safe-Tcl running the untrusted applications, the other for full unrestricted Tcl. The untrusted application can interact directly only with the Safe-Tcl interpreter. **[Tho97]**

6.3 Summary

This chapter displayed four major mobile agent languages which are: Java, Telescript, Safe-TCL, and Limbo. The language chosen for implementation in this thesis is the Java language. This choice came after surveying the possible available languages and studying their advantages and drawbacks. The choice of Java as the programming language in this thesis was based on the following features of Java:

1. *Platform-independence*: Java is designed to execute anywhere on the network due to the privilege of the byte code generation offered by the compiler. Therefore, we can create mobile agents without considering the types of computers that they will run on.

2. *Secure Execution:* Due to the supported Java pointer model, overwriting of memory and data corruption is prevented. Also, Java bans illegal type casting and pointer arithmetic. In addition to that, java supports security libraries that provide encryption and digital signatures which is what we need in the implementation in this thesis.
3. *Independent execution:* This feature allows each agent to execute independently and safely from other agents by providing it with a protective name space.
4. *Multithread programming:* Java allows multithread programming in which each agent is allowed to execute autonomously and independently from any other agent. And this is what we need to build the timer.

Chapter 7 – Implementation Tool

7.1 Introduction

Concordia is a product of Mitsubishi Electric Information Technology Center America **[Con99]**. It is a software framework for developing, running and administering mobile agents. Concordia mobile agents are Java programs, which travel in the network to perform their function.

Concordia is considered to be the next generation middleware infrastructure for developing distributed applications. It makes distributed application programming easy and greatly reduces development time. In addition to that, Concordia is a full-featured framework for the development and management of network-efficient Mobile Agent applications that extend to any device supporting Java. It consists of multiple components, all written wholly in Java, which are combined together to provide a complete, robust environment for applications.

7.2 Concordia Architecture

The Concordia system, as shown in figure 9.1 below, is made up of a Java VM, a Concordia Server, and at least one Agent. The Java VM can be on any machine: it is a standard environment. The Concordia Server is a Java program which runs there, and at any other node on the network where agents may need to travel. The agent is also a Java program which the Concordia Server manages, including its code, data, and movement.

Usually, there are many Concordia Servers, one on each of the various nodes of a network, both user and server nodes. The Concordia Servers are aware of one another and connect on demand to transfer agents in a secure and reliable fashion. The agent initiates the transfer by invoking the Concordia Server’s methods. This signals the Concordia Server to suspend the agent and to create a persistent image of it to be transferred. The Concordia Server inspects an object called the Itinerary, created and owned by each agent, to determine the appropriate destination. That destination is contacted and the agent’s image is transferred, where it is again stored persistently before being acknowledged. In this way the agent is given a reliable guarantee of transfer.

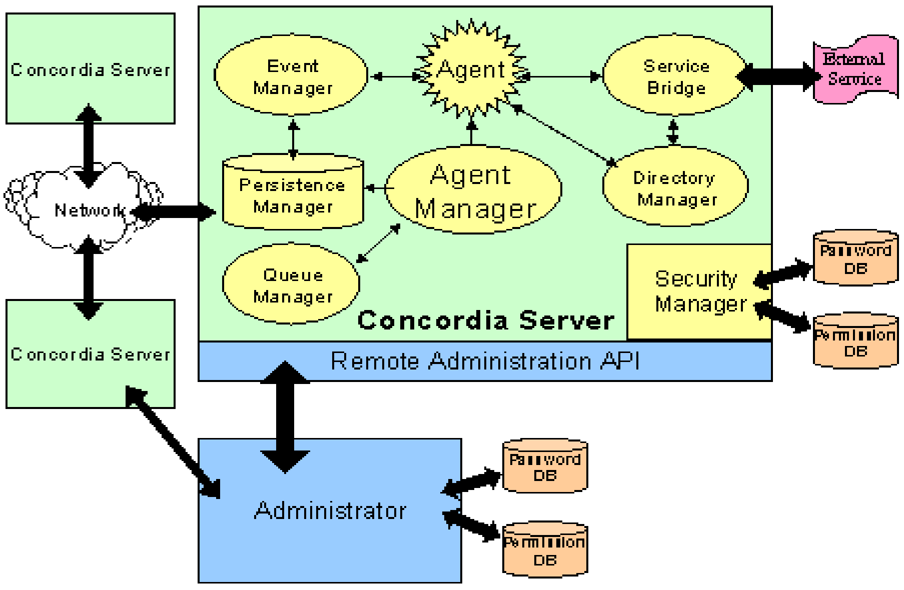


Fig. 7.1 - Concordia Architecture

After being transferred, the agent is queued for execution on the receiving node. This happens promptly but possibly subject to certain administrative constraints. When the agent again begins executing, it is restarted on the new node according to the method specified in its itinerary, and it carries with it those objects which the programmer requested. Its security credentials are transferred with it automatically and its access to services is under local administrative control at all times.

The work that the agent performs depends on its purpose, that is, the code that it was programmed to execute. Generally, agents have several components, just as any program has. An agent might start interactively, by prompting the user for search information, then may travel to a server to perform the query. Or, the agent may simply be a kind of remote demon, such as a mailbox filter or notification sender. As its methods complete, the itinerary causes the agent to be moved to other Concordia nodes. Therefore agents with different purposes will typically have different itineraries.

In all cases, the Concordia agent is autonomous and self-determining in its operation. In this way, it is unique since it is in control of its own itinerary.

The Concordia system is made up of numerous components, each of which integrates together to create the full Mobile Agent framework. The Concordia Server is the major building block, inside which the various Concordia Managers reside. Certain Concordia components have a user interface, such as the Concordia Administrator. In any case, each Concordia component is responsible for a portion of the overall Concordia design, in a modular and extensible fashion.

7.3 Concordia Components

All Concordia components are coded completely in the Java language.

7.3.1 Concordia Server

The Concordia Server is the name of the complete Concordia component installed and running on a machine in the Concordia network. It is comprised of manager components as follows.

7.3.1.1 Agent Manager

The Agent Manager provides the communications infrastructure that allows for agents to be transmitted from and received by nodes on the network. It abstracts the network interface in order that Agent programmers need not know any network specifics nor need to program any network interfaces. The Agent Manager Server also manages the life cycle of the agent. It provides for agent creation and destruction, and provides an environment in which the agents execute.

7.3.1.2 Administrator

Administration of the Concordia network is provided by the Administration Manager, in cooperation with Concordia services running on the various nodes under administration. The Administration Manager manages all of the services provided by Concordia, including

Agent Managers, Security Managers, Event Managers, etc. The Administration Manager supports remote administration from a central location, so only one Administration Manager is required in the Concordia network, although more can be employed as desired. The Administration Manager has a user interface component that is its primary means of use.

7.3.1.3 Security Manager

The Security Manager is responsible for identifying users, authenticating their agents, protecting server resources and ensuring the security and integrity of agents and their accumulated data objects as the agent moves among systems. The Security Manager is also responsible for authorizing the use of dynamically loaded Java classes which satisfy the needs of agents. The Security Manager has a user interface component, in order to configure and monitor the security attributes of the various users and services known to Concordia. This user interface function is integrated into the Administration Manager interface.

Security credentials used by the Security Manager may come from a number of sources. For secure, self-contained systems, it may be that no credentials are needed. For systems that traverse public or semi-public networks, encryption may be required but credentials may need only reflect user identity, such as user name or group id. For fully fledged agent systems deployed on the Internet, strong authentication and security can be provided from external authorities. All these security levels can be supported by Concordia’s Security Manager.

7.3.1.4 Persistence Manager

The Persistence Manager maintains the state of agents in transit around the network. As a side benefit, it allows for the checkpoint and restart of agents in the event of system failure. Additionally, it can checkpoint objects upon request by agents, to provide finer granularity of reliability guarantees for critical procedures. The Persistence Manager is completely transparent in its operation, that is, neither the agents nor the administrator need control or monitor its operation. However, management access is available if needed.

7.3.1.5 Event Manager

The Event Manager handles the registration, posting and notification of events to and from agents. The Event Manager can pass event notification to agents on any node in the Concordia network. The Event Manager works in conjunction with the Concordia Server to distribute events as needed. An important function of the Event Manager is to support Concordia agent collaboration.

7.3.1.6 Queue Manager

The Queue Manager is responsible for the scheduling and possibly retrying the movement of agents between Concordia systems. These features include the maintenance of agents as they await the opportunity to perform their work, maintaining their persistent state as they enter and leave a system, and retrying as necessary when Concordia systems are disconnected from the network. The Queue Manager provides the mechanism for prioritizing and managing the execution of agents on entry to Concordia nodes.

7.3.1.7 Directory Manager

The Directory Manager provides naming service in the Concordia network. The administrator may configure the name service in a number of ways, chosen according to the needs of the programmer and services. The Directory Manager may consult a local name service or may be set up to pass requests to other, existing name servers.

7.3.2 Service Bridge

The Service Bridge provides the interface from Concordia agents to the services available at the various machines in the Concordia network. It comprises a set of programming extensions to provide access the native API’s as well as interfacing these to the Directory Manager and Security Manager.

7.3.3 Agent Tools Library

The ATL is a library which provides all the classes needed to develop Concordia Mobile Agents. This of course includes the ***Agent*** class, and others derived from Java base classes, with interfaces to the Concordia infrastructure.

Concordia provides all the basic functionality of a mobile agent system:

- Efficient communication
- Easy distributed applications development
- Asynchronous and autonomous agents
- Flexible, scaleable development of distributed systems

7.4 Distinguished Concordia Features

Concordia provides the following features which help to set it apart from other mobile agent systems:

- Concordia agents can adapt dynamically, they can change their travel and execution plans dynamically based on communication with other Concordia agents.
- Concordia agents are naturally heterogeneous. They are platform and transport-layer independent
- Concordia agents are robust and fault-tolerant. This provides an advantage over wireless alternatives.
- Concordia agents can easily integrate with legacy applications. The Application gateways provide a bridge to legacy applications.
- Concordia provides comprehensive security support. Its Agents are protected in transmission from one host to another via encryption. Each agent carries the identity of the user that created it, and the operations the agent requests are subjected to the same user's permissions. Each agent is securely transmitted across the network, and no additional code is required to provide for secure, distributed operation.

- Concordia Agents can only access server resources that have been authorized by the systems administrator.
- Concordia is written in Java, therefore it is portable, even ubiquitous. It runs on platforms large and small, and integrates easily with existing applications and frameworks.
- Concordia agents provide for mobile applications. Agents support mobile computing as well as off-line processing and disconnected operation. These applications are in turn written with little or no knowledge of the underlying communications that they will employ. Concordia both hides the details from the programmer and user, as well as allows the agent to adapt to its environment and administration.
- Concordia agents are reliable. All Concordia agents are checkpointed before execution by the Persistence Manager, and they may return to these checkpoints if necessary. Objects the agents may create are checkpointed as well. Coupled with the services of the Queue Manager while they are being exchanged across the network, Concordia agents are assured of reliability at every stage of their operation.
- Concordia agents can collaborate. The concept of collaboration is important and useful to the agent programmer. It can provide a number of benefits, such as enabling parallel operation over multiple servers or multiple networks. It can divide a task into suitable pieces, and these pieces can be carried out in the most appropriate places. The results of these sub-tasks are then assembled by collaboration. The collaboration framework then permits a decision to made based upon the results, which can be used to determine destination, action, or other appropriate behavior.

7.5 Concordia Expanded Agents' Functionalities

With Concordia, the agent's functionality is expanded to include:

Mobility – The agents can travel from host to host on a network as determined by the application or user, performing various sub-tasks on each host in its path. Computations are performed closer to the data source, reducing network connect time and costs.

Security - Agents are only allowed to access resources for which they have been authorized. Concordia uses advanced encryption techniques to provide tamper-resistance to agents while they are on a device and when traveling across networks.

Persistence – The agents transparently resume computations that are affected by system or network interruptions or failures. An agent's internal state information is maintained as it travels across the network. This feature is not available in the freeware version.

Collaboration – The agents cooperate with each other to perform tasks. An application can be composed of several agents with specific sub-tasks.

Disconnected Computing – The agents continue to perform their assigned tasks even when the user is not connected to the network.

7.6 Concordia as a Powerful Framework for Developing Network-Efficient Mobile Agent Applications

Concordia delivers a rich set of features that support the implementation of agents for a wide range of industries and end users. Agents implemented with Concordia are mobile objects. They can travel to different locations and hosts on a network and perform work at those locations. The mobility of Concordia Agents sets them apart from distributed objects and Java applets.

Concordia hides the complexities of application program mobility from programmers, so developing an Agent-enabled application is similar to developing a non-mobile or stationary program. Agents maintain their internal state while traveling in a network, so they are able to resume execution upon arrival at a new location. All Agent transport work is handled transparently without programmer intervention.

A Concordia Agent's network travels are defined by its Itinerary. The Itinerary specifies where the agent is to travel and what task it should perform when it arrives.

Concordia Itineraries are specified at run-time. Agents may change their own Itineraries based upon information and events discovered as the agent travels.

Concordia's powerful capabilities help developers deliver complete, anytime-anywhere information access systems. Users of these systems can gain access to information regardless of their location, mobility, or network connection. By shielding developers from the complexities of network communications, Concordia reduces development time.

7.7 Concordia Technology at a Glance

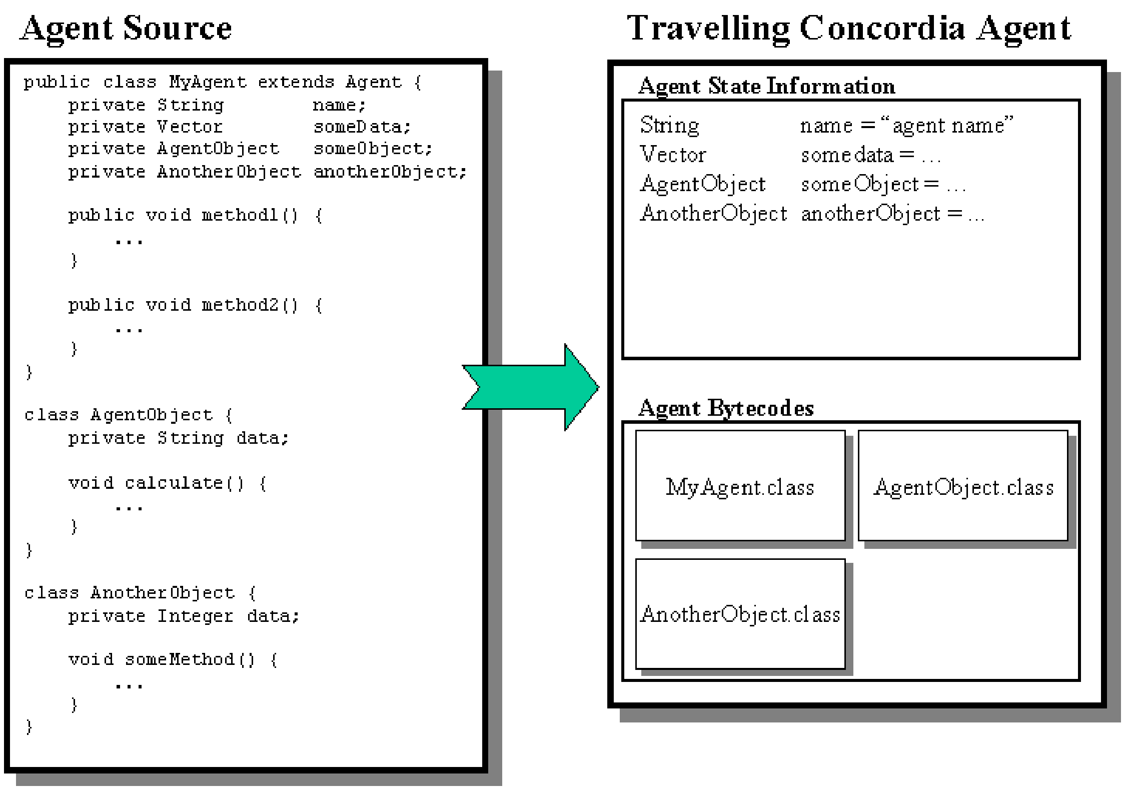
Features	Benefits
Java Language Support	• Platform independence lets applications work with wide variety of devices. • No special programming tools required to develop Concordia Agents. • Use standard Java Communication protocols. Applications developed with Concordia Agents maintain the benefits of the rich networking features of the Java language. • Reduce product development time.
Agent Collaboration	• Multiple agents work together to perform complex tasks more efficiently. • Parallel application execution over multiple devices improves overall execution efficiency. • Simple API eases implementation of systems requiring complex coordination of tasks.
Mobility	• Processing of data close to the data source. • Reduces network traffic congestion.
Transparent Network Communications	• Shields developers from complexities of network communications, resulting in shorter development time.
Agent Cloning	• Agents can replicate themselves to complete a task more quickly.
Java Database Connectivity (JDBC) Compatible	• Standard SQL database access interface for uniform access to a wide range of relational databases.
Dynamic (Network) Class Loading	• Developer flexibility in code distribution. • Improved network efficiency.
Queuing Support	• Buffering of transmissions alleviates problems with transmission delays. • Reliable transmission of agents and information.
Disconnected Computing	• Reduces communication costs for remote and mobile workers. • Increases worker productivity.
User-Specifiable Itinerary	• May change itinerary without modifying agent source code.

Agent-Specifiable Itinerary	• Agent-Specifiable Itinerary
Persistence	• Automatic resumption of interrupted computations caused by system or network failures. • Agent internal state information maintained across the network.
Security	• Protects system resources from malicious agents. • Tamper-resistant Agents.

7.8 Agent Mobility Overview

Concordia Agents have the ability to travel through a computer network. At programmer discretion, an agent can be instructed to move from one machine to another. During its travels, an agent maintains its internal state, meaning any information it retrieves along its travels is remembered when it arrives at a new destination. More exactly, the state of all of an agent's member variables persists through each network transmission. The work of transporting the agent is handled transparently by the agent system.

Concordia Agents are themselves Java objects. As objects, agents are composed of both data (the agent's state) and executable code (the agent's Java bytecodes). Moving an agent around a network requires the movement of the agent's state as well as its bytecodes. Thus a traveling agent can be pictured like the following:



```
public class MyAgent extends Agent {
    private String        name;
    private Vector        someData;
    private AgentObject   someObject;
    private AnotherObject anotherObject;

    public void method1() {
        ...
    }

    public void method2() {
        ...
    }
}

class AgentObject {
    private String data;

    void calculate() {
        ...
    }
}

class AnotherObject {
    private Integer data;

    void someMethod() {
        ...
    }
}
```

Fig. 7.2 - A Running Agent

The fact that Mobile Agents move both their state and bytecodes around a network makes them very different from either distributed objects or web applets. In the world of distributed object systems, such as CORBA or DCOM, stationary objects can use the network to remotely call the methods of other objects. While two objects communicate across the network, neither object actually moves. World Wide Web applets provide a mechanism for bytecodes to be downloaded from a web server to a web browser, but provide no mechanism for any state information to travel. An applet object does not travel from the server to the client, rather the applet bytecodes are downloaded, in exactly the same way as an HTML page is downloaded, and then those bytecodes are used to create an object. Once the applet is created, it stays within the browser and cannot move.

The fact that agents are mobile also means that the mechanisms needed for retrieving and loading of bytecodes are different than in the case of Java applications and applets. In the above example, MyAgent contains two member variables of type AgentObject and AnotherObject (figure 7.2). The bytecodes for these classes would be stored on a computer filesystem, in files named AgentObject.class and AnotherObject.class. Since Concordia Agents travel, it is possible for an agent to travel to a machine where the AgentObject and AnotherObject classes are not installed and the AgentObject.class and AnotherObject.class files do not exist on the local file system. For example, in a development environment, such as the one shown in figure 7.3, it is certainly possible that the bytecodes of an agent under development would not have been installed on a server used for testing.

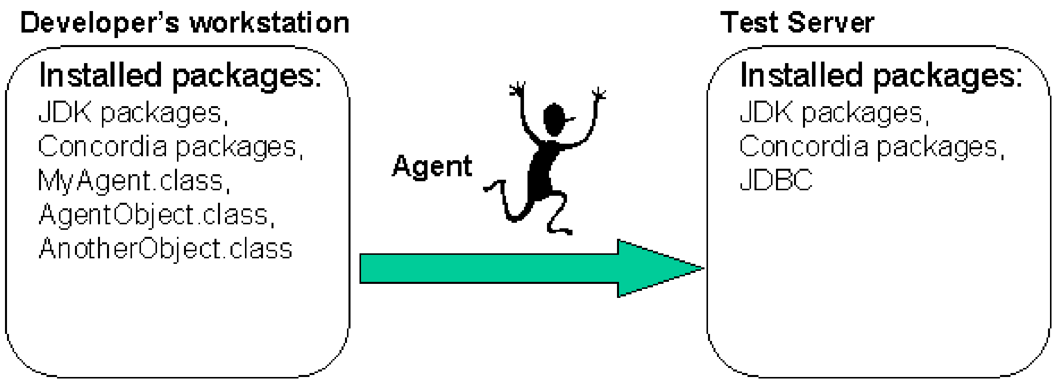


Fig. 7.3 - Class Loading Scenario

This scenario can lead to a potential problem in that if the MyAgent creates an object of type AgentObject or AnotherObject while executing on the test server, a mechanism is needed to move the bytecodes from the developer's workstation to the test server. To handle such cases, Concordia utilizes a hybrid push-pull model for moving of bytecodes.

Concordia's pull model operates in a manner very similar to the way that a web browser downloads the bytecodes for an applet. Every agent carries a URL pointing to the location of its codebase on its home machine (referred to as its homeCodebaseURL). In most cases this codebase will be a location on a web server. If the Concordia Server discovers that the bytecodes required by an agent are not present on its local file system, the server sends a network request to the codebase to load the bytecodes. In most production cases, this is an HTTP request to a web server. Concordia also incorporates a automatically pushed with push model in that once a class has been downloaded, it will, from then on, be pushed with agent's bytecodes around the network as depicted in Figure 7.2.

In some cases, a simple pull model for class loading is not appropriate. For example, if an agent is going to travel far from its home machine, across a WAN or the Internet, a network request to load classes can be slow and very expensive. If an agent travels through a firewall, it may be impossible for it to send a request back to its codebase. To handle such cases, Concordia allows the user or developer to specify a list of related classes when launching the agent. The related classes automatically travel with the agent and are always available to it during its travels. In the example shown in Figure 7.2 and Figure 7.3, if the Concordia developer included AgentObject and AnotherObject in the related classes list, then the bytecodes for these classes would be pushed with the agent to each server to which it travels. If the related class list contains a complete list of all the classes the agent needs during its travels, then thehomeCodebaseURL does not need to be specified since no requests to the codebase are necessary.

Performance of an agent application can be tuned using the related class list. The more bytecodes that travel with the agent, the more network resources are needed to transmit the agent. However, network requests to retrieve code, also consume network resources. In general, if the agent will be travelling across a WAN, wireless network, or the Internet, reliability can be gained by specifying a fairly complete related classes list. If a class will definitely be needed by an agent at each stop in its travels, then pushing that class with the agent will yield the best performance. If a class may not be needed by an agent (i.e. an instance of the class is only created under exceptional circumstances) and the agent is traveling in a high speed, reliable network, then the pull model can yield the best performance.

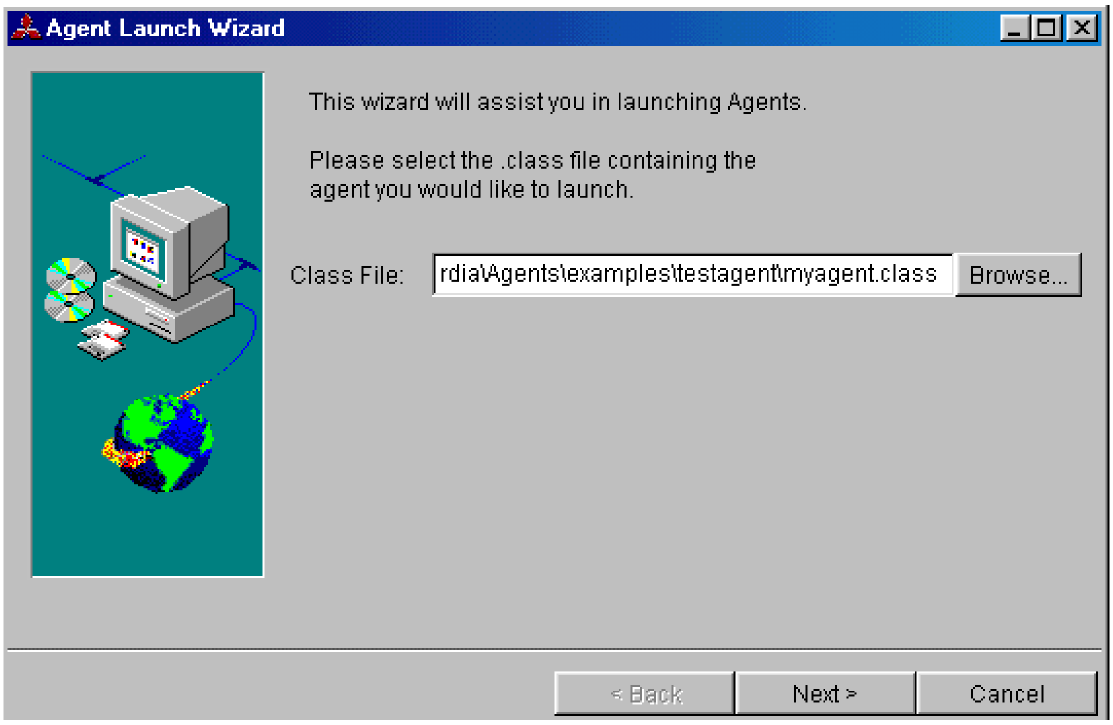
An agent's Itinerary defines where it travels on the network . The Itinerary specifies a list of Destinations, each of which specifies where the agent is to travel and what work it is to do when it arrives there. Within a Concordia system, where the agent travels is specified as the hostname of the particular server, and what work to accomplish is specified as the name of a method of the agent which should be allowed to execute on that machine.

An agent's travel and execution is managed by an object called the AgentManager which runs as a component of the Concordia Server. The AgentManager handles **[Con99]**:

- The transmission of the agent across the network
- The reception of agent
- Provides the thread in which the agent executes
- Initiates the agent's execution.

7.9 The Concordia Wizard

In order to help agent creators to launch their agents, Concordia provided a wizard that guides the developer step-by-step to launch the agent. The three figures below show the three steps of the Concordia wizard to launch the agent. Fig 9.4 shows how the user can enter the agent class file path. In figure 9.5, the wizard prompts the user to choose any class files to be sent with the agent. Finally, Fig. 9.6 asks the user to enter the addresses of the hosts to be visited by the agent and the functions to be called by each host. Hence, the Concordia wizard facilitates the developer’s task to launch an agent.

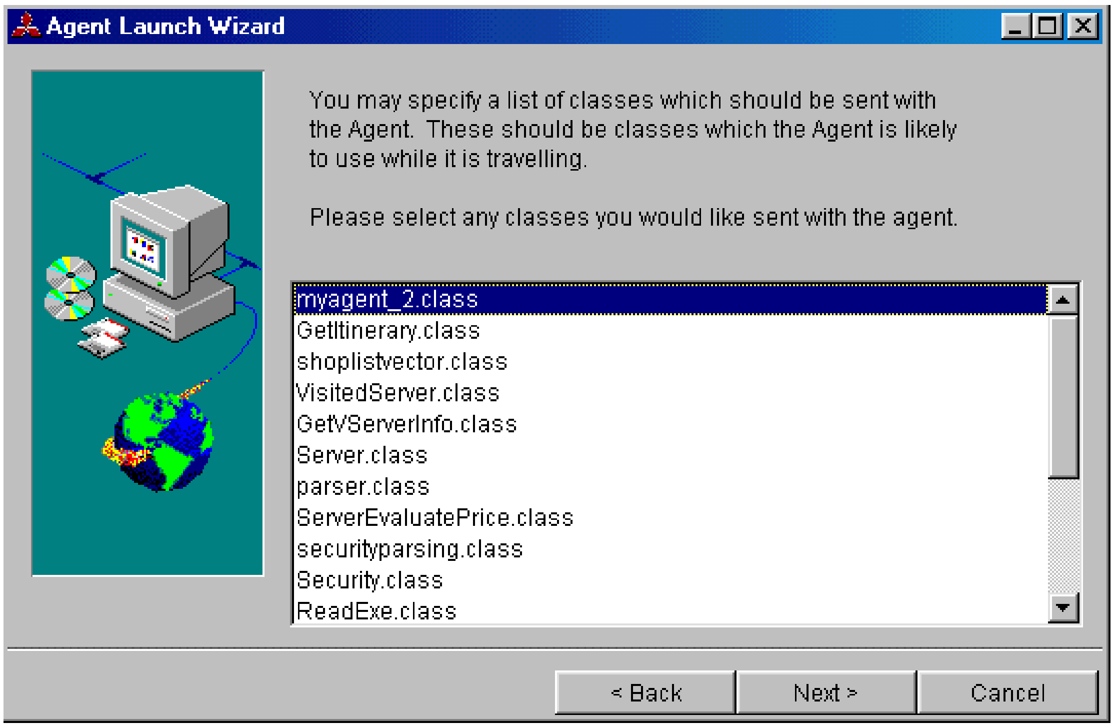

Agent Launch Wizard

This wizard will assist you in launching Agents.

Please select the .class file containing the agent you would like to launch.

Class File: rdia\Agents\examples\testagent\myagent.class

Fig. 7.4 Choosing the agent class file

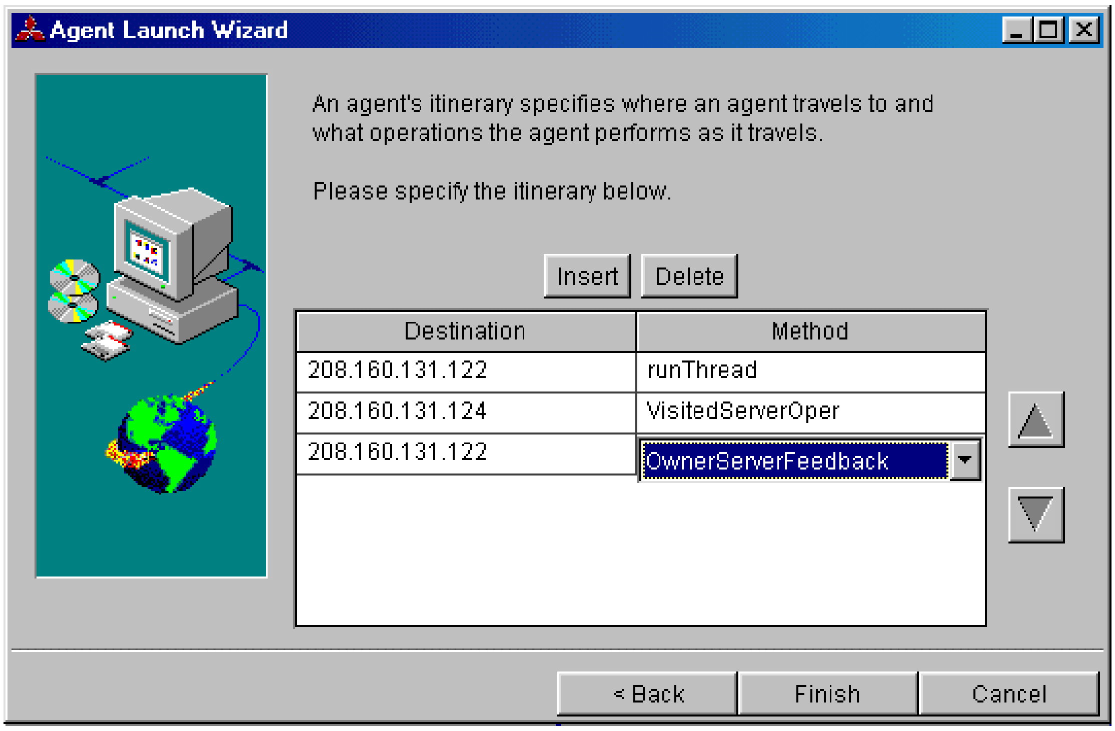

Agent Launch Wizard

You may specify a list of classes which should be sent with the Agent. These should be classes which the Agent is likely to use while it is travelling.

Please select any classes you would like sent with the agent.

myagent_2.class
GetItinerary.class
shoplistvector.class
VisitedServer.class
GetVServerInfo.class
Server.class
parser.class
ServerEvaluatePrice.class
securityparsing.class
Security.class
ReadExe.class

Fig. 7.5 Choosing any class files to be sent with the agent


Agent Launch Wizard

An agent's itinerary specifies where an agent travels to and what operations the agent performs as it travels.

Please specify the itinerary below.

Destination	Method
208.160.131.122	runThread
208.160.131.124	VisitedServerOper
208.160.131.122	OwnerServerFeedback

Fig. 7.6 Specifying the Agent's Itinerary

7.10 Summary

In this chapter**,** an overview of Concordia is given in order to pinpoint its strengths, structure, characteristics and advantages. As it is clearly shown in this chapter, Concordia is chosen to be the working mobile framework for the proposed system due to many factors. These factors include:

- Concordia is a full-featured framework for the development and management of network efficient mobile agent applications.
- All Concordia components are written in Java which is a very powerful language for developing mobile agents.
- Concordia agents are secure, and each agent carries the identity of the user that created it.
- Concordia agents are reliable and they can collaborate together.

- The Concordia interface is user friendly and helps any user to launch his/her agents with no difficulties.

These are not the only advantages of Concordia but are the ones that helped in the experimentations for this thesis.

Chapter 8 - The Proposed Solution

8.1 - Introduction

As shown in chapter five, the previous solution techniques are partial solutions and do not guarantee full protection for the agent against possible malicious host attacks. In this chapter, we present as solution that involves a combination of code mess up, encryption and time out, The code mess up algorithm presented here is a new algorithm that uses the concept of variable disguising by altering the values of strings and numerical values.

8.2 Three-Tier Protection Mechanism

In order to protect the code of the agent from possible attacks of malicious hosts, a combination of code mess-up, encryption and limited lifetime of code and data (timing), will help preventing the attackers from modifying the agent's code. A *code-encryption transfer* module will take the agent's code, apply a code mess up algorithm to change the code's main features, and create a new version of the code which does the same tasks but with a totally different inner structure. Then, the code-encryption transfer module will encrypt the messed up code using an effective encryption algorithm. Each mobile agent will possess a timer that imposes a limited lifetime for its code and data. If the agent does not return to its owner server in the assigned time, the server will automatically kill this agent or recharge it, putting in mind that time limitedness requires synchronized clocks, and thus both the sending and receiving parties should have a correct global time.

8.2.1 Limited Lifetime of Code and Data

It is assumed in this design that attacks after the time interval allowed for each agent will not have effects on the agent. This way, the attacker will not be able to decrypt the code. And even if it manages to do so, it will take time to resolve the messed up code, and using a timer, if the time of the agent's execution was exceeded, the agent will be either automatically killed or recharged and thus the attacker will not have enough time to modify the agent's code. This timer will exist in each agent internally; it can be represented as a predefined number of ticks decided by the home host. If this number of ticks was exceeded, then the agent will be killed or recharged depending on the design. After the agent is executed, it will return back to its home host where it will be decrypted and the original code will be recovered using a *code-decryption transfer module.* Both the code Encryption/Decryption transfer modules will be located at the agent's host.

As was mentioned before, if an agent expires, it can either be killed or recharged. Killing the agent will end its task completely. Sometimes an agent is delayed due to network problems, so killing an agent when its time expires will prevent it from performing its intended task. That is why, agent recharging can be employed in order to allow the agent to migrate further. The problem with this approach is that the agent has to be assigned with a new expiration date and signed digitally by a party that the agent trusts. Also, the agent needs to get a new identity, and the tokens have to be replaced by new ones. All of these requirements have to be performed by a trusted host. The expired agent first checks the identity of the trusted host, and then it delivers the tokens that have to be replaced and gets new ones instead.

Actually, this Three-Tier protection mechanism is suggested in order to provide more guaranteed protection for the agent against the host's possible attacks. The code mess-up approach by its own is a powerful approach, but it has a weakness that might affect the security of the agent. This approach relies on changing the code's appearance. But if an agent "**A"** visits a specific malicious host **"H"** continuously, **"H"** might not be able to understand the code of **"A"** in the first few times because it has never seen that form of code before, and because time does not allow the attacker to discover it. But after several visits of **"A"**, **"H"** might develop a familiarity and an understanding of **"A"**'s code and might finally attack it. In other words, the code mess-up approach can be breakable if there is enough time or after several visits of the agent to the malicious host.

Including an expiration date for each agent can be achieved whether by using key certificates for each agent as shown in the previous chapter, or by including the expiration date into the constant part of the agent. In this thesis, timing is implemented by using threads. One thread launches the agent, and the other thread is kept in the home server. Both threads start together. The thread kept in the home server is the timer thread. It sleeps for “x” amount of time. When this time elapses, the thread in the home server wakes up and issues an alert to the home server that kills the agent and stops its functionality.

8.2.2 Code Mess Up

Code mess up involves creating a different messed up code that does the same task but with a different inner structure. Also, the data elements will have different obfuscated values. In other words, our aim is to develop a mechanism to transform a normal piece of code to a form, which is less readable and understandable. The mechanism to implement code mess up in this thesis involves three major parts:

- Inserting dummy code
- Altering the values of numeric variables.
- Altering the values of string variables.

First, Inserting dummy code involves adding dummy classes with functions that have nothing to do with the agent’s functionality. This aims to make the code more obscure and complex for any attacker to understand. In other words, if an attacker managed to attack the code, it will not find the direct agent’s code, rather, it will find a more complex version of the code with many functions and classes. This will force the attacker to spend more time in order to understand the flow of the agent’s code and its exact functionality.

Second, Altering the values of numeric variables involves taking every integer, float, double or any other numeric value and changing its value by multiplying it by a randomly generated seed that is kept in the home server in order to be used later to re-generate the original numerical value of the agent by an inverse operation.

Third, Altering the values of string variables involves taking every string value and changing it. For every index in the string, its numeric value is multiplied by a randomly generated seed that is kept in the home server in order to be used later to re-generate the original string value.

This way, the agent's code is greatly altered and the attacker will find a hard time to unmess up the code. And even if an attacker managed to unmess up the code, it will take a long time to do so, and here comes the role of the timer to do that.

8.2.3 Encryption

Encryption by itself is a very powerful protection mechanism. There are several encryption algorithms that currently exist, we presented some in chapter four. The encryption algorithm used in the implementation is the DES algorithm. This algorithm is proved to have a reasonable key length, and is supported by the Java security classes.

The main aim of using encryption is to protect the agent's important information. In this thesis, the agent's important data is mainly stored in the list of prices that the agent collects from host to host. Thus our main goal is to protect this list using encryption. This will be explained in detail in chapter 10. But again, Encryption alone does not guarantee a full protection for the agent.

Let's put the assumption that this encryption algorithm can be attacked. In this case if encryption was used alone then the agent's data will be simply read and may be altered. But if we added code mess up, then the attacker will be faced with the problem of the messed-up code. And thus, these two methods of encryption and code mess up are complementary to each other putting in mind that each agent's existence in any other host than the home one is timed, and if this time is expired, the agent will be either automatically killed or recharged.

The two figures below show a demonstration of the proposed solution. Figure 8.1 shows a block diagram of the proposed solution. The agent's host will contain the agent's code. The host will apply the mess-up algorithm to the code then encrypt it and the agent moves to the receiving host, performs its operations then returns back to its host where its code is decrypted and un-messed up. The agent itself is timed so that if it exceeds a specific time without returning back to its host, the host will kill it. This is shown in the diagram by including the receiving host inside a timer frame. Figure 8.2 shows a detailed description of the operations performed inside the code-encryption and code-decryption transfer modules.

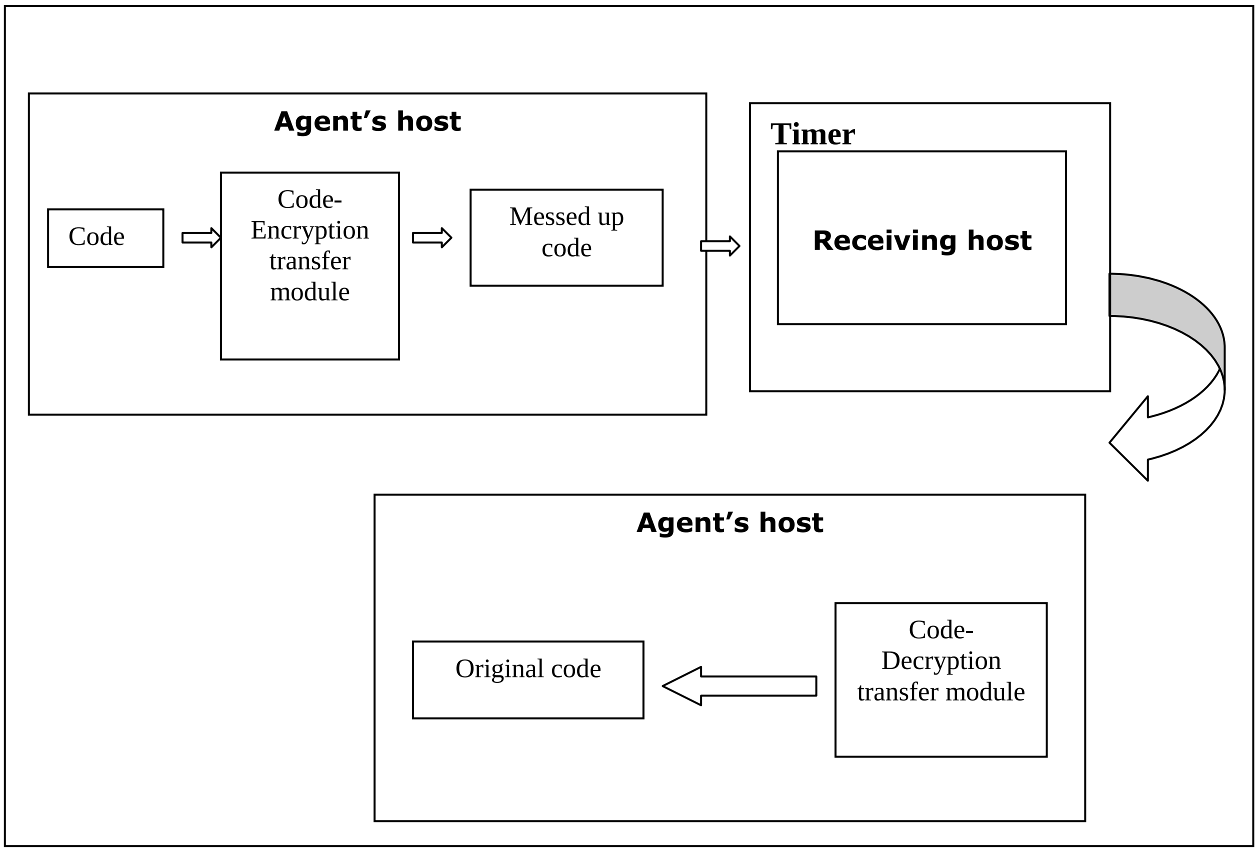
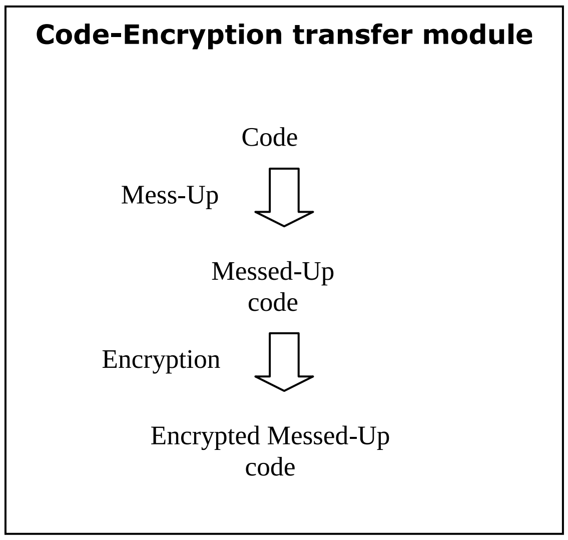


Fig. 8.1 - Diagram of the Proposed Solution



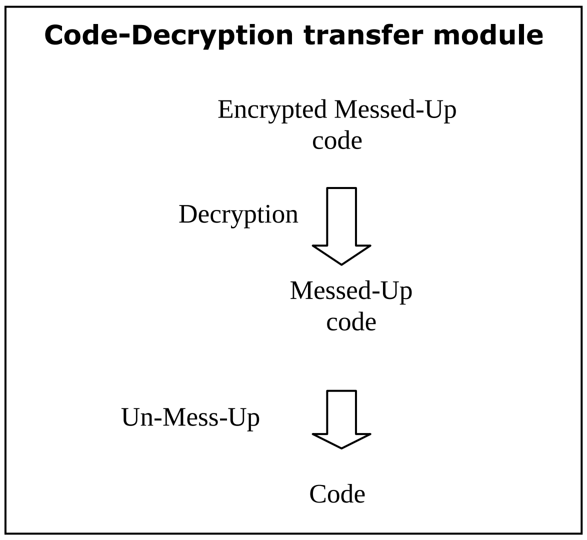


Fig. 8.2 – Encryption/Decryption transfer modules

8.3 Summary

This chapter presented a three-tier solution for the problem of protecting agents from possible malicious host attacks. This solution is a combination of code mess up, encryption and time out.

Chapter 9 - Testing and Security Assessment

9.1 Introduction

Actually, it is unrealistic and almost impossible to believe that a system can be completely, 100% secure. Even if that was theoretically possible, the nature of verification "limits the confidence that we can have it" **[Wul98]**. But there will always be something we can do to increase our confidence in the security of our system. Normally, any system designer should strive for the best verification possible. That is why verification resources should be applied to areas with maximum expected returns, and only when the verification resources are exhausted, the developer can be sure that his/her system is secure given the verification resources and development constraints.

Software trustworthiness is very difficult to define and is used in many different ways by different organizations, that is why many programmers see that this word is only associated with security. On the other hand, other organizations define trustworthiness as the degree to which software engineering techniques are used to reduce the possible occurrences of errors in the development and maintenance lifecycle **[Edw94]**.

9.2 Non-Interactive Evaluation of Encrypted Functions

Since the first model presented in this thesis is a trio model, that is it consists of a code mess-up step, an encryption step and a timer, we should find a way in order to prove that this model is trustworthy. In order to do that, we will start by the encryption step.

The ability of hiding a function *f* inside an executable program is of great importance for mobile agent protection. This is because if code privacy is obtained, this will immediately yield code integrity. That is why we have to encrypt a function such that its encrypted form remains executable. The general form of non-interactive evaluation of encrypted functions is presented in **[Sat97]** as follows:

" Alice (the originating host) has an algorithm to compute a function f. Bob (the remote host) has an input x and is willing to compute f(x) for her, but Alice wants Bob to learn nothing substantial about f. Moreover, Bob should not need to interact with Alice during the computation of f(x). "

The protocol for the above description is as follows:

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).

This protocol is called non-interactive because it has no further interactions than the exchange of the program and the resulting value at the end. So, it has an optimal number of communication rounds.

A major question that was asked by several people who attempted to protect mobile agents is "Is there a public-key encryption function E such that both E(x+y) and E(xy) are easy to compute from E(x) and E(y)?"

If an encryption function having these properties exists, it would have allowed evaluating polynomial expressions in the encrypted data without revealing the input and the result. However, such a function has not been found until today, which is why the problem of evaluating non-interactively a polynomial for encrypted data is still open. Sander and Tschudin **[Edw94]**, mentioned that for computing with encrypted polynomials, it is not necessary to have both the additive and multiplicative property of an encryption function as mentioned in the above question; rather, it is enough that the encryption supports addition and "mixed multiplication". We explain this in the following definition:

"**Definition 1** Let R and S be rings. We call an (encryption) function E: R -> S

- *Additively homomorphic* if there is an *efficient algorithm PLUS* to compute E (x+y) from E(x) and E(y) that does not reveal x and y,
- *Mixed multiplicatively homomorphic* if there is *an efficient algorithm MIXED-MULT* to compute E (xy) from E(x) and E(y) that does not reveal x.

Proposition 2 Let E :R -> S be an additively and mixed multiplicatively homomorphic encryption scheme. Then we can implement non-interactive evaluation of encrypted functions (EEF) for polynomials p $\in$ R[X_1, …, X_s] with E.

Proof Let p be the polynomial $\sum \left(a_{i_1} \ldots i_s \overset{i_1}{\underset{1}{X}} \ldots \overset{i_s}{\underset{s}{X}} \right)$

1. Alice creates a program P(X) that implements p in the following way:

- Each coefficient $a_{i_1} \ldots i_s$ of p is replaced by E($a_{i_1} \ldots i_s$),
- The monomials of p are evaluated on the input x_1,…, x_s and stored in a list

  L:= […, ($\overset{i_1}{\underset{1}{x}} \ldots \overset{i_s}{\underset{s}{x}}$), …],
- The list M:=[…, E($a_{i_1} \ldots i_s \overset{i_1}{\underset{1}{x}} \ldots \overset{i_s}{\underset{s}{x}}$),…] is produced by calling MIXED_MULT for the elements of L and the coefficients E($a_{i_1} \ldots i_s$)
- The elements of M are added up by calling PLUS.

2. Alice sends the program P to Bob.

3. Bob runs P on his private input x_1, …, x_s and obtains P(x_1, …, x_s)

4. Bob sends the result P(x_1, …, x_s) = E(P(x)) back to Alice.

5. Alice decrypts the result by applying E^{-1} and obtains p(x). ”

The above protocol is not 100 percent secure although its proof sounds convincing. It leaks information about the coefficients of the unencrypted polynomials. This leakage is unacceptable for certain applications. As shown in the above proof, E(f) can be easily reconstructed from the program P. So, a set of monomials U containing the monomials with non-zero coefficients of f is inherently revealed by the protocol. The set U is called the skeleton of f. If the monomials are given in a certain order then f can be identified with the list of coefficients of f that is called m. Also, E(f) can be identified with its list of coefficients called E(m). So the message spaces M_n consist of n tuples of elements from Z / NZ on which the encryption function E operates. The above protocol does not leak any information about f except its skeleton. Unfortunately, the limitation of the above protocol is that there exist applications where the information leakage of the skeleton may allow recovering of the full polynomial f.

9.3 Evaluation of Encrypted Functions via Composition Techniques

Function composition is another way of realizing the evaluation of encrypted functions. For example, assume that Alice, the sender, wants to evaluate a linear map A at Bob's input x on Bob's (receiver) computer. Alice does not want to reveal A to Bob, so she picks an invertible matrix S at random, and computes B:=SA and sends B to Bob. Bob then computes y:=Bx and sends y back to Alice. Alice then computes the inverse of S and multiply it by y and obtains the result Ax without having disclosed A to Bob. To increase the security of the above scheme, the matrix example is generalized as follows: we assume that 'f' is a rational function, 's' is also a rational function that can be inverted and 'o' is the circle function. Let E(f) := s o f, then the security of this method is based on the difficulty of decomposing E(f), that is the difficulty of reconstructing f from E(f).

Actually, as presented in **[Sat97]**, it is insufficient to protect a signing function as an independent building block. What is needed is a way to stick the signature routine **s** to the function **f** that produces the output to be signed. In order to sign the output of a function **f** securely, Sander and Tshudin suggested the following idea: **[Sat97]**

We have a rational function **f**, and another one **s** that is used by Alice to produce the digital signature **s(m)** of an arbitrary message **m**. We want the message **m** to be the result of a rational function **f** applied to some input data **x**. Finally, we have a function **v** that Alice, the sender, publishes in order to let others check the validity of the digital signature. The function **z** is regarded as a valid signature of **m** if and only if **v(z) = m**. For letting the mobile agent create undetachable signatures, that is to glue the signature routine to the function f that produces the output to be signed, the sender computes a function called f_{signed}:= s o f. The sender sends both f and f_{signed} to the receiver who evaluates both f(x) and z = f_{signed}(x). When applying the verification function v to z, every user can check that the message f(x) is a valid output of the function f. So, the security of this method lies in the receiver's inability to construct s(m) for a given message m.

But again, the above-suggested scheme by Sander and Tshudin is not fully secure. For instance, the following four attacks appeared against the scheme.

1. *Left decomposition attack*: Given the rational functions h:=s o f and f, determine s.
2. *Interpolation attack I:* Since the function v is public, a list of pairs (z, v(z)) can be produced. And since s is a rational function, it is feasible to reconstruct s by interpolation techniques which is a realistic threat for the secrecy of s specially that s has to be a function of low degree to decrease the size of s o f.
3. *Interpolation attack II:* The pairs (l, s(l)) can be produced. L is obtained by using the output function f.
4. *Inversion attack*: If the receiver is able to find a preimage x of n under f (that is F(x) = n), then he can produce a valid signature z for n by computing z :=f_{signed}(x)). **[Sat97]**

9.4 A Multivariate Approach for the Evaluation of Encrypted Functions

The multivariate approach is another method that attempts to overcome the threats of the above mentioned interpolation attacks. The following notation is used (as presented in [Sat97]:

1. Let s = (s1, …, sk): R^k->R^k be a bijective function whose components are given by rational functions (where R is a ring or a field).

2. Let $v = (v1, \ldots, vk)$: $R^k \to R^k$ be the inverse function of s; that is $s \circ v = v \circ s = id_{R^k}$.

3. Let f: $R^l \to R^t$ be the function whose output the sender wants to be signed.

4. It is assumed that rational functions $G_2, \ldots G_k$: $R^t \to R$ are known to the public.

The above scheme works as follows:

Public key The sender's public key for the signature verification is given by the function $v_2, \ldots v_k$

Construction of the signed program The sender chooses a random rational function $r:R^l \to R$. The sender then constructs the map f_{signed}: $R^l \to R^k$ with components given by $f_{signed, I} := s_i(r, G_2 \circ f, \ldots, G_k \circ f)$, $1 < = I < = k$. Then it computes the representation of f_{signed} and sends the tuple of functions (f, f_{signed}) to the receiver.

Execution of the signed program The receiver then computes $y:=f(x)$ and $z:= f_{signed}(x)$. (y,z) will then be a signed output of the program.

Signature Verification $G_i(y)$, $2 < = I < = k$ and $v_i(z)$, $2 < = I < = k$ is computed. Z is a signature of y if and only if $v_i(z) = G_i(y)$ for $2 < = I < = k$.

In the above scheme, it is assumed that an adversary does not know the functions r and v1. Since the adversary does not know r, the left decomposition attack mentioned above to obtain s_i becomes harder.

And since it does not know v1, it cannot compute input/output pairs for the interpolation of the s_i.

Also, because the adversary does not know r, he cannot compute input/output pairs for the interpolation of the s_i as mentioned in the second interpolation attack above.

Finally, even if an adversary was able to invert f, the scheme is not broken, because he does not know r and so he does not know how to compute preimages of $(r, G_2 \circ f, \ldots, G_k \circ f):R^l \to R^k$.

9.5 Evaluating Code Mess-Up and Timer Algorithms

The strength of a code mess-up algorithm relies on the technique used in order to obfuscate the code. That is why evaluating such algorithms and proving their security is a relative process. Exactly like proving the strength of an encryption, it can be proven to be secure and after a while it can be attacked and hence become insecure.

In order to assess the strength of the code mess-up approach used in this thesis, we are going to adopt the example of Alice and Bob, two parties who want to communicate securely. As mentioned above, "Alice (the originating host) has an algorithm to compute a function f. Bob (the remote host) has an input x and is willing to compute f(x) for her, but Alice wants Bob to learn nothing substantial about f. Moreover, Bob should not need to interact with Alice during the computation of f(x). "

The protocol for the above description is as follows:

1. Alice starts by inserting dummy code in f's code.
2. Alice then performs two transformation proceses:
   - Replacing numerical values in the algorithm by other non-true values
   - Replacing string values in the algorithm by other obfuscated strings
3. Alice Sends the algorithm to Bob.

Now, let's assume that it takes x = 1000 milliseconds to execute f with no code mess-up.

And then, let's assume that inserting dummy code will increase the code length by 30%. Hence, it will take

(30/100) * 1000) /(100/100) = 300

milliseconds more time to execute the algorithm. -----------------------------(1)

Now let's assume that it takes 200 milliseconds to unmess-up an obfuscated string value, and 120 milliseconds to unmess – up a numeric one.

Thus, unmessing-up an algorithm with 4 string values and 5 numerical values will need: **((4*200) + (5 * 120)) = 1400 milliseconds.**

Hence, an algorithm f with 4 string values and 5 numerical values will need **300 + 1400 = 1700** extra milliseconds of a malicious host's time in order to be hacked.

This might not be a big deal if the malicious host has enough time to do that. But **IF** Alice had a timer that counts the estimated time for the function f to be executed, then she can easily know if it took a longer time to execute f than it should be or not. Talking in mobile agents language, a host can easily keep track of the time that its agent takes to be executed on other hosts, if this time is exceeded, the host can take actions to kill the agent and hence protects it from malicious attacks.

9.6 Summary

In order to increase our confidence in the security of our system we tried to show in this some verification methods for our proposed solution. We applied verification resources to encryption, code mess up, time out mechanism, read only and append only lists. Some of these verification methods was theoretical, like what we did with encryption and time out, while others are verified by experimentation like code mess up, denial of execution and the read only and append only mechanisms. These experiments are presented clearly in chapter 10.

Chapter 10 – Results and Analysis

10.1 Introduction

The purpose of testing the proposed design is to analyze and prove the strength of its proposed solution in this thesis. Six experiments are presented in this chapter to test the different parts of the proposed design. In brief, these experiments are designed for:

1. Showing the changes occurring in the agent's structure after applying the code mess-up algorithm.
2. Attempting to re-generate the messed up code from its generated byte code
3. Applying encryption to parts of the agent's code.
4. Showing the effect of timing an agent.
5. Simulating the denial of execution attack to show the behavior of a malicious host who denies execution of the agent.
6. Comparing two versions of the agent launched by the Concordia mobile agent system. One version with applied security and another with no security applied at all. Time was measured to see the difference in launching time between the two agents and to show the security overhead.

The implementation language used is Java and the mobile agent system used to launch the agent is the Concordia mobile agent system.

10.2 Test Case Description

The experimental agent created is a flower-buying agent. The owner aims to buy a bouquet of flowers and it only allocates a specific budget for that purpose. The role of the agent is to visit a list of servers that is pre-assigned by the owner server (agents' itinerary), and return the minimum price found to the owner server, which has the option to either approve or deny the purchase.

Experiment 1 - Code Mess-up

1. Objective - Showing the effect of applying code mess up on the agent's structure and code.

2. Set Up - Before the agent is launched to the list of to-be-visited servers, our code mess-up algorithm is applied to the agent's major class so that its features are altered.

The code mess-up algorithm involves the following steps:

a. Adding a dummy class to the agents' main class

b. Changing numeric values in the agents' code using a specific algorithm that is reversed in the server side to re-obtain the original values, below is a sample of code mess-up for integers:

```
public int ChangeInt(String Token, Server ServerInstance){
  //check if the token is an integer or not

    Random RandomSeed = new Random();
    int RandomValue = RandomSeed.nextInt();
    RandomValue = RandomValue / 10000000;
    setnewInt(RandomValue); //this is the random value to be
sent over the network,
  //it should be recovered in the server!!!
    int NewValue;
      //remove the ; from the end of the token
    int Length = Token.length();
    char semiColon = Token.charAt(Length-1);
    if (semiColon == ';')
      Token = Token.substring(0, Length-1);

    Integer temp = new Integer(Token);
    NewValue = temp.intValue() * RandomValue;
    ServerInstance.setRandomInt(RandomValue);
  return NewValue;
}
```

Fig. 10.1 – Altering Integer Values

c. Changing string values in the agents' code using a specific algorithm that is reversed in the server side to re-obtain the original strings. Below is a sample of a code mess-up function for strings:

```
public String ChangeString (String Token, Server
ServerInstance) {
  String NewValue = new String();
  int container;
  Random RandomSeed = new Random();
  int RandomValue = 0;
  //remove the ; from the end of the token
  int Length = Token.length();
  char semiColon = Token.charAt(Length-1);
  if (semiColon == ';')
      Token = Token.substring(0, Length-1);
  for (int counter = 1; counter < Token.length()-1; counter
++){
      RandomValue = RandomSeed.nextInt();
      RandomValue = RandomValue / 100000000;
      container = Token.indexOf(counter) * RandomValue;
      if (container < 0)
        container = container * -1;
      String t = Integer.toString(container);
       NewValue = NewValue.concat(t);
  }
  setnewString(NewValue);
  ServerInstance.setRandomString(RandomValue);
  return NewValue;
}
```

Fig. 10.2 – Altering String Values

3. Results and analysis

As shown below, the result of the above mechanisms is a messed-up mobile agent code. This is an example of an agent file before and after applying the parsing and code mess-up processes. Parsing is part of the code mess up, it is done to read the agent file and detect when there is a variable and what is its type in order to apply on it the appropriate mess up function.

```
Package examples.testagent;
import java.net.*;
import java.io.*;
import java.util.*;
import java.lang.Object;
import COM.meitca.concordia.*;
import java.io.FilePermission;
import java.security.*;

public class myagent_2 extends Agent{
  double OwnerBestPrice = 20.0 ; // value before being parsed
  Vector AddressList = new Vector();
  private double DivideBy = 3; // value before being parsed
  private String Product = "RedFlowers";// value before being
parsed
  private int Denied = 0;
public void SayHello_Owner() {
    try {
      BufferedWriter OwnerResults = new BufferedWriter(new
FileWriter("c:\\Program Files\\Mitsubishi
Electric\\Concordia\\Agents\\examples\\testagent\\OwnerResults
.java"));
      GetItinerary GItinerary = new GetItinerary();

  GItinerary.setServerFile("c:\\windows\\desktop\\serverlist.tx
t");
      GItinerary.setDivideBy(3);
      OwnerBestPrice =
GItinerary.getOwnerServerInfo(AddressList, OwnerResults);
      parser ParserInst = new parser();
      Server ServerInstance = new Server();
      ParserInst.ParseFile("c:\\Program Files\\Mitsubishi
Electric\\Concordia\\Agents\\examples\\testagent\\myagent_3.ja
va", ServerInstance, OwnerResults);
      System.out.println("Owner Results after parsing");
      OwnerResults.write("Owner Results after parsing");
      OwnerResults.newLine();
      OwnerBestPrice =
GItinerary.getOwnerServerInfo(AddressList, OwnerResults);
      ReadExe ReadInst = new ReadExe();

      System.out.println("Bye Owner");
      OwnerResults.close();
    } catch (Exception error) {
      System.out.println(error.getMessage());
  }
}

public void SayHello_Visited() {
  try {
    BufferedWriter VisitedResults = new BufferedWriter(new
FileWriter("c:\\Program Files\\Mitsubishi
Electric\\Concordia\\Agents\\examples\\testagent\\VisitedResul
ts.java"));
    GetVServerInfo GInfo = new GetVServerInfo();
    VisitedServer sInstance = new VisitedServer();
    shoplistvector ShopListInst = new shoplistvector();
    String Address = new String();
    for (int counter = 0;counter < 2;counter ++){
      Address = (String) AddressList.elementAt(counter);
      Vector VInstance = GInfo.getVisitedServerInfo(Address,
sInstance, OwnerBestPrice, ShopListInst, VisitedResults);
      System.out.println("Finished getting visited server
info");
      ServerEvaluatePrice EvaluateInst = new
ServerEvaluatePrice();
```

```
      EvaluateInst.evaluateprice(VInstance, VisitedResults) ;
      System.out.println("After evaluate price");
      securityparsing secInst = new securityparsing();
      secInst.ServermainCalls(VInstance, VisitedResults);
      System.out.println("Bye Visited");
      VisitedResults.close();
    }
  } catch (Exception error) {
    System.out.println(error.getMessage());
  }
 }
}
```

Fig. 10.3 – Agent's code BeforeCode Mess-up

```
Package examples.testagent;
import java.net.*;
import java.io.*;
import java.util.*;
import java.lang.Object;
import COM.meitca.concordia.*;
import java.io.FilePermission;
import java.security.*;

class TheParser{ //Dummy Class Inserted
  private double newdoubleVar = 0.0 ;
// value after being parsed
  private String newStringVar = "121710415";
// value after being parsed
  private String strVar = "121710415";
  private StringTokenizer TheActualString;
}
public class myagent_2 extends Agent{
  double OwnerBestPrice = 5.907872033473425E-7 ;
// value after being parse
  Vector AddressList = new Vector();
  private double DivideBy = 8.802684527589691E-8 ;
// value after being parse
  private String Product = "121181615151421613";
// value after being parse
  private int Denied = 0;

public void SayHello_Owner() {
try {
   BufferedWriter OwnerResults = new BufferedWriter(new
   FileWriter("c:\\Program Files\\Mitsubishi
   Electric\\Concordia\\Agents\\examples\\testagent\\OwnerResul
   ts.java"));
  GItinerary GItinerary = new GItinerary();
  GItinerary.setServerFile("c:\\windows\\desktop\\serverlist.tx
   t");
  GItinerary.setDivideBy(3);
  OwnerBestPrice = GItinerary.getOwnerServerInfo(AddressList,
OwnerResults);
parser ParserInst = new parser();
Server ServerInstance = new Server();
ParserInst.ParseFile("c:\\Program Files\\Mitsubishi
Electric\\Concordia\\Agents\\examples\\testagent\\myagent_3.ja
va", ServerInstance, OwnerResults);
System.out.println("Owner Results after parsing");
OwnerResults.write("Owner Results after parsing");
OwnerResults.newLine();
OwnerBestPrice = GItinerary.getOwnerServerInfo(AddressList,
OwnerResults);
ReadExe ReadInst = new ReadExe();
System.out.println("Bye Owner");
OwnerResults.close();
} catch (Exception error) {
  System.out.println(error.getMessage());
   }
  }
public void SayHello_Visited() {
try {
  BufferedWriter VisitedResults = new BufferedWriter(new
  FileWriter("c:\\Program Files\\Mitsubishi
  Electric\\Concordia\\Agents\\examples\\testagent\\VisitedRes
  ults.java"));
  GetVServerInfo GInfo = new GetVServerInfo();
  VisitedServer sInstance = new VisitedServer();
  shoplistvector ShopListInst = new shoplistvector();
  String Address = new String();
  for (int counter = 0;counter < 2;counter ++){
    Address = (String) AddressList.elementAt(counter);
    Vector VInstance = GInfo.getVisitedServerInfo(Address,
     sInstance, OwnerBestPrice, ShopListInst, VisitedResults);
    System.out.println("Finished getting visited server info");
     ServerEvaluatePrice EvaluateInst = new
     ServerEvaluatePrice();
     EvaluateInst.evaluateprice(VInstance, VisitedResults) ;
     System.out.println("After evaluate price");
     securityparsing secInst = new securityparsing();
     secInst.ServermainCalls(VInstance, VisitedResults);
     System.out.println("Bye Visited");
     VisitedResults.close();
    }
    } catch (Exception error) {
System.out.println(error.getMessage());
   }
  } }
```

Fig. 10.4 Agent's code after Code Mess-up

Experiment 2 - Re-generating the messed up code

1. Objective - Attempting to re-generate the messed up code from its generated byte code

2. Set up – Using a Java Decompiler called JAD, we attempted to simulate the role of a malicious attacker to re-generate the agent's source code from its class file which roams the Internet from one host to another. The JAD software takes a ".class" file and generates its ".java" source.

3. Results and analysis – as shown in figure 10.5, the regenerated ".class" contains the agent's code after being messed-up, and thus, the attacker will not be able to understand it directly. The attacker must spend a longer time to un-mess up the code and re-generate the original values carried by the agent.

```
// Decompiled by Jad v1.5.7f. Copyright 2000 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/SiliconValley/Bridge/8617/jad.html
// Decompiler options: packimports(3)
// Source File Name:   myagent_2.java

package examples.testagent;

import COM.meitca.concordia.Agent;
import java.io.*;
import java.net.InetAddress;
import java.util.Vector;

// Referenced classes of package examples.testagent:
//            GetItinerary, parser, Server, ReadExe,
//            GetVServerInfo, VisitedServer, shoplistvector, ServerEvaluatePrice,
//            securityparsing

public class myagent_2 extends Agent
{

    public myagent_2()
    {
        OwnerBestPrice = 1.2269808678393334E-006D;
        AddressList = new Vector();
        DivideBy = 1.8345589494969552E-007D;
        Product = "16016911181411317";
        OwnerIP = new String();
        VisitedIP = new String();
        Denied = 0;
    }

    public void SetDenialOfExecution(int i)
    {
        Denied = i;
    }

    public void SayHello_Owner()
    {
      try
      {
         BufferedWriter bufferedwriter = new BufferedWriter(new FileWriter("c:\\Program Files\\Mitsubishi Electric\\Concordia\\Agents\\examples\\testagent\\OwnerResults.java"));
         InetAddress inetaddress = InetAddress.getLocalHost();
         byte abyte0[] = inetaddress.getAddress();
         String s = new String();
         String s1 = new String();
         for(int i = 0; i < abyte0.length; i++)
         {
            int j = abyte0[i] >= 0 ? ((int) (abyte0[i])) : abyte0[i] + 256;
            s1 = s1 + String.valueOf(j);
         }

         System.out.println("Owner Server's IP = " + s1);
         bufferedwriter.write("Owner Server's IP = " + s1);
         bufferedwriter.newLine();
         OwnerIP = s1;
         GetItinerary getitinerary = new GetItinerary();
         getitinerary.setServerFile("c:\\windows\\desktop\\serverlist.txt");
         getitinerary.setDivideBy(3D);
         OwnerBestPrice = getitinerary.getOwnerServerInfo(AddressList, bufferedwriter);
         parser parser1 = new parser();
         Server server = new Server();
         parser1.ParseFile("c:\\Program Files\\Mitsubishi Electric\\Concordia\\Agents\\examples\\testagent\\myagent_2.java", server, bufferedwriter);
         System.out.println("Owner Results after parsing");
         bufferedwriter.write("Owner Results after parsing");
         bufferedwriter.newLine();
         OwnerBestPrice = getitinerary.getOwnerServerInfo(AddressList, bufferedwriter);
         ReadExe readexe = new ReadExe();
         System.out.println("Bye Owner.");
         bufferedwriter.write("Bye Owner.");
         bufferedwriter.newLine();
         bufferedwriter.close();
      }
      catch(Exception exception)
      {
         System.out.println(exception.getMessage());
      }
   }

   public void SayHello_Visited()
   {
      try
      {
         BufferedWriter bufferedwriter = new BufferedWriter(new FileWriter("c:\\Program Files\\Mitsubishi Electric\\Concordia\\Agents\\examples\\testagent\\VisitedResults.java"));
         if(OwnerIP.equals("20816013196"))
         {
            System.out.println("EXECUTION DENIED");
            bufferedwriter.write("EXECUTION DENIED");
            bufferedwriter.newLine();
            SetDenialOfExecution(1);
            InetAddress inetaddress = InetAddress.getLocalHost();
            byte abyte0[] = inetaddress.getAddress();
            String s = new String();
            String s1 = new String();
            for(int i = 0; i < abyte0.length; i++)
            {
                int k = abyte0[i] >= 0 ? ((int) (abyte0[i])) : abyte0[i] + 256;
                s1 = s1 + String.valueOf(k);
            }

            System.out.println("Visited Server IP = " + s1);
            bufferedwriter.write("Visited Server IP = " + s1);
            bufferedwriter.newLine();
            VisitedIP = s1;
         } else
         {
            GetVServerInfo getvserverinfo = new GetVServerInfo();
            VisitedServer visitedserver = new VisitedServer();
            shoplistvector shoplistvector1 = new shoplistvector();
            String s2 = new String();
            for(int j = 0; j < 2; j++)
            {
                String s3 = (String)AddressList.elementAt(j);
                Vector vector = getvserverinfo.getVisitedServerInfo(s3, visitedserver, OwnerBestPrice,
shoplistvector1, bufferedwriter);
                System.out.println("Finished getting visited server info");
                bufferedwriter.write("Finished getting visited server info");
                bufferedwriter.newLine();
                ServerEvaluatePrice serverevaluateprice = new ServerEvaluatePrice();
                serverevaluateprice.evaluateprice(vector, bufferedwriter);
                System.out.println("After evaluate price");
                bufferedwriter.write("After evaluate price");
                bufferedwriter.newLine();
                securityparsing securityparsing1 = new securityparsing();
                securityparsing1.ServermainCalls(vector, bufferedwriter);
                System.out.println("Bye Visited Server");
                bufferedwriter.write("Bye Visited Server");
                bufferedwriter.newLine();
            }

            bufferedwriter.close();
         }
      }
      catch(Exception exception)
      {
         System.out.println(exception.getMessage());
      }
   }

   double OwnerBestPrice;
   Vector AddressList;
   private double DivideBy;
   private String Product;
   String OwnerIP;
   String VisitedIP;
   int Denied;
}
```

Fig. 10.5 Generated .JAVA file

Experiment 3 - Encryption

1. **Objective –** Applying encryption to the agent's code, specifically to the list carried by the agent.

2. **Set Up -** After applying the code mess-up algorithm and parsing, public key cryptography is used to encrypt the agents' code. Actually, for testing purposes, encryption was applied by the owner server to the agents' list that is supposed to carry the visited servers' prices.

3. **Results and Analysis**

Figure 10.6 below shows the Owner Results after applying the parsing (or code mess up). Notice how the owners' best price was altered. Figures 10.6 and 10.7 show the results after encrypting the agent's list.

```
Hello Owner
Owner Best Price = 20.0
Owner Best Price After Parsing = 6.666666666666667
The NewFileName = myagent_2
Finished Parsing
Bye Owner
```

Fig. 10.6 – Owner Results after code mess up

```
Getting the visited server info
    this is the server price 20 //Getting the visited server Offered Price
end of getserverprice
Finished getting visited server info
The length of the shoplist is: 2 //The shop list carries the data that the agent collects
from the visited servers.
Elements in the List (server URL and price)=

cs11/208.160.131.96/Agent1a0d74:cfe2895e92:-7f74

After evaluate price

Applying security to contents of the list

//Key Pair Generation using Sun DSA

Generated Key pair = java.security.KeyPair@1a9405
```

```
private key = Sun DSA Private Key
parameters: //Parameters of the private  key
p:
fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6
df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17
q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5
g:
678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b71fd7
3da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4
 x: 8191cd235b4ad5c8e5a26056633eb7c67c5bbb41
signature = [B@1a9086
public key = Sun DSA Public Key
parameters: //Parameters of the public key
p:
fca682ce8e12caba26efccf7110e526db078b05edecbcd1eb4a208f3ae1617ae01f35b91a47e6
df63413c5e12ed0899bcd132acd50d99151bdc43ee737592e17
q: 962eddcc369cba8ebb260ee6b6a126d9346e38c5
g: 678471b27a9cf44ee91a49c5147db1a9aaf244f05a434d6486931d2d14271b9e35030b7
1fd73da179069b32e2935630e1c2062354d0da20a6c416e50be794ca4
y: 3ed7f9b803a5a756e6dac49b3df115921e5dfa2493e79f41babcfcbc66d0d5dfd46fb3e
57d6d45803eaea245d34147140dc5bdbf75b92648c87d45315036ba5
signature verifies: true //Verifying the generated signature
finished security
Secured List data = [B@1a9013 //Agents's List after encryption
Bye Visited Server
Getting the visited server info
```

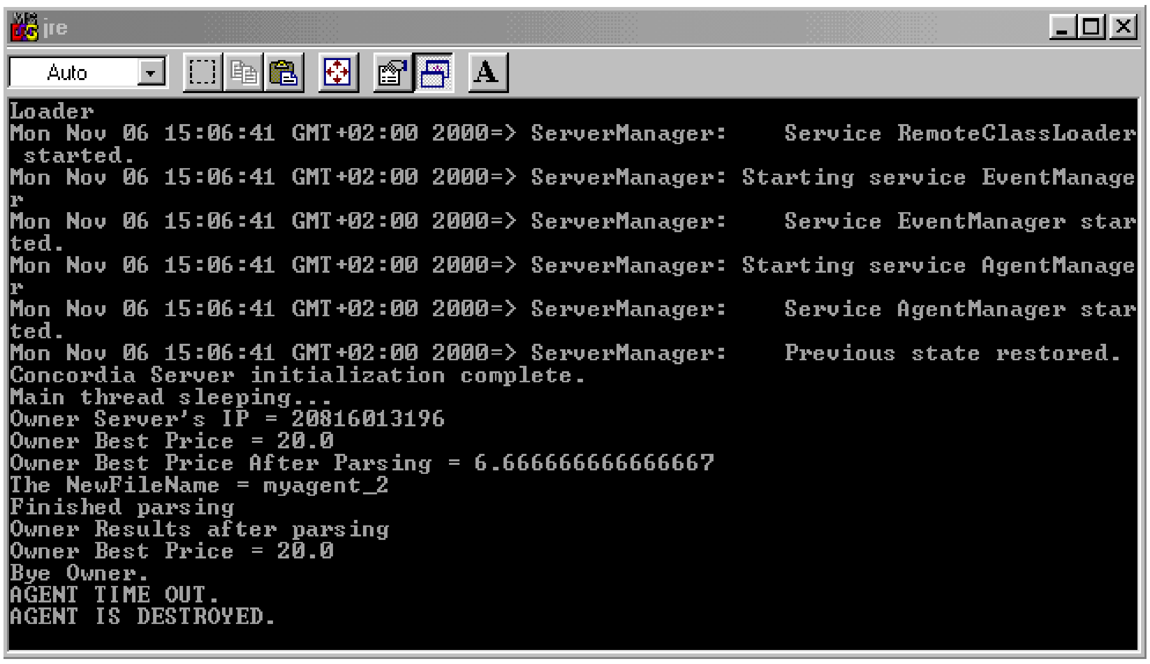
Fig. 10.7 - Visited Server results with encryption

Experiment 4 - Showing the Effect of timing an agent.

1. Objective – Showing that after a specific pre-set time interval, the agent launched will be destroyed if not returned before the pre-set time to its launching server.

2. Set up – Setting a thread to sleep for 60 seconds. When the agent does not return to its home server within the 60 seconds, it will be destroyed by the home server by killing the thread running the agent.

3. Results and analysis – The figure below shows the response of the home server to a delayed agent.



```
Loader
Mon Nov 06 15:06:41 GMT+02:00 2000=> ServerManager:    Service RemoteClassLoader
 started.
Mon Nov 06 15:06:41 GMT+02:00 2000=> ServerManager: Starting service EventManage
r
Mon Nov 06 15:06:41 GMT+02:00 2000=> ServerManager:    Service EventManager star
ted.
Mon Nov 06 15:06:41 GMT+02:00 2000=> ServerManager: Starting service AgentManage
r
Mon Nov 06 15:06:41 GMT+02:00 2000=> ServerManager:    Service AgentManager star
ted.
Mon Nov 06 15:06:41 GMT+02:00 2000=> ServerManager:    Previous state restored.
Concordia Server initialization complete.
Main thread sleeping...
Owner Server's IP = 20816013196
Owner Best Price = 20.0
Owner Best Price After Parsing = 6.666666666666667
The NewFileName = myagent_2
Finished parsing
Owner Results after parsing
Owner Best Price = 20.0
Bye Owner.
AGENT TIME OUT.
AGENT IS DESTROYED.
```

Fig. 10.8 – The owner server's behavior to a timed out agent

Experiment 5 - Simulating the denial of execution attack

1. **Objective -** To show the behavior of a malicious host who denies execution of the agent.
2. **Set Up -** In this experiment, simulation of denial of execution attack by a specific visited server is done. The visited server starts by checking the Agent’s home server IP, if it matches a “to-be-Denied” IP, the visited server denies execution of the agent and the agent reports this attempt to its owner server that destroys the agent. Reporting to the home server is done by setting an Execution Status flag in the agent. If this flag is set, the homer server will know that its agent has been denied execution.
3. **Results and analysis -** Figure 10.9 shows the results appearing in the owner server. The owner server IP is displayed and the results are displayed normally until a denial of service attack is detected, the owner server then displays a message indicating which server IP denied executing the agent then it kills the agents. Figure 10.10 shows the behavior of the malicious visited server that denies the execution of the visiting agent.

```
Owner Server's IP = 208.160.131.96

Owner Best Price = 20.0

Owner Best Price After Parsing = 6.666666666666667

Owner Results after parsing

Owner Best Price = 20.0

Bye Owner.

//This appears after the visited server denies execution

The Server with IP = 208.160.131.97 Denied Executing my Agent!

Agent Is Destroyed
```

Fig. 10.9 - Owner server’s results

```
EXECUTION DENIED

Visited Server IP = 208.160.131.97
```

Fig. 10.10 – Visited server's results

Experiment 6 - Comparing between agents' performance speed with and without applying security.

1. Objective – Showing the effect of applying the Three-Tier Security mechanism on the time taken by the agent to reach back to the owner server, using the Concordia launch wizard.

2. Set up – The time needed by the agent to move from the home host to another one and return back to the home host is measured with and without applying the three-tier security mechanism to the agent.

3. Results and analysis – The time taken by the agent with security = 1 minute, 37 seconds

The time taken by the agent without security = 5 seconds

These results show the overhead imposed by the Three-Tier protection mechanism on time. Time difference between the start and end times of the agent's journey was calculated using the system clock. The figure below shows the results generated by the owner server and visited server without applying any security mechanisms.

```
Hello Owner
Owner Best Price = 20.0 //Price as it is with no parsing
Bye Owner
```

Fig. 10.11 Owner Results with no parsing

```
Hello Visited
Getting the visited server info
this is the visited server price 20
end of getserverprice 20
This is the visited server's best price: 20.0
this is the server price 20
end of getserverprice 20
This is the visited server's best price: 20.0
shoplistinstprice 20.0
shoplistinstAddress 208.160.131.122
shoplistinstprice 20.0
```

```
shoplistinstAddress 208.160.131.122
Finished getting visited server info
the length of the shoplist is: 2
x= www/208.160.131.124/Agent1a0fad:e18fc4c75c:-7fd2
this is the server price 20
end of getserverprice 20
temp= 20.0
After evaluate price
Bye Visited
Getting the visited server info
this is the server price 20
end of getserverprice 20
This is the visited server's best price: 20.0
this is the server price 20
end of getserverprice 20
This is the visited server's best price: 20.0
shoplistinstprice 20.0
shoplistinstAddress 208.160.131.122
shoplistinstprice 20.0
shoplistinstAddress 208.160.131.122
Finished getting visited server info
the length of the shoplist is: 4
x= www/208.160.131.124/Agent1a0fad:e18fc4c75c:-7fd2
this is the server price 20
end of getserverprice 20
temp= 20.0
After evaluate price
Bye Visited
```

Fig. 10.12 Visited Server Results with no security applied

10.3 Summary

In this chapter we presented six experiments to test the proposed design. These experiments covered the areas of code mess up, time out, encryption, denial of execution, launching time, read only and append only lists as well as the JAD attack. The results showed that combining the code messup technique with encryption and timing gives the agent the privilege of securely moving from one host to another within a pre-set time without the fear of malicious attacks from other hosts. But encryption time is one drawback of the above technique.

Chapter 11 – Conclusion and Future Work

11.1 Summary of the Thesis Contributions

The objective of this thesis is to find a complete, reliable and secure method to protect mobile agents from possible malicious host attacks. A three-tier solution was presented in the thesis. This solution is based on a combination of code mess up, encryption and time out. The thesis developed a flower-buying mobile agent and introduced a new code mess up algorithm that is based on inserting dummy code in the agent's code and obfuscating numerical and string values in the agent's code. The encryption algorithm used to protect the agent's list is the DES algorithm, and the time out operation was implemented using multi-threading. Those three methods complement each other for providing a comprehensive solution to the problem of protecting mobile agents against possible host attacks. For instance, if the attacker managed to decrypt the agent's code, it will be faced with a messed up code. In order to understand this code, the attacker will require an additional time. Here comes the role of the timer. If the agent did not return to its home host in time, the host will destroy it and the attacker will only have the time to alter a useless agent's code.

11.2 Problems Faced

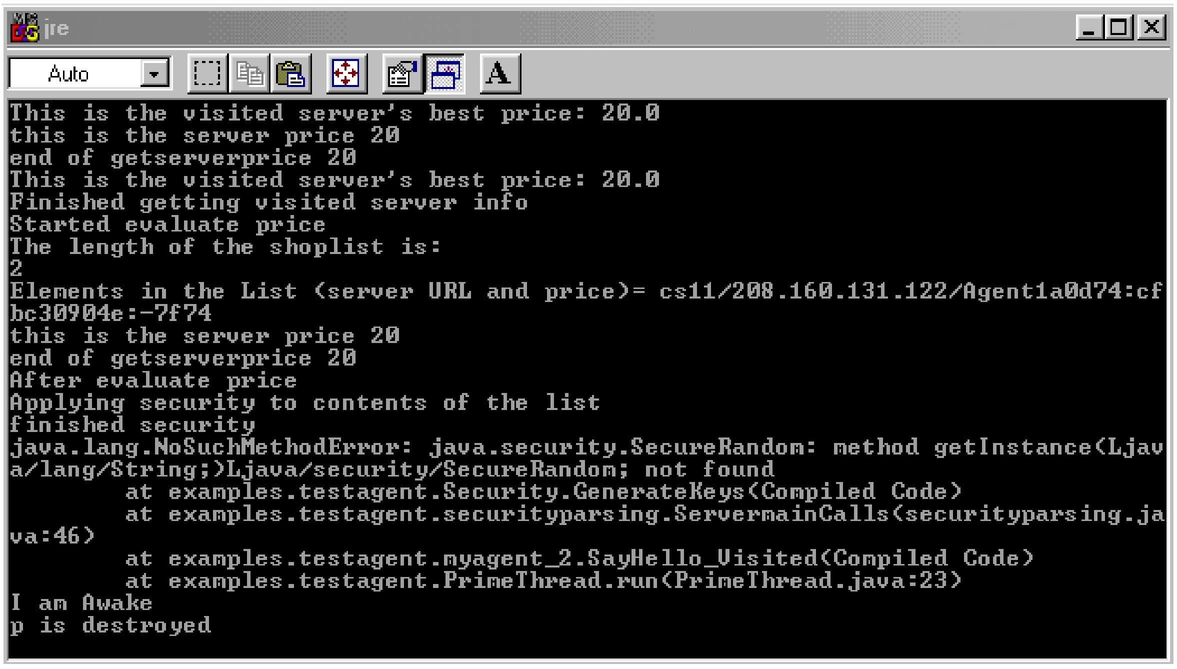
Actually there were several problems faced during the development of the proposed design. This is due to the factor that the issue of protecting an agent against possible host attacks is still a new subject and remains an open question with no comprehensive solution until this time **[Gra98]**.

In addition to that, major problems were faced during the implementation, mainly with the Concordia mobile agent system. Concordia managed to successfully launch the agent and performed most of the functions required by the agent code. But there were 2 major problems. These problems were:

11.2.1 Unsupported swing classes

Concordia did not accept to launch the agent when it called any functions from the JDK 1.3 swing classes. These functions were required in several parts of the code to implement the timer and to use some of the security classes.

Figures 11.1 shows a sample snapshot of an error message generated from the Concordia wizard showing that Concordia did not accept the method GetInstance() from class java.security.SecureRandom. There is a probability that this refusal might be attributed to the fact that we are using the shareware version of Concordia, not the commercial one.



```
This is the visited server's best price: 20.0
this is the server price 20
end of getserverprice 20
This is the visited server's best price: 20.0
Finished getting visited server info
Started evaluate price
The length of the shoplist is:
2
Elements in the List (server URL and price)= cs11/208.160.131.122/Agent1a0d74:cf
bc30904e:-7f74
this is the server price 20
end of getserverprice 20
After evaluate price
Applying security to contents of the list
finished security
java.lang.NoSuchMethodError: java.security.SecureRandom: method getInstance(Ljav
a/lang/String;)Ljava/security/SecureRandom; not found
        at examples.testagent.Security.GenerateKeys(Compiled Code)
        at examples.testagent.securityparsing.ServermainCalls(securityparsing.ja
va:46)
        at examples.testagent.myagent_2.SayHello_Visited(Compiled Code)
        at examples.testagent.PrimeThread.run(PrimeThread.java:23)
I am Awake
p is destroyed
```

Fig. 11.1 – Error Generated from Concordia Wizard

11.2.2 Unsupported Timer Class

Concordia did not accept to launch the agent when it called any functions from the JDK 1.3 timer classes. These functions were required to implement the timer object. In order to solve this problem, multi-threading was used in the implementation to simulate the timer behavior and get around this problem.

11.3 Future Work

Future enhancements for this thesis can be done in the parser, code mess up and encryption modules. A more powerful parser can replace the existing one in order to cover all possible variable types that might exist in the agent's code. In addition to that, the code mess up algorithm can be enhanced and replaced with a more complicated one. And the same is for the encryption algorithm used, more powerful encryption algorithm can be used instead of the DES. Furthermore, work can be done to reduce the time overhead imposed by the agent when applying the three-tier security mechanism.

It is worth mentioning here that Concordia was used as a means for launching our agent. This does not mean that for future enhancements, Concordia must be used. Rather, other mobile agent systems can be used to launch the agent.

The advantage of the implemented code structure in the implementation of this thesis is that it is easy to replace any of the modules since they are clearly defined and structured modules. For instance, we have the security module, the parsing module and the code mess up module and several others that can be easily traced and replaced by other more powerful ones.

A complementary solution that can strengthen the existing proposed one can involve keeping the parts with sensitive data, like the parts containing the target price of a product to be searched, secret in the server. For instance, taking an airfare agent example. The agent starts its itinerary, aiming to gather all the possible offered prices and append them into a list. This list will be an append only list like the one used by the Ajanta mobile agent system **[Kar99]**. This means that the agent will append the data it collects from the different servers it visits, to its append only list. Data in this list will be marked read only. That is no other server can play with or tamper the data. Entries in this list cannot be deleted or modified by any other malicious part. In order to prevent other servers from reading the data in the append-only list, one can encrypt this data with the agent's public key. Each piece of data appended to the list should include the ID of the server it was obtained from so that the home server can easily know which data belongs to which server. In addition to that, each entry will be digitally signed by the owner's servers' key.

After the agent finishes, it creates a child agent, as shown in figure 6.3, and sends it with the list of possible prices to the home server. The server takes the list, performs the comparison and decides which offer it will go for. It then sends the result, digitally signed or encrypted, with the child agent back to the agent that performs the purchase and kills the child agent, and then it returns back to the home server. The list sent by the agent to the server should be encrypted as well by the agent key, and server will decrypt it. And the result sent from the server to the agent will be encrypted by the server's key to prevent any eavesdropping in the way. But this will impose an extra overhead on the network since the child agent will be sent to server and back to the agent. Also, the agent will wait idle waiting for the child agent to return back. The figure below shows a diagram that presents this solution. Future work can implement and encapsulate this solution in the existing one.

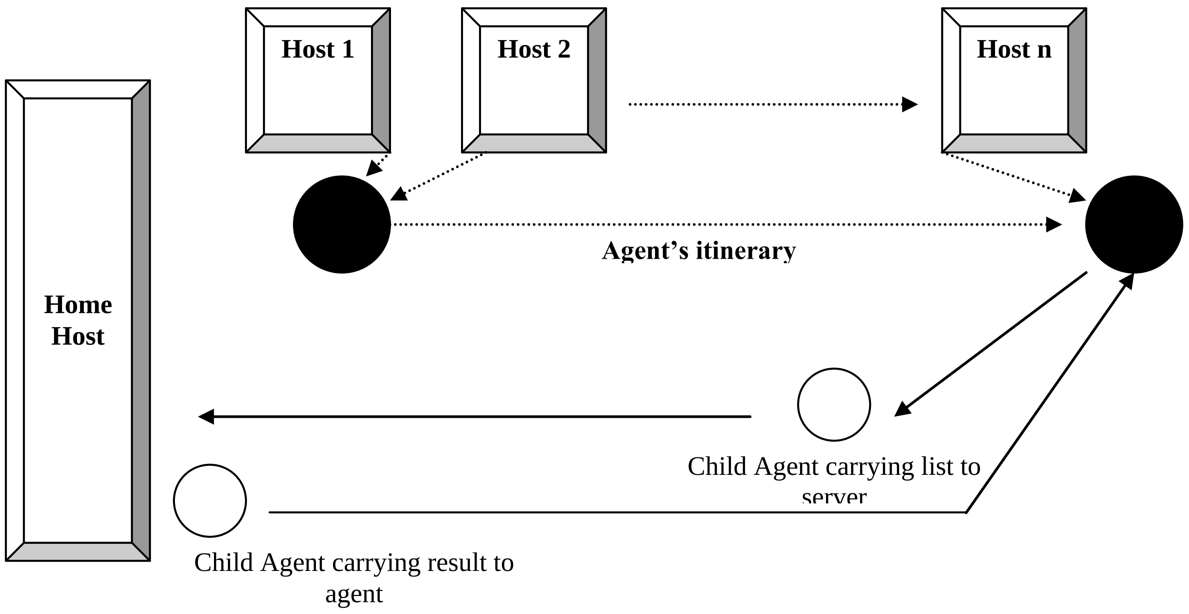

Host 1
Host 2
Host n
Agent's itinerary
Home Host
Child Agent carrying list to server
Child Agent carrying result to agent

Fig. 11.2 - Diagram of the alternative solution

Finally, enhancements can be done in the area of the home server’s response to a timed-out agent. For instance, instead of killing it, recharging the agent might take place in order for it to continue its journey safely, after making sure that it was not delayed because of malicious attacks.

Bibliography

[Ans95] Anselm et al. "*An Infrastructure for Mobile Agents: Requirements and Architecture.*" Frankfurt am Main, Germany.

[Bau97] Baumann J. et al. "Mole-Concepts of a mobile Agent System". Bericht 1997/15, August 1997.

[Con99] Concordia Documentation Roadmap, Copyright © 1999 Mitsubishi Electric Information Technology Center America. All rights reserved. Revised: November 02, 1999.

[Dal98] ElMansy, Dalia. "*Deciding About Agent Mobility Using A performance Cost Model*." Computer Science Department, American university in Cairo, 1998.

[Dav95] David et al. "*Mobile Agents: Are They a Good Idea?*" IBM Research Division, New York, 1995

[Edw94] Edward Amoroso et al. "*A process-Oriented Methodology for Assessing and Improving Software Trustworthiness*". CCS '94- 11/94 Fairfax Va., USA. 1994.

[Gra98] Gray S. Robert. "Agent Tcl: *A Flexible and secure mobile-agent system.*" January 1998.

[Hal99] Halevi, S. and Krawczyk, H. "Public-Key Cryptography and Password protocols." ACM transactions on information and system security, Vol 2, No. 3, Aug. 1999.

[Har97] Harold, Elliotte Rusty. "JAVA Network Programming." O'Reilly & Associates, 1997.

[Hoh98] Hohl, Fritz. "*Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts.*" Institute of Parallel and Distributed High-Performance Systems (IPVR), University of Stuttgart, Germany, 1998.

[Hoh97] Hohl, Fritz. "*An Approach to Solve the Problem of Malicious Hosts in Mobile Agent Systems". Institute of Parallel and Distributed High-Performance Systems (IPVR), University of Stuttgart, Germany, Mar. 1997.

[HohF98] Hohl, Fritz, *"A Model of Attacks of Malicious Hosts Against mobile Agents".* Institute of Parallel and Distributed High-Performance Systems (IPVR), University of Stuttgart, Germany, 1998.

[JAZ00] Jazayeri, M. and Lugmayr, W. "Gypsy: A Component-Oriented Mobile Agent System."8th Euromicro Workshop on Parallel and Distributed Processing, January 2000.

[Kar99] Karnik, N. and Tripath, A. "Security in the Ajanta Mobile Agent System". Department of computer science, university of Minnesota, Minneapolis. May 17, 1999.

[Lan99] Lange, D. and Oshima, M. "Seven Good Reasons for Mobile Agents". Communications of the ACM. Vol. 42, No.3, Mar. 1999.

[Mat99] An Introduction to Java Remote Method Invocation (RMI), (http://www.edm2.com/0601/rmi1.html)

[Mer91] R.C. Merkle, "Fast Software Encryption Functions," Advances in Cryptology-CRYPTO '90 Proceedings, Springer-Verlag, 1991, pp. 476-501.

[Neu98] Neuenhofen, K. and Thompson, M. " A Secure Marketplace for Mobile Java Agents." ACM transactions on autonomous agents. 1998.

[San97] Sander, T. and Tschudin, C. "*Protecting Mobile Agents Against Malicious Hosts."* International Computer Science Institute, Berkeley. 1997.

[Sat97] Sander, T and Tschudin, C. *"Towards Mobile Cryptography."* International Computer Science Institute. Nov. 22, 1997.

[Sch96] Schneier, Bruce. “*Applied Cryptogrphy”,* John Willey & sons, 1996.

[Sha] C.E. Shannon, “Communication Theory of Secrecy Systems, “*Bell System Technical Journal*, v.28, n. 4, 1949, pp.656-715.

[Tak97] Takanori et al, “*A Tentative Approach to Constructing Tamper-Resistant Software:,* New Security Paradigms Workshop Langdale, Cumbria UK. 1997.

[Tho97] Thorn, Tommy. *“Programming languages for Mobile Code”.* ACM Computing surveys. V29. Sep. 1997.

[Val96] Valente, L. and Tardo, J. “*Mobile Agent Security and Telescript.”.* General Magic, Inc. Feb. 1996

[Ven97] Venners, Bill. “Under the Hood: The architecture of aglets” Java World, 1997.

[Whi90] White, S.R. “Covert Distributed Processing with Computer Viruses”, Advances in Cryptology – CRYPTO ’89 Proceedings, Springer-Verlag, 1990, pp. 616-619.

[Wul98] Wulf, W. and Kienzle, M. *“A Practical Approach to Security Assessment”*, New Security Paradigms Workshop Langdale, Cumbria UK. 1998.

[Yee97] Yee, B. “*A Sanctuary for Mobile Agents”.* April 1997.