A Security Architecture for Mobile Agent Systems

by

PENG FU

B.Eng., Tsinghua University
China, 1999

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(The Department of Computer Science)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 2001

© Peng Fu, 2001
In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Oct. 11, 2001
Date

Department of Computer Science
Abstract

Unlike traditional mobile intelligent agent systems where a small set of APIs are provided to support limited agent (code) mobility capabilities, a novel agent system, called WAVE, offers a complete high-level language that is, despite its fairly simple syntax, rich in semantics and mechanisms for integration, control and management for rapid, effective realization of seamless, cooperative distributed applications. However like many other mobile agent systems, the lack of security in WAVE highly restricts its scope of applications. In this thesis, we propose a security architecture and implement a security system based on this architecture to secure the original WAVE system. This security system makes use of a rich security model that gives identification to each principal user and provides access control to a very fine level of granularity. The security system also provides methods for detecting whether the behavior or data of a wave agent has been tampered. Although the security architecture was developed for WAVE, its applicability can be generally suited to any mobile intelligent system.
Table of Contents

Abstract ......................................................................................................................... ii
Table of Contents ........................................................................................................ iii
List of Figures ................................................................................................................ vi
List of Tables ................................................................................................................ vii
Acknowledgements ..................................................................................................... viii
Dedication ..................................................................................................................... ix

Chapter 1 Introduction .................................................................................................. 1
  1.1 Mobile Agent Paradigm ..................................................................................... 1
  1.2 Benefits of Mobile Agent Paradigm ................................................................. 3
  1.3 Security Concerns ........................................................................................... 4
    1.3.1 Malicious Agent ...................................................................................... 5
    1.3.2 Malicious Host ...................................................................................... 6
    1.3.3 Unsecured Network Communication .................................................... 7
  1.4 Goals ................................................................................................................ 7
  1.5 Thesis Outline .................................................................................................. 8

Chapter 2 Background ................................................................................................ 9
  2.1 WAVE – a Novel Mobile Agent System ............................................................ 9
  2.2 Cryptographic Mechanisms ........................................................................... 12
    2.2.1 One-way Hash Function ....................................................................... 12
    2.2.2 Public-key Cryptography ..................................................................... 13
    2.2.3 Digital Signature ............................................................................... 14
  2.3 Related Works ................................................................................................ 15
    2.3.1 Aglet ................................................................................................. 15
    2.3.2 Concordia ........................................................................................ 16
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A Generic Mobile Agent System</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Structure of a Wave Host</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Knowledge Network of WAVE Paradigm</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>The Wave Secure System</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Naming Scheme of a WSS Domain</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>Structure of the Authentication Agent</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>An ASN.1 DER Encoding of Wave Certificate</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>WCI as a Branch of X.509 Directory Service</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>The Read Only Data</td>
<td>56</td>
</tr>
<tr>
<td>6.2</td>
<td>The Append Only Data</td>
<td>58</td>
</tr>
<tr>
<td>6.3</td>
<td>The Destination Specified Data</td>
<td>59</td>
</tr>
<tr>
<td>6.4</td>
<td>The Code Integrity Check</td>
<td>62</td>
</tr>
<tr>
<td>7.1</td>
<td>An Example of RMI Interface Lookup</td>
<td>67</td>
</tr>
<tr>
<td>7.2</td>
<td>Interface Between Wave Interpreter and the Authentication Agent</td>
<td>69</td>
</tr>
<tr>
<td>7.3</td>
<td>Structure of the Wavetella</td>
<td>73</td>
</tr>
<tr>
<td>A.1</td>
<td>Wave Grammar</td>
<td>85</td>
</tr>
</tbody>
</table>
List of Tables

2.1 A Comparison of the Security Solutions of Five Mobile Agent Systems.... 17
5.1 WSS File System Descriptions.................................................. 46
6.1 An Example of an Agent Registration Table..................................61
6.2 A Refined ART to Prevent Signature Replay..................................63
7.1 Server Configurations.................................................................70
7.2 Overhead Measurements.............................................................72
Acknowledgements

The author would like to thank Dr. Son Vuong for his supervision and guidance without which this work could not have been completed. The author would also like to thank Dr. Alan Wagner for his time and comments on this thesis work and his suggestions for improvement. The author would also like to thank Ms. Kirsty Barclay and my wife, Jessica who dedicating their time and effort in reviewing the writing.

Peng Fu

The Department of Computer Science
The University of British Columbia
October 2001
Chapter 1

Introduction

1.1 Mobile Agent Paradigm

Mobile intelligent agent technology has drawn a tremendous amount of attention from researchers in Internet computing recently as it promises to provide an elegant and efficient way of solving complex distributed problems, as well as offering a new approach to human-computer-interaction. In most mobile intelligent agent systems, the software agent travels autonomously within the agent enabled networks, executes itself in the agent execution environment, collects useful information and makes its own decision on behalf of its owner. Because of the rich context of the problem, many researchers have tried to give definitions of the mobile agent from different perspectives. For example, Pattie Maess defines agents by emphasizing on their autonomy [Maes 97]; Michael Coen gave a definition focusing on the negotiation and coordination of information transferring by agents [Coen 94]; Stan Frankline and Art Graesser's definition captures the essence of an agent, which is not, however, directly related to the software agent [FG 96]. Despite the rapid growth of research interest in the topic, there is no agreement yet among researchers on what an agent actually is [NW 97]. In our view, a mobile (software) agent is a program that performs specific tasks on behalf of its owner. It must have the following three properties:
• **Autonomy**
The agent must operate without the direct intervention of its owners and have some kind of control over its actions and internal states. At this point, the "autonomy" is to the agent owner's perspective: after the owner injects the agent, the agent can perform tasks autonomously. This autonomy can be coded into the agent in advance.

• **Mobility**
The agent must be able to perform a task over a distributed environment. Thus it can suspend its execution on one machine and dispatch itself to another machine and resume execution there. The migration path can be predetermined or decided on during the execution of the agent.

• **Reactivity**
The agent must be able to interact with its environment and other agents. Also, they should perform proper responses in a timely fashion.

From this definition we can see that a mobile agent is a rather complex piece of software. It is usually written in a script language that can be interpreted by the agent platform. The agent platform, mobile agent, together with their underlying software and hardware infrastructure, form the Mobile Agent Paradigm. Figure 1.1 shows a general architecture of a mobile agent system.

<table>
<thead>
<tr>
<th>Application</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile Agent</td>
<td>Mobile Agent</td>
</tr>
<tr>
<td>Agent Platform</td>
<td>Agent Platform</td>
</tr>
<tr>
<td>Operating System</td>
<td>Operating System</td>
</tr>
</tbody>
</table>

**Figure 1.1** A Generic Mobile Agent System
1.2 **Benefits of Mobile Agent Paradigm**

The mobility and autonomous nature of mobile agents bring potential advantages beyond the concept of Remote Procedure Call (RPC) [PRC] and Mobile Code [CPV 97]. In RPC, a process on one computer can communicate with another computer to perform a procedure call on the remote machine and receive the result. The remotely invokable procedure is predefined and resides, stationary, on the remote server. Only the parameter and result data are sent back and forth between the two hosts. Mobile Code gives mobility to code where code and possibly additional data or parameters are sent from a sender's site to a receiver, where the code is executed. When executed, the code may produce a result that can be sent back to the original sender or other hosts. The result can be passive data or other code. With Mobile Code, the code is no longer stationary; the code itself can be sent back and forth through the network and executed on the remote host. When the code starts to execute, it will continue until finished. The mobile agent paradigm extends the concept of mobile code in that an agent (the code) can suspend its execution and dispatch to another host, where the process is resumed. The paradigm brings many benefits due to its autonomy and code mobility.

**Higher reliability and lower network traffic**

Because a mobile agent can perform a series of actions in a distributed environment automatically without contacting the original host, less communication to the original host is needed, and thus less error arises in communication. In the traditional computing model of RPC, computations are performed with interactions between parties on each of two ends. Every remote function call will normally involve the exchange of parameters and the result. If we need more function calls, we may still need coordination information to synchronize the communications. If we use a mobile agent, only two interactions are necessary: send out the agent and receive the result. Although the overall network traffic is not in all cases less [SS 97], communication becomes asynchronous and less interaction is needed, which will in turn provide higher reliability.
Facilitate real-time interaction
Because network communication is often unstable and sometimes has high latency, RPC or Mobile Code is often not suitable for the real time control of some vital functions. If we use an agent, which can move near to the spot of the event and then make decision and take actions there, we can ensure a quick response to unexpected events without involving the slow and unstable communication process.

Better Support for Mobile Clients
The mobile (wireless) network is one of the fastest growing fields in communications. Within a mobile network, users are connected and disconnected frequently because of their mobility and the unreliability of communication. Mobile agents are able to permanently stay in a network once they have been “injected” into it. This enables them to perform tasks for the user, even if he is offline, and obtain necessary information until the user re-connects to the network, thus allows asynchronous interaction.

Collaboration between geographically wide spread individuals or organizations
The agents could be used as “proxies” for people involved in some kind of collaboration. They could, for example, meet at a central server and negotiate with one another, based on the interaction policies defined by their owners.

1.3 Security Concerns
As we mentioned previously, mobile agents facilitate several kinds of network applications. However, it is the autonomous behavior of the mobile agents and the malicious environment of the Internet that give rise to various important security issues, for both the software agent and its execution environment. Three key security issues have been identified affecting common mobile intelligent agent systems [CHK 95, Chess 98, Gray 98]:

- Security thread of malicious host
- Security thread of malicious agent
1.3.1 Malicious Agent

In 1988, a self-replicating program was distributed on the Internet. Within hours it spread throughout the U.S., overloading hundreds of thousands of computers by spawning out processes unlimitedly until using up all local resources [WORM, Spafford 88]. This program is a "worm" as we called it after the incident, which is one kind of Mobile Code. It also marked the beginning of the long counter-evolution between computer system administrators and the malicious programs from the Internet. The most recent examples of such programs are e-mail carried worms such as Magistr, MyBabyPic and the notorious loveletter [F-SEC].

The agent platform is the execution environment for the mobile agents. It provides access to the file system, local executable code, peripherals, system memory and CPU cycles to mobile agents. These services have to be provided to mobile agents in an autonomous way so that the agent can perform its task without human intervention. However, mobile agent's ability will put the agent platform at risk if an agent becomes malicious. The security threats an agent platform faces from a malicious agent have been discussed in many papers [Chess 98, CHK 95, Farmer 96, Gray 98, Ylitalo 00]. In general, a malicious agent may launch the following attacks.

Abusing Resources
A malicious agent can partially or completely impede one or more computer services, or a mobile agent's access to some resource or services. For example, an executing mobile agent can constantly consume network connections so that network services cannot be provided to other agents or processes. An agent may also continually print garbage out of the printer. These kinds of attacks will lead to a deny-of-service of other host processes or even crash the host.
Damage to Files or File System
An agent may destroy the agent platform’s file system by deleting configuration files or other critical files. It may also change existing files to its advantage.

Unauthorized Access of Information
An unauthorized agent may again illegal or undesired access, or remove data from the host or other mobile agent. A mobile agent may access and steal sensitive information, for example, a user’s personal e-mail. Or an agent may use covered channels to transmit data in a hidden way that violates a host's security policy.

1.3.2 Malicious Host
While the agent platform may face attacks from a malicious agent, the agent faces even more security danger from the host. An agent relies on the resources provided by the agent platform to perform its functionalities, and it can do nothing without the services of the platform. The fact that all the resources (data and code) an agent has are exposed to its execution environment makes the problem much more complicated than that of the malicious agent.

Reveal and Manipulate Sensitive Information
A malicious host can spy out the code and data of the agent so that it may understand the task the agent will perform. Furthermore, it can alter the code or data, leading to the malfunction of the agent or change of the agent behavior.

Impersonate
A malicious host may impersonate another host. For example, a mobile agent may bring some secrets to a trusted party. A malicious host could make the agent believe it has reached the correct destination. In this case, the agent will release the secrets to the unauthorized host. A malicious host may also impersonate a mobile agent, which has a similar effect.
1.3.3 Unsecured Network Communication

Network communication on the Internet is always insecure. An eavesdropper may audit network communication and analyze the traffic, compromising communication secrecy. A man-in-the-middle attacker may intercept a communication and manipulate messages. He or she can even impersonate an authorized user and attempt to intercept messages intended for an authorized user.

1.4 Goals

Unless the security drawbacks described above can be addressed, the practical use of the mobile agent system will be highly restricted. A secure architecture for the mobile agent system is, thus, crucial. In this thesis, we try to design and implement a secure architecture for mobile agent systems in general. However, in order to explore security issues, we need to base our research on a typical mobile agent system. We use WAVE [Sapaty 92, 99] as our mobile agent infrastructure, which cannot ensure security yet. Also, We will give the design of a Wave Security Add-on (WSA) architecture to WAVE to make it a Wave Secure System (WSS) [VF 01a, VF 01b]. The design of the architecture is kept as general as possible so that it can fit into any mobile agent system that has a structure described in Figure 1.1.

Given the extensive scope of the topic, a complete software implementation of the system is far beyond the scope of one master’s thesis. Only part of the whole architecture can be implemented. However, this thesis will give a detailed design of the whole framework so that it can be used as a programming guide for further research.
1.5 Thesis Outline

In Chapter 2 we give an introduction to some background knowledge related to the thesis, including a brief overview of WAVE; a description of some cryptography mechanisms, and a survey of related work. In Chapter 3, we gave an overview description of our proposed Wave Secure System (WSS). The Wave Certification Infrastructure, which is used to distribute and authenticate identities in WSS, is explained in Chapter 4. The mechanisms we use to protect mobile agent platforms (wave hosts) and mobile agents (wave agents) are presented in Chapter 5 and 6 respectively. In Chapter 7, we describe some of the implementation details of WSS, and an application using WSS called wavetella. Finally, the conclusion and possibilities for future work are given in Chapter 8.
Chapter 2

Background

2.1 WAVE – a Novel Mobile Agent System

WAVE [Sapaty 99] is a novel spatial programming paradigm, language, and system for parallel processing of open distributed systems. It can be applied to arbitrary open, heterogeneous and unknown network environments and is able to solve problems in parallel, without any need for centralized control or information. WAVE models the physical and virtual world as a Knowledge Network (KN), which resides on a physical network such as the Internet. There is no restriction on the topology of the KN; the mapping between the KN and the underlying physical network can be arbitrary. A physical node, which one or more nodes of the wave KN are mapped into, is called a wave host. A wave host that provides the wave agent execution environment contains three major software components: 1) a wave interpreter, which executes the wave agent and manages the local KN nodes; 2) an OS interface processor, which enables the wave interpreter to access and update the local resources; and 3) a communication processor, which handles the network communication to allow the wave agent to migrate from one wave host to another via UDP or TCP. The Wave Interpreter (WI) is the grammar engine that parses and evaluates the WAVE program. The general structure of a Wave host is shown in Figure 2.1.
A WAVE program, or so-called wave agent, migrates itself autonomously from one KN node to another and interacts with the local environment to perform the task on behalf of the user who injects it. The wave agents are represented as character strings. They can move themselves to a different position in KN by using a hop operation. A wave agent is composed of an environment, variables, and code.

The environment contains the agent's identification and routing information; the variables are the data that an agent carries along; and code can be executed by a wave host. Figure 2.2 shows the computational model of WAVE. In the figure, there are 8 logical nodes (a – h) residing on a physical network of four wave hosts. The solid lines linking wave hosts are the physical network, while the dashed lines connecting the logic nodes form the knowledge network. The wave agent, which is represented as a rectangle, will start its execution from one node (node “a” in this example) and spread itself into neighboring nodes b and c, where it will continue execution, then spread further into e and h recursively until a goal node or a goal condition (node f) is met. It will then bring back the result to the initial node (node a). A detailed description of WAVE grammar can be found in Appendix A.
The WAVE system has been developed in C in the UNIX and LINUX environments by P.M. Borst of the University of Surrey [Borst 95]. A variety of applications have been developed in WAVE, including the modeling and control of mobile telecommunication networks, integration of distributed databases, simulating the mobile-IP protocol, BGP and some congestion control and routing protocols [SCBW 94, Sapaty 99, VM 96, VI 96, GLV 01]. As WAVE is a powerful mobile agent system which allows the creation and launching of agents of arbitrary sizes and behaviors, its lack of the basic security properties renders it extremely dangerous. For example, a wave agent can take the full control of a wave host, consuming all of its computing cycles and accessing all of its resources, or can even change the topology of the KN. WAVE has currently no
mechanisms to prevent or cope with a malicious wave agent’s behaviors. It is thus essential to develop and incorporate a security system into WAVE.

2.2 Cryptographic Mechanisms

The recent development of cryptographic techniques makes security in the open environment possible. It provides methods to protect communication as well as to identify communicating parties. In this section we will give a brief introduction to the cryptography mechanisms used in WSS.

2.2.1 One-way Hash Function

The one-way hash function, or Message Digest, is a very important concept in modern cryptography theory. "One-way" means it is easy to generate the code given the message, but almost impossible to generate the message give the code. For example, given x, f(x) can be easily produced, but given f(x), we can’t practically calculate the x. Here 'practically' means the reverse process can’t be finished in a long time, based on current hardware (e.g. 1000 years). A hash function maps an input of varying length into a fixed length output. A one-way hash function is like a ‘fingerprint’ maker, in that no matter how big the original data might be, we can always produce a unique and fixed size ‘fingerprint’. The fingerprint is unique, because given the fingerprint and the message, it is infeasible to produce another message that will produce the same fingerprint. Most of the one-way hash algorithms are publicly available; the security of the function is not the algorithm itself but its one-way property.

There are many one-way hash functions available. To name a few, there are MD4, MD5 and MD2 algorithms [Robshaw 94]. In WSS we will use the Secure Hash Algorithm (SHA-1) [NIST 94], one of most secure one-way hash functions known. It was
developed by the National Institute of Standards and Technology (NIST). It can produce a 160 bits fingerprint from the input.

2.2.2 Public-key Cryptography

Public-key cryptography [DH 76] is one of the most widely used encryption ideologies on the Internet. It uses asymmetric cryptography. The main idea is to use a pair of keys, a public key and a private key for encryption and decryption. Both keys can be used to encrypt a message, but only the key not used can decrypt it, that is when the private key is used to encrypt the message, the cipher text can only be decrypted by the corresponding public key. As their name suggests, the private key is kept secret and the public key is available to the public. If two parties on the Internet, A and B, want to exchange messages securely; the public key encryption can be used. Suppose A wants to send message to B, A will use B’s public key to encrypt the message and send it to B. In this case, only the private key held by B can decrypt the message. The security of the method depends on the secrecy of the private key. As long as the private key remains secret no one can decrypt the cipher text, even if he or she has both the encryption key (public key of B) and the cipher text.

RSA [RSA 78] is one of the first and most influential public-key algorithms existing. Developed by Ron Rivest, Adi Shamir and Leonard Adleman, it is named after its inventors’ three initials. The key pair in RSA is obtained from two large prime numbers, “p” and “q”. With p and q, public modulus is calculated using the formula \( n = p \times q \). After that, an encryption exponent, \( e \), is selected in such a way that the greatest common divisor of \( e \) and \( (p - 1) \times (q - 1) \) is 1. The public key of RSA is then made up of the pair of exponent and modulus \( \{e, n\} \). Then the decryption exponent, \( d \), is calculated so that

\[
d \times e = 1 \mod ((p - 1) \times (q - 1))
\]

The number pair \( \{d, n\} \) is then the private key. Assume we want to encrypt a message, \( M \). We divide it into blocks so that each block can be represented as a number less than the modulus \( n \). Without losing generality, we assume \( M \) has only one such block. To compute the cipher text, \( C \), we use the following function:
The decryption of this $C$ is similar. We use the following function:

$$M = C^d \mod n$$

The security of the algorithm relies on the fact that it is infeasible to determine $d$ given $e$ and $n$.

### 2.2.3 Digital Signature

By using a public-key cryptography entity, A can send a message to entity B securely, but the question remains: how can the entity B make sure that the message is really from A, and not an impersonator? To solve this problem, a digital signature is used. Entity A can do the following: one, use its private key to sign (encrypt) the message; two, use the public key of B to encrypt the result from step 1; and three, send the encrypted message to B. When B receives the message it will do: one, use its private key to decrypt the message; and two, use A’s public key to decrypt the result from step 2. If B can successfully finish the two steps then B can ensure that the message is from A, because only A can use A’s private key to sign the message. Here, we assume B knows the correct public key of A. This algorithm is inefficient, however in that we must encrypt the whole message twice. If the message is very long the encryption process will become lengthy. Instead, we can sign only the fingerprint of the original message, which is much shorter. The fingerprint is unforgeable, so a signature over it is equivalent to that of the whole message. The signing algorithm for A becomes this: one, calculate the fingerprint of the message; two, use A’s private key to sign (encrypt) the fingerprint; three, use B’s public key to encrypt the message; four, send out the cipher text together with the signature. When B receives the message he or she will verify the message this way: one, Decrypt the cipher text with its own private key; two, calculate the fingerprint of the decrypted message; three, use A’s public key to decrypt the signature; four, compare the decrypted signature with the calculated fingerprint; if they are match, it guarantees that the message is from A. An extensive survey of digital signature can be found in [Mitchell 92].
This protocol has several characteristics [Schneier 96]:

1. The signature is authentic
2. The signature is unforgeable since it can be produced only by A’s private key.
3. The signature is not reusable since the signature is a function of the document and cannot be transferred to any other document.
4. The signed document is unalterable.
5. The signature cannot be repudiated since B does not need A’s help to verify the signature.

In this thesis, we will use the RSA algorithm as the cryptographic algorithm for digital signature.

2.3 Related Works

For the past decade, many mobile agent systems have been built for both academic research and commercial purposes. Some of the systems did not address security problems at all, such as Messenger [FBDM 97]. Some of the systems only provided rudimental solutions for security. Therefore, security still remains the major issue of all systems to date. A complete collection of all the existing mobile agent systems is unnecessary since many of them did not have any security considerations, and many others had similar security enforcement models. Therefore, we have selected some of the most influential projects to discuss here. A comparison of the discussed systems is summarized in Table 2.1. A relatively complete list of all the existing mobile agent systems can be found in Appendix B.

2.3.1 Aglet

Aglet was developed by IBM Tokyo Research Lab. Aglets (mobile agents) are serialized Java objects that execute on Aglets Workbench (the agent platform) [Venners 97]. A developer can use the classes and methods defined in Java Aglet API [Lange 97] for aglet
creation and manipulation. The mobility of the aglet is achieved by the sterilization and dynamic class loading techniques of Java. An aglet serializes itself and dispatches to another Aglet Workbench, where it is loaded (executed) by the class loader.

A security model [KLO 97] has been defined for the Aglet. Every aglet has an identifier, to which appropriate security policies are applied. However, the system does not enforce the access control based on the aglet’s owner. A server simply trusts the aglets if they were sent from the server in the same domain [Aglet]. The servers within a trusted domain must authenticate each other by using a MAC (Message Authentication Code). Aglets are shielded by proxy objects, which provide language-level protection as well as location transparency. The Aglet Workbench will enforce the access control according to the aglet’s manufacturer, owner, and the aglet itself. The manufacturer and aglet’s identification can be built into the aglet. Also, no technique for the protecting aglet is provided.

2.3.2 Concordia

Concordia is developed by Mitsubishi Electric Information Technology Center America [Wong 97]. It is also a Java-based mobile agent system. It provides a Concordia Server that executes on top of the Java Virtual Machine as the agent platform. Like Aglet, agents in Concordia are serializable Java objects.

Concordia’s security model provides support for three types of protection: one, agent storage protection, two, agent transmission protection and three, server resource protection [WPW 97]. Each agent is associated with a particular user, and carries a hash code of its user’s password. The access control is enforced by using the SecurityManager feature in Java according to the user’s identity. The user passwords are stored in a global password file, which makes Concordia hard to scale up, if not impossible. Furthermore, protecting an agent from a malicious server is not supported.
<table>
<thead>
<tr>
<th>System Name</th>
<th>Identify Agent</th>
<th>Public Key Distribution</th>
<th>Protect Host</th>
<th>Protect Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aglet</td>
<td>No identification method is needed. Any aglet sent from the same domain is trusted.</td>
<td>N/A</td>
<td>Aglet Workbench enforces access control by using proxy objects. Only two security categories: trusted and untrusted.</td>
<td>Not available</td>
</tr>
<tr>
<td>Concordia</td>
<td>Global password</td>
<td>N/A</td>
<td>Use SecurityManager, which is a Java built-in feature.</td>
<td>Not available</td>
</tr>
<tr>
<td>D'Agent</td>
<td>Digital Signature using PGP</td>
<td>No global distribution method available</td>
<td>Agent will execute on Safe Tcl environment. The access control is enforced according to security policies. Only two security categories: anonymous and authenticated.</td>
<td>Not available</td>
</tr>
<tr>
<td>SOMA</td>
<td>Digital Signature. Only the agents from untrusted domains need to be authenticated.</td>
<td>X.509 Certification</td>
<td>Access control is enforced according to layered security policies based on the agent's role.</td>
<td>MH is used to protect the integrity of the collectable data.</td>
</tr>
<tr>
<td>WSS</td>
<td>Digital signature</td>
<td>WCI</td>
<td>AA Resource Manager enforces the access control rules and policies based on the agent's identity and privilege.</td>
<td>Provides mechanisms to protect agent's code and data integrity.</td>
</tr>
</tbody>
</table>

Table 2.1 A Comparison of the Security Solutions of Five Mobile Agent Systems

2.3.3 D'Agent

D'Agent, formerly named Agent Tcl, is one of earliest mobile agent systems. It was developed in Dartmouth College [Gray 95, 98]. Unlike Java based mobile agent systems, D'Agent uses Tcl interpreter as the agent platform. D'Agent also supports several kinds of programming languages, including Tcl, Java, and Scheme.
D'Agent uses a Safe Tcl execution environment [LO 95] to enforce the access restrictions. Every agent has an identity that can be verified by its signature, based on which the access control is enforced. However, the access control is of a very coarse granularity, such that all agents arriving from a particular machine are subjected to the same access rules. D'Agent uses an external program – PGP [Zimmermann 95] to realize key distribution and encryption functionalities. D'Agent also does not provide any technique to protect the agent.

2.3.4 SOMA

The Secure and Open Mobile Agent (SOMA) [BCS 99], developed at the University of Bologna, is another mobile agent system implemented in Java. What makes SOMA unique from other Java based mobile agent systems is that it interoperates with CORBA [CORBA], so that it can be integrated into CORBA-compliant environments. A SOMA agent (a Java program) executes in an environment (the agent platform) called SOMA place, which represents physical machines, and the SOMA places can be grouped into domains that represent LANs. Places and domains provide two layers of abstraction that represent the Internet.

As its name suggests, SOMA takes security into consideration at a very early stage of its design; therefore, it provides a relatively rich and comprehensive solution for security problems. It uses a location-independent naming scheme for mobile agents' identities, which can be verified by the agent owner's digital signatures. The public keys of the agent owners are distributed by using X.509 Certification Infrastructure. Only the agents from the untrusted domains are subject to authentication checks and the agents from trusted domains will be trusted automatically. Furthermore, all the agents are subject to an authorization check before any access to a place’s resource, based on the agent’s role. However, the access capability is not differentiated by the agent’s identity; all the agents with the same role have the same access privilege. SOMA uses a so-called Multiple-Hops (MH) protocol to protect the integrity of agents [CMS 99]. MH only protects the
integrity of the *Application Data* (AD), which is collected from *places* during the migration of the agent, however, the integrity of the agent’s code and other possible kinds of data is not protected.
Chapter 3

System Design Overview

Our goal is to design and build a Wave Secure System (WSS). This is achieved by integrating a WAVE Security Add-on (WSA) to the original WAVE system. We call the security architecture an “Add-on” because we want to leave the original WAVE system untouched. We attempt to keep changes to the original WAVE system to a minimum. The WSA exists as a thin layer that wraps around the wave host to guard the interaction between the wave agent and host, as well as between the wave host and the underlying operating system. Figure 3.1 shows the general architecture of a Wave Secure System.

![Figure 3.1 The Wave Secure System](image)

Figure 3.1 The Wave Secure System
3.1 Design Criteria

Secure: Security, of course, is the first property that the WSS should entail. WSS should also provide a secure environment both for the wave agent and the wave host. The WSA, which performs all the security functionalities for WAVE, involves network communication itself, so the communications should also be secure.

Transparent: The user should not be made aware of the authentication, access control and secure communications that take place in the WSS. Because of the long history of WAVE, there are many existing applications for it. The new WSS should ensure the proper execution of these applications. The grammar of the WAVE should remain unchanged or should be "backward compatible".

Scalable: The WAVE system is an emerging technique. A secure architecture must meet the huge growth potential of the WAVE system. That requires the naming and identification scheme of the WSS be scalable.

Potable: As we emphasized, the design should be kept as general as possible. Only minor changes should be required if we port the system to other mobile agent systems. It also should work with different operating systems.

3.2 A Research Framework

3.2.1 Security Functions

In the introduction chapter, we described the security challenges that mobile agent systems face. To provide security solutions to these challenges, our system must provide the capabilities to one, protect the wave host from malicious agents, two, protect the agent from malicious hosts, and three, protect network communication. Solving the
problem of protecting the host entails two tasks: authentication and authorization. Authentication involves identification of all the entities within the system. Authorization concerns the access control of the entities based on their identities. Proper access privileges should be identified before the allocation of resources to ensure a safe access control procedure. The problem of protecting the agent is more difficult to solve than the other two. It is believed that no complete solution yet can be provided without the help of tamperproof hardware. We try to provide some schemes to address the following four issues: authentication, authorization, protecting agents and protecting communication.

Authentication
The problem actually contains two tasks. The first task is to give identification to all the entities in the WSS; the second task is to authenticate these identities. Because the WSS will exist in an open and distributed network such as the Internet, its identity and authentication must be enforced without any central knowledge. Therefore, a simple approach using a user name and password to identify entities in the system is not suitable. Instead, the decentralized method (e.g. public-key cryptography and digital signature) must be employed. However, to ensure the quality of secure services, there must be some degree of authority to supervise the distribution of trust. A trade-off between uncontrolled distribution and central control must be considered in the authentication system.

Authorization
After the authentication of a wave agent, proper authorization must be recognized. The access control is what authorization is all about. That is, to what extent an agent with a certain identity can use the services provided by the agent platform. The access control is not only a means to control its access to the physical resources, such as a file system (hard drive), but also the behavior of the agent. Systematic access control rules must be provided. The security policies on how to carry out these access rules should also be explained. Before a wave host provides any services to an agent, a strict security check must be performed based on these policies and rules.
**Protecting the Agent**

The system must provide techniques to protect the code and data integrity of the agent. One aspect of the problem is to protect the agent from the malicious host, which is so difficult that indeed, Gray claimed that without the hardware support, it is impossible to prevent a machine from doing whatever it wants with an agent that is currently executing on that machine [Gray 98]. However, we believe some schema can be proposed to protect the agent, such as its code and data. Another important aspect of the problem is how to protect an agent from the attack of other agents. We believe the solution of the second problem is analogous to protecting agent from the malicious host, since the interaction between the agents is always through a host, so that from an agent’s point of view, it cannot tell who initiates the interaction.

**Protecting Communication**

The communication between the wave interpreters involves the migration of the wave agent and the wave system management information. Communication can be protected by setting up secure channel between the wave hosts. The Secure Socket Layer (SSL) [SSL] is the most widely used protocol for secure network communication nowadays, which provides authentication and encryption services for TCP connections. The SSL provides encrypted communication so that eavesdropping attacks can be prevented. The SSL also provides mutual authentication of both sides of the connection so that man-in-the-middle attacks can be prevented. The SSL can be plugged into applications at the socket layer; thus the application does not need any special security knowledge or security related code regarding SSL. In this case, the network communication security is hidden at the socket level, so that we can simply replace the original TCP socket with the SSL socket in the wave communication processor without any change of the wave interpreter or the wave OS interface processor. A C implementation of SSL is available as open source. Because the problem of protecting network communication is separated from other security considerations, and because it can be solved by simply integrating the SSL into the WAVE, we will not discuss in detail this problem in this thesis.
3.2.2 Modularity

To meet the system criteria of transparency and scalability, we decided to put all the security functions into a Wave Security Add-on. The design and implementation of the WSA is strictly separate from the original WAVE system; therefore, the functions of the WSA can be encapsulated as a separate module and the following goals can be met.

Separate the Secure Functions from the Wave Interpretation Process
The wave interpreter (WI) has minimum responsibilities to the security issues; almost no change needs to be made. The design of the security functionalities will not be restricted by the WI interpretation process.

Easy to Port the WSA to Other Mobile Agent Systems
Because of the separation of the WSA, the design will be very general. It will be easy to port the system to other mobile agent systems, which have similar system architecture.

Flexible Design of the WSA
We will have more flexibility in choosing programming language and algorithm implementation once the interface between the WSA and WAVE is fixed. This can facilitate our use of up-to-date security techniques, such as cryptograph techniques.

3.3 Principals in the System

The Wave Secure System is composed of so-called principals. A principal is an entity whose identity can be authenticated. Each principal performs a specific role and has its own responsibilities and interests. Each principal, except the wave agent, is also associated with a digital certificate (see Chapter 4 for details). The certificate binds a principal to its privilege, role and public key, where the privilege represents the trust the certificate issuer gives to the certificate holder (subject); the role restricts the behaviors the subject can perform; and the public key is used to encrypt data or verify the signature.
Inside the system, the identity is determined by the signature of the principal and the security policy is performed according to the principal's privilege. The principals in the WSS are described as follows:

**Injector**

An *injector* is a user who owns the agent, and who injects the agent into the wave KN. In some systems such as Aglet, the code manufacturers, code injector and agent owner are considered to be different entities. In WSS, we only have the injector, who is the owner of the agent, but may or may not program the agent itself. The injector can also be called a user, since every injector is a registered user of an Authentication Agent.

**Identity:** An injector’s identifier together with an Authentication Agent’s identifier uniquely identifies the injector. The injector’s identifier is unique only within an authentication agent’s domain.

**Roles:** An injector initiates the execution of a wave agent at a particular KN node.

**Wave Agent (WA)**

A *wave agent* belongs to the injector who injects it into the system.

**Identity:** The agent’s Serial Number (SN) and its owner’s ID. The agent’s SN needs to be unique only among the agents injected by the same injector. The identity will be signed by the agent’s injector and the agent’s privilege is inherited from its injector.

**Roles:** The mobile agent performs a task on behalf of its injector.

**Authentication Agent (AA)**

An *Authentication Agent* is a stationary agent, which always resides beside a wave interpreter. Every AA will have an administrator as its super user. The administrator has the right to configure the AA and maintains the AA’s injectors.

**Identity:** AA is one per wave host so that an AA’s identity is determined by the address of its residing host, for example, AA1.cs.ubc.ca. The wave interpreter does not have its own identity, but it can be identified via the AA’s identity.

**Roles:** The AA fulfills several roles, such as:

- User account management
Authentication Center (AC)

An Authentication Center is a special-purpose host. It manages the local WSS domain. Unlike the AA, AC does not have any wave interpreter associated with it. An AC has only one super user: the administrator who can configure and change the states of the AC.

Identity: AC’s identity is also determined by the residing host’s address, for example, AC1.ubc.ca.

Roles: The AC fulfills several roles, such as:

- AA certificate issuing and management
- Wave agent registration and management
- Local domain security policy maintenance
- Passport issuing
- Visa issuing

3.4 Naming Scheme of The Principals

Inside a WSS domain, the AC is the root of all the principals. One WSS domain is composed of one AC and several AAs, so users (injectors) register under an AA. A DNS style naming system is used to name the principals in WSS. The name of an AC depends on the domain name of its residing host. The home AC is at the root of the local namespace. If we have a WSS domain named cs.ubc.ca, all the principals can be named as in Figure 3.2.
In designing the WSS, our goal is to develop a generalized security approach that requires little or no change to the original WAVE System. Thus, we came up with the architecture of an Authentication Agent as shown in Figure 3.3. In this security architecture, an Authentication Agent resides beside the wave interpreter; the AA acts as a “wrapper” that around the wave interpreter so that all interactions between the wave interpreter and its environment go through the AA for security actions. An AA interacts directly with the wave interpreter through three components: an AA Authentication Manager, an AA Communication Manager and an AA Resource Manager. In addition, there are three other components, which are not shown in Figure 3; they are the AA User Interface Manager, the AA Security Policy Manager and the AA-AC Interface. The user interface manager provides a Graphic User Interface (GUI) to the user for account management and wave agent injection. The security policy manager selects and implements a local security policy. The six components of an AA can be summarized as follows:
AA User Interface Manager: Provides the interface for the user to access and manage the AA (wave interpreter). The user can only inject a wave agent through the interface manager.

![Diagram of Domain 1](image)

**Figure 3.3 Structure of the Authentication Agent**

AA Authentication Manager: Maintains a local user's certificate list as well as an executing agent's certificate list. It checks the identity and the privilege of an injector/user when the user injects an agent; the authentication manager also registers the wave agent.

AA Resource Manager: Supervises the wave agent's access to local resources through the wave interpreter. The privilege of the agent is inherited from its injector; the identity of the agent's injector is also considered if the security policy requires. It will operate with the AA Authentication Manager and the AA Security Policy Manger to perform the access control.

AA Communication Manager: Adds the agent's identification information to the agent (wave program) when the agent goes out of the interpreter (wave host), and removes and checks the identification information when the agent arrives at the wave host from another host. However, encryption for secure communication is not the purpose of this manager.
AA Security Policy Manager: Maintains a table of access permissions named *Local Policy Map* (LPM), which maps the privilege to local resources. The access table can be customized at the wave host level according to local interests. The security policy manager, together with the resource manager, allow access control to be customized and enforced.

AA—AC Interface: Manages all communication between the AA and its home domain AC. These communications are separated from the agent's migration; thus, separate concerns of security are applied.

### 3.6 Structure of the Authentication Center

To achieve scalability criteria as well as to meet the structural requirements of Internet, the WSS is organized by domains. These domains form an inter-network of WSSs. All the wave hosts within one domain usually belong to a particular organization. An AC is the authority on security issues of such a domain. One AC manages all the wave hosts in the domain. An AC controls two aspects of the task, one is the intra-domain task, and the other is the inter-domain task. Both tasks involve distribution and management of certificates and security policies. The AC is not accompanied by a wave interpreter, so it does not provide the execution environment for the wave agent. The AC has following basic components:

AC Certificate Manager: It will assign the certificate to underlying AAs and manage the certificates of agents traveling across domains.

AC Policy Manager: It maintains the foreign policies of a wave domain. One aspect of foreign policy is the balance of different policies defined by different wave domains. The visa-issuing policy is defined to decide on privileges when assigning visas to foreign agents. The other aspect is deciding the privilege when assigning passport to local agents, as defined in the passport-issuing policy. (see chapters 4 and 5 for details on passports and visas)
AC Agent Registration: It maintains a table that stores the agent registration information. It is used for agent integrity check (see chapter 6).

AC—AA Interface: It manages all the communication between the AC and its AAs. These are for certificate distribution and agent integrity check.
Chapter 4

The Distribution and Authentication of Identities

In an open network environment, the wave agents sent between wave hosts are like anonymous postcards that are anonymously routed and answered. The identities of these agents are hard to define due to the distributed nature of the Internet. Digital signature technique provides a solution to the problem so that we can assure communication is happening between the desired endpoints, and that it is tamperproof.

In WSS, we assign each principal a pair of keys – one is a public key another is private key. The identity of a wave agent is guaranteed by its injector’s signature, signed by its injector’s private key. Given the injector’s public key, every principal can verify the agent’s identity and no one can forge the signature without the corresponding private key. However, there are still problems that can’t be solved by public-key cryptography and digital signature themselves. They are the following ones: one, the problem of distributing the public keys, how one principal obtains the public key of another principal; two, the problem of binding the key to the principal, how does one principal know which public key belongs to whom in a tamperproof way; and three, the problem of key revocation, if one principal’s public key is changed or tampered with, how can other principals know and take appropriate action about the key. Also, in an environment where it is safe to freely exchange keys via public servers, man-in-the-middle attacks are
a potential threat. In this type of attack, someone posts a phony key with the name and user ID of the user's intended recipient. Data encrypted to — and intercepted by — the true owner of this bogus key is now in the wrong hands. These problems can be solved by digital certification technology, where certificates provide a strong binding between the public-key and some other attributes. These attributes are tamperproof and can be used as convenient references to help someone receiving a message decide whether that message, the key and possibly the sender's name are what they appear to be without asking the sender [Gerck 00].

4.1 Two different approaches

There are two widely used certification systems. One is based on X.509 [X509] protocol and the other is based on Pretty Good Privacy (PGP) [PGP] protocol. These two protocols reflect two methodologies and two different approaches to the distribution of trust on the Internet.

X. 509

X. 509 is a standard recommended by International Telecommunication Union (ITU). It is based on the concept of directory service where directory means a server or a distributed set of servers that maintains information about users. These servers are called the Certification Authority (CA). The CA issues certificates to subscribers (CA clients). A certificate is, according to X.509v3 definition, “the public keys of a user, together with some other information, rendered unforgeable by decipherment with the private key of the certification authority which issued it”. It is a top-down approach, where you can build a hierarchical tree: the top level CA issues the certificates of immediate lower level CAs, where the lower level CAs are subscribers of the top level CA (the issuer). A Certification Practice Statement (CPS) governs the legal and technical relationships between a CA and its subscribers. When an issuer assigns and signs the certificate to a subscriber, it will guarantee the correctness of the certificate under CPS. The correctness of a certificate, and thus that of the identity claimed by the certificate holder, depends on
the issuer. When a certificate is revoked, its serial number will be posted on the Certification Revocation List (CRL). Because a CA has no idea of where the certificate going to be used, it is the user's responsibility to check the CRL regularly to update a revoked certificate. Before a user checks the CRL, he or she never knows a certificate is revoked.

Pretty Good Privacy

The Pretty Good Privacy (PGP) is based on a referral model where referral means that the certificate depends on the integrity of a chain of authenticators. The authenticators are the users themselves. The users and their keys are referred from one user to the other, as in a friendship circle, forming an authentication ring. The public keys and certificates are usually self-certified or certified by friends. You may not know somebody in the ring, but you hope someone else you trust does. There is no central control on the certificates, so the maintenance (adding or removing of certificates etc.) of them is also performed by the users themselves. This is due to the fact that PGP is an Internet phenomenon. It is not designed by an official organization, it is invented and developed largely by one person, Phil Zimmermann, and its source is publicly available.

The X.509 model used centralized control of trust where a common trusted authority CA certifies and manages all the certificates. However, it is in natural opposition to the concept of the open network. If an upper-level CA's certificate is revoked due to private key corruption or update, all the underlying subscribers' certificates have to be revoked as well. This is because if the private key of the CA is corrupted, with that CA's private key, the whole certification infrastructure underneath it can be forged, and all the certificates signed by the CA will be considered to have been tampered with.

The PGP, on the other hand, is totally decentralized and no central authority like CA is needed to issue the certificates and govern their management, so that there is no single point of failure. However, because the maintenance of the certificate is done in a do-it-yourself fashion, it is very difficult to scale up the whole system. Also, if someone's private key is compromised, it is very hard to indicate to other users that the according
public key should be revoked, since there is no complete list of users who have that public key in their key ring and there is no "official" and publicly trusted place (i.e. CA in X.509) to publish the revoked key list. With these drawbacks, PGP faces the problem of no guarantee of accountability, coherence, dependability and correct authentication, which restrict its commercial application.

Also, both approaches will bind a public key with its owner's identity so that the correctness of the public key is to some extend guaranteed, however it cannot provide any information on how much you can trust the subject with that identity.

With the advantages and disadvantages of X.509 and PGP in mind, we would like to design the WCI as a mixed model of both, which can obtain the benefits of both approaches and avoid their weaknesses. The AA Authentication Manager and AC Certificate Manager provide the functionalities of the WCI.

4.2 Wave Certification Infrastructure (WCI)

The WSS is organized by domains. All the wave hosts in one domain usually belong to one organization so that a consistent and integrated trust can be deployed and centralized control over the certificates made practical. Within a WSS domain, a hierarchical certificate management structure is used. Between wave domains, there is usually no common authority that both domains trust, so that a commonly agreed security policy between the two domains must be decided. Based on mutual agreement, a passport-visa approach is used for between-domain certificate distribution.

4.2.1 Intra-domain Approach

Every injector in the WSS domain has an account at an AA, which is called the home AA of the injector. Each injector only belongs to one home AA, so that the account is kept
local to the particular AA. Home AA is responsible for the issuing certificate to its injectors. The domain AC, so called home AC to all other principals in the same domain, issues certificate to its AAs. Home AC and AAs are issuers who issue certificates to AAs and injectors respectively and AAs and injectors are holders who obtain certificates from issuers.

**Key pair generation**

Key pair generation is the responsibility of the certificate holders. It only happens in two cases: when the holder registers the issuer for the first time, or when the holder's key pair needs to be updated. In both cases, the holder will generate the key pair and then submit the public key to the issuer. The issuer only manages the account and the public keys of the holders; it does not know the holder's private key. Because the holder and issuer usually physically reside on different machines, if an issuer is tampered with, the private keys of its registered holders will not be tampered with.

**Certificate distribution**

The AC will distribute the certificates of all the AAs so that every AA knows the certificate of all the other AAs. The user's certificate needs not be distributed explicitly. It will be carried by the wave agent during its migration. The validity of the user's certificate can be verified by the issuer's (home AA's) certificate. The AC is "self-certified", which means that the AC issues itself a certificate. It will distribute its own certificate to all the AAs in the domain. This is usually done manually, for example, the administrator copies the certificate to a disk and physically distributes to all the AAs.

**Certificate Verification**

To verify a certificate, we need its issuer's public key to verify an issuer's signature on the certificate. The issuer's public key can be found from its certificate, which must be verified by its issuer's public key. This verification process forms a certificate chain, on top of which is the home AC's certificate. In one WSS domain, at most two certificates are needed to verify an injector's certificate.
Certificate revocation

A certificate needs to be revoked when its content is subject to change or suspected of being tampered with before it is expired. The revocation problem is critical because the original certificate at large appears to still be correct until we revoke it and anyone who uses it will be at risk. When a certificate needs to be revoked its issuer will generate a certificate revocation request (CRR) to the home AC and the serial numbers of the revoked certificates will be submitted. The AC has two ways to circulate the information to all the AAs, one, publish all the revoked certificates on a Certificate Revocation List (CRL), and all the AAs will periodically lookup the CRL and update their Revoked Certificate Buffer (RCB). Two, push the information to all the other AAs while the AC receives a certificate revocation request (CRR) from an AA.

Identify a Wave Agent

A wave agent’s identity is certified by its injector’s signature. The agent will always carry the signature during its execution and migration. The signature can be verified by a chain of certificates: the injector’s certificate + home AA’s certificate + home AC’s certificate.

4.2.2 Inter-domain Passport and Visa Approach

In the discussion above, the certificates are kept locally inside one WSS domain; however, the wave agents may migrate across domain boundaries. When a wave agent moves itself outside of the home WSS domain into another domain, how can we identify it? There is no common authority between the two WSS domains that both trust, therefore, a certificate from a foreign domain cannot be verified. Also, the privilege in a certificate defined in one domain will be meaningless in another, since different domains may deploy different security policies and define different meanings for the same privilege. We can explore the problem with an analogy of people traveling between countries. Suppose Bob wants to travel to Canada from China. What Bob needs to do is to get a passport from the Chinese government and then apply for a visa from the
Canadian embassy. With the passport and the appropriate visa, Bob can legally enter Canada. However, Bob's behavior abroad is restricted by the visa, for example, Bob cannot legally attend school in Canada with only holding a visitor's visa. (In reality, a Student Authorization is required for a foreigner to study inside Canada.) The same approach can also apply to the WSS. When a wave agent travels out of a local domain, it needs to obtain a passport from the home AC, and when it arrives at the other WSS domain's AC, it has to request a visa. In this case, we suppose the two ACs of the two WSS domains have already had an agreement on the security policy, which is usually obtained off-line and stored in the Policy Mapping Table (PMT). A passport is a certificate that is issued by the home AC and the visa is a certificate issued by the foreign AC. Given the fact that the AA and the AC are stationary, the approach is only applied to the wave agents.

**Passport issuing**

The passport is a special certificate (same format as other certificates) prepared for a local agent who will travel to foreign WSS domains. It is issued by the agent's home AC. The AC will assign the agent a new privilege in the passport based on the AC's foreign policy. The agent's role and public key remain the same. When the agent travels outside of its local domain, it will carry the passport but not its injector's certificate. The public key in the certificate (passport) is still its injector's, however, the issuer of the certificate (passport) becomes its home AC. All the wave agents that were injected by the same injector will use the same passport.

**Visa issuing**

The Visa is a special certificate (same format as other certificates) that an AC issues for foreign wave agents. Before a visa can be issued to the foreign agent, two prerequisites must be met, one, the agent must carry a passport, and two, the AC who issues the passport must already have an inter-domain agreement with the visa issuing AC. The content of the visa is based on the agent's passport, its access request and the inter-domain agreement. A privilege for the agent in the foreign domain is assigned in the
visa. The visa binds the agent injector’s public key with the new privilege. Inside the foreign domain, the agent will always carry the visa.

Passport/visa revocation
The passport of an injector is associated with its certificate, and usually they have the same lifetime. When a certificate is revoked, its according passport, if any, will be revoked as well. We use the same revocation strategy to it as to the ordinary certificate. Because a visa is only for the foreign agent, it has only a very short lifetime, such as a few hours. No explicit revocation for the visa is needed. Even if the passport is revoked after it is used in a foreign domain, no explicit message is needed from its home AC to the foreign domain.

Identifying a Wave Agent
When the foreign wave agent is executed on the wave host, its identification is also identified by its injector’s signature. Because the visa binds the public key of the agent injector’s public key with its identity, the signature can be verified by the agent’s visa.

4.2.3 Certificate Structure
The certificate, passport and visa all have the same format, so the description of the certificate’s structure also applies to the other two. Figure 4.1 is the certificate structure encoded in ASN.1 [Dubuisson 00] distinguished encoding rules (DER) [X208]. ASN.1 DER encoding is a tag, length and value encoding system for each element.

A wave certificate contains a certificate body and a signature value. The signature value is computed upon the ASN.1 DER encoded WAVECertificate using SHA-1/RSA algorithm. By generating the signature, the issuer of the certificate certifies the validity of the information in the WAVECertificate field. In this case, the issuer binds the public key, role and privilege to the subject who holds the certificate. The certificate body WAVECertificate field has the following fields:
Serial Number: A serial number is an integer the issuer assigns to each certificate. It must be unique for each certificate issued by a given issuer. The issuer's name together with the serial number can uniquely identify the certificate.

Figure 4.1 An ASN.1 DER Encoding of Wave Certificate

Certificate Type: The certificate type is an integer that indicates the certificate is a passport, a visa or an ordinary certificate.

Role: There are three roles in the WSS that can appear in a certificate; they are the injector, the AA and the AC. A role of the subject restricts the behavior of it, for example, an injector cannot sign any certificate and an AA cannot sign a passport or a visa.
Issuer: The issuer field is the name of the principal who has signed and issued the certificate. An issuer can be an AC or AA.

Validity: The certificate validity period is the time interval during which the issuer warrants that the information provided by the certificate is maintained. This field has two dates: the date when the certificate validity period begins (notBefore) and the date on which the certificate validity period ends (notAfter). The date is encoded as UTC (Universal Time Coordinated) Time.

Subject: The subject is the holder of the certificate. It can be an injector, AA or AC. The name of the subject must be unique within one issuer. The issuer name and subject name will uniquely identify the subject.

Privilege: This field is the privilege of the subject. The privilege represents the trustiness the issuer gives to the subject and it has a value from 0 to 255. The value 255 means full trust, and the value zero means no trust at all.

Subject public key: This field carries the public key of the subject. In the implementation, it is a 1024 bits RSA public key.

4.2.4 Three Levels of Trust on a Wave Agent

The signature and certificate carried by a wave agent give a tamperproof method to verify the identity of the agent. Also, this authentication process determines to what extent we can trust the agent. When an agent is executed on a wave host, there are three levels of trust of the agent.

Anonymous Agent: An anonymous agent is an agent whose identity cannot be verified. This includes an agent who has no signature or whose signature cannot be verified due to insufficient information.

Identified Agent: An identified agent is usually an agent coming from a foreign domain without holding a valid visa. Its identification can be verified by the domain AC; however, a visa cannot be assigned because the agent does not carry a passport or there is
no inter-domain agreement among the two ACs (this domain AC and the agent's home AC).

**Trusted Agent:** The trustworthiness of a trusted agent depends on its privilege. All the agents from local domain are trusted. A foreign agent with valid visa is also trusted.

![Diagram of WCI as a Branch of X.509 Directory Service](image)

**Figure 4.2** WCI as a Branch of X.509 Directory Service

### 4.2.5 WCI as a Branch of X.509 Directory Service

The X.509 protocol is the overwhelming commercial standard nowadays. It is widely used in e-commerce and is implemented in most modern programming languages, and supported by most large software companies and web sites. There are quite a few security technologies using the X.509 protocol, including Netscape, IPSec and SSL etc. Conforming to the standard is one of the most important tasks a software designer will face. Our WCI can be easily adapted to the X.509 infrastructure without much change. Conceptually, an AC has all the functionalities of a Certification Authority (CA) in
X.509. By registering the AC to an existing X.509, CA can naturally turn a WSS domain into a branch of a X.509 directory service. In this case, the certificate of a domain AC will be certified (signed) by a CA, so that everyone who honors the CA’s certificate will honor the AC’s certificate. Figure 4.2 shows three wave domains registered to a X.509 CA. The arrow shows the direction of certificate issuing. The double-headed arrow with dashed lines indicates that there is an inter-domain security agreement existing between the two wave domains so that when an agent travels across domains a passport and visa approach is practiced. All the agents originating from domain 1 or 2 are trusted agents in both domains, however, if these agents travels to domain 3, they are not trusted but can be identified, since the certificate of the injector can be verified by a chain of certificates whose root is the CA’s certificate.

4.3 Benefits of Using WCI

Matches the real world structure
The Internet is the interconnection of the networks. These networks belong to different organizations. There is no central control between these networks. However, inside these networks some central control must take part to guarantee the accountability, dependability and correctness of security checks. The WCI is designed to reflect this characteristic of the Internet. Inside one WSS domain, the AC acts as a central authority. Between domains, domain ACs must negotiate with each other on security issues, and no centralized arrangement is applied.

Easy to revoke certificate
In a WSS domain, there are only two levels of certificate issuing. The home AC can do all the revocation, including the AA’s and the injector’s. Also, because an AC knows all the AAs inside the domain, an AC can push all the revocation information to the AAs without delay. All the certificates as well as passports can be revoked in this way. An ordinary certificate will not be distributed across WSS domains, so we do not need to
revoke a certificate from the foreign domain. Because the visa usually has a very short life, no explicit revocation is needed.

**Hide Injector’s identity**

An AC is like a proxy for the agents from its domain because for the outside world, all the agents from that domain carry a certificate issued by their home AC. The identity of the agent’s real injector will not be revealed to foreign domains.
Chapter 5

Protecting the Wave Host

By using the Wave Certification Infrastructure we can authenticate the identity of an agent, therefore we can protect the wave host by restricting the behavior of the agent based on its injector's identity. A wave agent uses the services a wave host provides by performing a series of operations. These operations are restricted by access control rules, which categorized operations into different security levels. Each security level is subjected to a certain set of security checks. The security policy exercised by the host restricts the behavior of the principals and maps the individual operation in a certain security level. By defining their local security policies, all wave hosts are able to enforce their own access control. Also, there are foreign security policies to guide passport and visa issuing. The AA Resource Manager is used to practice these access control rules. The AA Policy Manager maintains the local security policies and the AC Policy Manager maintains the foreign security policies.

5.1 Protected Resources

When we discuss the issues of protecting the wave host, we are always mean to protect the resources of the wave host. We would like to restrict a principal's access to these
resources as well as the amount of the resource one can consume. The resources we want to protect are: the file system, native program, network, peripherals, CPU and memory.

5.1.1 File System

The file system is the major part of the local host environment. The wave agents can use the file system to exchange information with other agents, collect information, and to perform file management tasks. The ability to access the file system is the basic function of a wave agent; however, misuse of this power can be hazardous to the local wave host. For example, the corruption of the file system may cause a crash of the host machine. Malicious agents can one, continuously writing hard disk to perform a deny-of-service attack to prevent other processes from using the disk, two, stealing unauthorized, critical information, three, manipulate/remove important files so that the agent platform or the underlying operating system cannot perform function properly. In order to prevent these attacks, we need to take full control of the read and write access to files and directories.

A wave agent looks at the local file systems as two categories of directories: the WSS directories and non-WSS directories. The WSS directories can be a public shared directory or temporary/permanent personal directory. A public shared directory can be accessed by all the agents with a minimum required privilege. All the files in the public shared directory are readable to any wave agents, however only the file owner can write the file. A quota is set for the whole directory, and every wave agent can use the storage up to its limit. All the files in the public shared directory are temporary and only exist during the owning agent’s running time. A personal directory is built and used by a specific injector’s agents. It can be either temporary or permanent. The temporary directory exists only when any of the injector’s agents is running on the local host and the permanent personal directory can exist on the host permanently. The WSS sets a quota limit to both directories. The directory owner’s agents have the read/write permission of the directory. Only the authorized agents have read permission of the directory, where the directory owner defines which agents are authorized. All the directories other than the WSS directories are non-WSS directories. Only an agent with very high privilege can
read these non-WSS directories, and no wave agents have write permission to these directories. An example of the WSS file system access is described in Table 5.1.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Life</th>
<th>Operation</th>
<th>Authentication</th>
<th>Quota</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSS Dir.</td>
<td>Public</td>
<td>Temperate</td>
<td>Read/Write</td>
<td>Owner's agent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Read</td>
<td>All agents</td>
</tr>
<tr>
<td></td>
<td>Shared</td>
<td></td>
<td>Read/Write</td>
<td>Owner's agent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Temperate</td>
<td>Read</td>
<td>Authorized agent</td>
</tr>
<tr>
<td></td>
<td>Personal</td>
<td>Permanent</td>
<td>Read/Write</td>
<td>Owner's agent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Read</td>
<td>Authorized agent</td>
</tr>
<tr>
<td></td>
<td>Permanent</td>
<td></td>
<td>Read</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 5.1 WSS File System Descriptions

5.1.2 Native Program Invocation

In the WAVE grammar, a wave agent can execute arbitrary programs on the wave host by using a "?" operator. For example, A sector of code "6 ? sleep" will cause the agent to suspend execution and sleep for six seconds. The ability to invoke a native program is one of the most flexible and powerful functions WAVE provides. However, because the execution of external programs is not subject to the security restrictions of WSS, we consider this ability is very dangerous in the wave agent, especially when combined with file system access. An agent can perform a "customized attack" by carrying attack code, such as carry a virus to the target host, storing it on the target's disk and then later executing the code. Usually, only a trusted agent with very high privilege can access the native programs.

5.1.3 Network Communication

With the ability to call native programs, an agent may set-up separate communication channels other than the one inside the WAVE system. This function is very useful if we want to build an application-specific communication. One possibility is to let the agent
have the access to some of the RPC-based services. Therefore, if an agent needs resources on a machine without the agent platform, the agent can migrate close to that machine and interact with it using the RPC calls [Crystaliz 97]. However, the ability to set up separate communication channels could also create security holes. If a user wants to steal unauthorized information from a host and does not want the agent to carry the information out, because this activity could be detected by the AC and recorded by the system log file, he/she can access the file and send the information out through a separate channel, one not under the protection of the WSS. A malicious agent may also occupy the whole network bandwidth by continually transferring large messages to prevent other programs from doing proper network communication.

5.1.4 Peripherals

Peripherals such as printers, video cameras, speakers, microphones and monitors are usually only accessible to local users; however, it would be desirable if some of them could be used by remote users. The use of these devices can only be performed through the AA resource manager.

5.1.5 CPU and Memory

Unlike other resources, the CPU and the memory are the most basic resources an agent platform must provide. The question is how much can an agent consume them. An agent can perform a deny-of-service attack to the CPU and the memory by overwhelmingly using them; thus, other agents and even the local host cannot perform their legal operations. Memory allocation in the WSS is done through the wave interpreter's memory management functions. To restrict the use of memory, we can add a threshold to those functions. However, the problem of protecting the CPU time (cycles) cannot be solved using the WSS alone; it also requires better support from the operating system scheduler.
5.2 Rule-Based Access Control

Access control is decided according to the AA's local security policy and the access rules. When a wave agent executed on a wave interpreter makes a request for a local resource, the request is passed to the AA resource manager. The resource manager will check the security policy and the privilege of the agent. Whether the access will be granted depends on the access rules.

The operations in the Wave Secure System are classified into 4 groups as in the following:

- UO – Unrestricted Operations
- IRO – Identification Required Operations
- PRO – Privilege Required Operations
- MCO – Most Critical Operations

According to the agent categorization in the previous chapter, an anonymous agent can only perform UO; an identified agent can perform UO and IRO; and a trusted agent can perform all the operations in UO and IRO, plus, it can also perform PRO and MCO based on its privilege.

Let \( op \) denote a single operation and \( OP \) denote a set of operations. Furthermore, let \( i \) denote a given injector and \( I(op) \) be a set of injectors that can perform a set of actions \( op \).

Then, we have the following relation:

\[
I(UO) \supseteq I(IR) \supseteq I(PRO) \supseteq I(MCO)
\]

Let \( \text{auth}(obj) \) denote the fact that an object \( obj \)'s (a wave host or an injector) identification is tested and approved. And let \( s \) denote a given sender. Furthermore, \( \text{PrivilegeReq}(op) \) denotes the minimum privilege an agent must have to perform an \( op \).

And \( \text{PrivilegeHas}(i) \) denotes the privilege an injector \( i \) has.
For any operation op, we apply the following access control rules:

- If op ∈ UO
  Rule: No Rule
  Perform the operation

- If op ∈ IRO
  Rules:
  Injector identity based rule
  If auth(i), then grant the access to op.
  Injector and sender based rule
  If auth(s) and auth(i), then grant the access to op.

- If op ∈ PRO
  Rule: Privilege based rule
  If PrivilegeHas(i) ≥ PrivilegeReq(op), then grant the access to op.

- If op ∈ MCO
  Rule: Consistency based rule
  If PrivilegeHas(i) > PrivilegeReq(op), do: check with AC on the consistency, if true, then grant access to op. Consistency check in the MCO rule is performed by the wave code integrity check mechanism (explained in the next chapter).

### 5.3 Security Policies

The access control rules define sets of security checks that WSS should perform before the critical operations take place. How these rules affect individual operation is defined in the security policies. There are two aspects of the security policies in the WSS: local security policies and foreign security policies.
5.3.1 Local Security Policies

On a wave host, the security policy categorizes each operation into four different operation sets, UO, IRO, PRO and MCO and for those operations in PRO and MCO, a minimum required privilege is defined. All that information is stored in an AA's Local Policy Map (LPM), which is maintained by the AA Policy Manger. Different AAs in the same domain may have different local policies. However, because the privilege defined in the certificate is global inside a wave domain, an agent may perform different levels of operations in different wave hosts.

5.3.2 Foreign Security Policies

The domain AC manages the foreign security policies for agents traveling across the domain boundary. The foreign security policies always involve two different kinds of policies. One is the policy of issuing local agent passport, which is called the passport-issuing policy; another is the policy on issuing foreign agent visas, which is called the visa-issuing policy.

Recall that when a local agent wants to travel outside the local domain, the domain AC will issue the agent a passport. The AC will assign a new level of privilege to the agent (the injector) in the passport according to the passport-issuing policy, which describes how to map the privilege in a certificate to a passport. The privilege in the passport represents the trust that the whole domain gives to the principal, which is usually very difficult to define. Each domain can decide on their own passport issuing policy, for example, a simple solution can be that the home AC assigns all the passports the same privilege as their according certificates.

The visa-issuing policy describes how to map a privilege in a passport into a new one in a visa. It is often formed after an inter-domain agreement is made with the according foreign domain. Then the domain AC can assign a visa with a new privilege to an incoming agent from a contract domain, according to the privilege in its passport and the
visa-issuing policy. The value of the new privilege is usually lower than the one on the passport.

The policies also restrict the behaviors of a principal, for example, an injector cannot issue certificates to other principals, so there will be no certificate signed by an injector; an AA can only issue the certificate for an injector, but not for an AC; and only the AC can issue the passport and visa. These kinds of behavior restrictions are usually implicitly coded into the WSA program, and they cannot be edited by any principal.

5.4 System Logs

The system log files record all the security related operations performed by the principals in the WSS. The logs could force agents and other principals to be accountable for their actions. The idea is that if we failed to prevent an attack from happening, we should at least know “who” did it, so that we can take precautions to prevent the suspicious action from occurring again. There will be several system logs to record different actions. We use the following four kinds of logs to record the agent behaviors as well as the system management actions.

- A local injector log keeps a record of all the local injectors and their agent’s behavior, such as user log in, and change password.
- A local agent log keeps the record of all the behaviors of local agents.
- A foreign agent log keeps the record of all the behaviors of foreign agents.
- A policy log records behaviors related to the management of security policies.

We can also use the system log to record the usage of some specific resources to make these accesses accountable. Also, this auditing information can be used for account management. For example, we can restrict how frequently a principal can access some specific resources, or use the access information to charge the principal using the service.
Chapter 6

Protecting the Agent

When a wave agent executes on a wave host, all of its resources are exposed to its execution environment. If the wave host is malicious, the wave agent is at risk. A malicious wave host is usually defined as a wave-enabled host run by other (usually hostile) people or organizations, whose interest or purpose is against the agent's injector. In Chapter 3, we described the different attacks that a malicious host can perform on an agent. In this chapter, we will provide various techniques to protect the wave agent against these attacks. The major tasks are these: protecting the code integrity and execution process of the agent, protecting the data integrity, and preventing the host from masquerade attack.

The attacks can also occur during an agent migration, where an agent is sent as a network message between hosts. For example, an eavesdropper may steal a secret carried by an agent, and more seriously, tamper with it by modifying the content; an interceptor may impersonate an authorized receiver. These categories of attacks can be addressed by using secure communication protocol SSL, in which the message is encrypted with secret key known only to the two communicating parties; no other parties without knowing the secret key can reveal the meaning of the message. The problem of solving this kind of attack belongs to the category of protecting network communication and is beyond the scope of this thesis.
6.1 Two Different Approaches

To protect a vulnerable agent, many researchers have proposed solutions. Two different approaches are generally taken which reflect the different views of the researchers. One approach is proactive; that is, to prevent an agent from tampering, by hiding the execution process of the agent; the other approach is reactive, which is to detect when the agent is being tampered with.

Prevention
The basic idea of preventing an agent from tampering with is straightforward: to hide the entire execution process (thus the code and data) of the agent from the agent platform so that the attacker cannot reverse-engineer the actual code. In [Hohl 98], a method called the 'time limited black box' is proposed. In that method, the code of the agent is 'mixed up' or 'obfuscated' in a way that the host cannot understand the exact intention of the agent within a given time, of which the duration of the time is enough for the agent to finish its execution and 'flee' to other hosts. However, no way has yet been found to quantify the protection interval provided by the obfuscated function. Furthermore, no algorithm is provided to obfuscate a general agent.

A more sophisticated approach, found in [ST 98], suggests the idea that an agent's code can be encrypted functions. The method differentiates the functions and the program implements them, so that the environment executing the program has no idea of the actual function that the program wants to perform. Again, this method only applies to a very limited category of functions, rational function, and no solution for a general function has yet been found.

Detection
Execution tracing [Vigna 97] records the behaviors of an agent to a log file during its execution on each host. This protocol is designed in such a way that a chain of non-repudiatable-log (traces) will be formed. When the results of the agent are suspicious, the malicious host can be identified by examining the appropriate traces and trace
summaries. This technique is promising because it can spot the tampered host, but such a “bad-guy-hunt” process requires querying the whole traces chain, which involves all the hosts an agent had visited. This will cause a tremendous amount of overhead, which usually outweighs the benefit of the security.

Farmer et al. [Farmer 96] proposed an idea of using state-appraisal functions to protect an agent’s state. Based on the agent’s current state, a host uses a state-appraisal function to compute the privilege an agent needs. The state-appraisal function is designed in such a way that a damaged agent cannot acquire sufficient privileges to perform any damage to the host. This approach relies on the assumption that such state-appraisal functions can detect a tampered state, which may not be true in many circumstances, as the author suggests.

We believe there is no complete solution to prevent malicious host attacks by hiding the execution process of an agent, without the help of tamperproof hardware. However, by using cryptography, some pure software methods can be provided to detect if a wave agent has been tampered with, and therefore protect its code and data integrity.

6.2 Protecting Data Integrity

One major task of an agent is to process data. In an open environment the data carried by the agent may be subjected to attack by malicious hosts. A malicious host may spy out secret content of data or manipulate it. We would like to protect data integrity by differentiating the characteristics of data and treat them in different ways. We categorize data in a wave agent into three categories, read only data, append only data, and destination specified data.
6.2.1 Read Only Data

An agent may often carry data that is constant after the agent acquired it. For example, an information propagation agent may be used to deliver a contract between different contractors. A malicious host may try to attack the agent by changing the content of the contract to mislead the other hosts who read the altered contract. In this case, we want to make the contract data read-only; therefore, no principal other than the contract creator itself can change the content. To satisfy this purpose, Read Only Data (ROD) is signed by the data owner's private key, so that any illegal change to the ROD can be detected by checking the signature. During its migration, an agent may also obtain data through bypassing hosts. The host can encapsulate the data into a ROD, and other hosts after it, in the agent's migration path, cannot manipulate the data without being detected. Figure 6.1 shows a pseudo-code of the ROD.

The content of the ROD is a vector of data objects with arbitrary types. When a principal creates a ROD to encapsulate a data object, the hash value of the object is calculated and the signature over the hash value by the creator's private key is obtained. The hash value is calculated using the SHA algorithm and the signature uses the RSA algorithm. The formula is as follows:

\[ \text{Signature} = \text{RSA\_sign( SHA\_hash(data) \), privateKey} \]

The data object, signature and ROD owner's identity are stored in the ROD. When the agent wants to read the data object from the ROD, the signature in the ROD must be verified first. The verification uses the ROD owner's public key to decrypt the signature. The result of the decryption should equal the hash value of the data object. The verification condition can be expressed as below:

\[ \text{SHA\_hash(dataObj) == RSA\_decrypt(signature, publicKey)} \]
Because the private key of a ROD owner is not forgeable, if a malicious host wants to alter the content of a ROD without being detected, it must alter the data in such a way that the hash value of the new data object remains the same as that of the old data object. However, this is not feasible because as we discussed in Chapter 2, the SHA hash function is collision-resistant, which means, given a hash value of a message, it is infeasible to calculate another message that produces the same hash value.

```
Class ReadOnlyData {
    OwnerID oid; // the identification of the ROD owner
    Vector dataObj; // the read only data
    byte[] signature; // the signature of the owner

    ReadOnlyData(Vector obj, OwnerID id, PrivateKey key) {
        oid = id;
        dataObj = obj;
        signature = RSA_Sign(SHA_hash(dataObj), key);
    }

    byte[] getSignature() {
        return signature;
    }

    Vector getData() {
        // Verify the owner's signature on the dataObj
        // by using owner's public key
        // If it can be verified, return dataObj
        // Otherwise return null
    }
}
```

**Figure 6.1 The Read Only Data**

### 6.2.2 Append Only Data

A common task that an agent has to perform is collecting necessary information from other hosts on the Internet. A widely used example is a price shopper agent, which collects air ticket price information among travel agencies' hosts for its injector. The injector of the agent will make a purchase decision based on the information collected.
Because these agencies are usually competing with each other, they want to take advantage of the competition by spying out other agencies' prices and adjusting their own prices. One may change or even remove other agencies' existing offers carried by the agent. To avoid these attacks, the price information provided by a travel agency should be unchangeable and irremovable after the wave agent collects it. The ROD provides a method so that the data carried by an agent is unchangeable; however, a malicious principal can simply remove the entire ROD without being detected, since the agent does not know it has ever existed. In this case, we can use *append only data* (AOD) to make the data irremovable.

The AOD is to collect a vector of RODs. The collection is in an append-only fashion so that we can only append new ROD data into the AOD but cannot remove it. If there is any data removed from an AOD, it can always be detected. The pseudo-code of the AOD is given in Figure 6.2. An AOD has a vector of ROD `rodData` and a byte array to store the accumulated hash value `hashValue`. When a new AOD is created, an initial value (a random number) must be provided. The initial value of `hashValue` variable is the SHA hash value of the `initValue`. When a ROD `data` is added into the AOD, a new `hashValue` is calculated using the function:

\[
\text{hashValue} = \text{SHA}\_\text{hash}(\text{hashValue} + \text{data}.\text{getSignature()})
\]

The function `data.getSignature()` gets the signature of the ROD `data`. When the agent finishes the execution and brings back the AOD, the injector will recalculate the accumulated hash value. The calculation will need the initial value of the AOD, which has to be in the same order as the RODs were added. If the value equals the `hashValue` of AOD, it means the AOD is complete and no data is maliciously removed. The integrity of each ROD can be verified by its signature. If a malicious principal tries to delete the RODs inside an AOD, it must forge the `hashValue` for the AOD to match the change, which prevents it from being detected. However, without the initial value of the AOD, the recalculation of the `hashValue` will not be possible.
In the method described above, the data carried by the AOD is readable to all of the principals. If we want to keep data secret, we can simply use the agent’s public key to encrypt every ROD when added into the AOD. In this case, no principals except the injector (the appropriate private key owner) can read the data because no one has the knowledge of the private key.

```java
Class AppendOnlyData {
    Vector rodData; // a vector of read only data
    byte[] hashValue;

    AppendOnlyData(byte[] initValue) {
        rodData = new Vector();
        hashValue = SHA_hash(initValue);
    }

    void storeData(ReadOnlyData data) {
        rodData.addElement(data);
        hashValue = SHA_hash(hash Value + data.getSignature());
    }

    boolean verify(int initValue) {
        byte[] checkHashValue;
        checkHashValue = SHA_hash(initValue);
        loop {
            checkHashValue = SHA_hash(checkHashValue + data.getSignature());
            } until all the rodData are processed;
        if (checkHashValue == hashValue)
            return true;
        return false;
    }

    // Other necessary code
}

Figure 6.2 The Append Only Data
```

6.2.3 Destination Specified Data

An agent may carry data destined to a specific host. For example, an agent carries personal mail to a specific user. We do not want other principals in the system to read the content. We can use the Destination Specified Data (DSD) to encapsulate the mail.
The DSD data will be encrypted by the receiver’s public key so that no principals without the appropriate private key can understand the content or tamper the data without being detected.

Figure 6.3 gives the pseudo-code of the DSD. When a new DSD object is created, the data is encrypted using the destination’s public key:

\[ \text{data} = \text{RSA}_\text{encrypt}(\text{plaintext}, \text{destinationPublicKey}) \]

When a principal wants to read the encapsulated data from a DSD, it must provide the destination’s private key and use the read() method, where the key is used to decrypt the data.

```java
Class DestinationSpecifiedData {
    OwnerID oid; // the identification of the ROD owner
    byte[] data;

    DestinationSpecifiedData(Object plaintext, OwnerID id, DestinationPublicKey key) {
        oid = id;
        data = RSA_encrypt(plaintext, key);
    }

    Object read(OwnerPrivateKey key) {
        return RSA_decrypt(data, key);
    }

    // Other necessary code
}
```

**Figure 6.3 The Destination Specified Data**

### 6.3 Protecting Code Integrity

In chapter three, we described that the signature binds the agent with the identity of its injector, therefore, it can be used to verify a wave agent’s identity. Also, the signature
can guarantee that the signed document is unalterable (Chapter 2.2.3). However, because
the signature of the agent is obtained by signing only over the constant part of the agent,
such as the agent ID, injector ID and original AA name, it can only provide a guarantee
of the integrity to these constant data. During the agent's dynamic execution on different
wave hosts, the agent's body (code, data, state) is constantly changing, and it is difficult
to determine whether the change is due to proper execution or malicious attack.
Therefore, as soon as the wave agent travels out of its home wave host, it becomes
malicious. There is no guarantee of the code or data integrity. A malicious host can
simply add malicious behaviors to the agent's code without affecting the agent's
signature. The malicious behaviors added will take the advantage of the agent's injector
since it is counted as the agent's legal behavior. The host may also remove some of the
existing behaviors from the agent; as a result it will no longer function as supposed.

To make a dynamic principal accountable is not an easy task in an open network
environment. Fortunately, we can take advantage of the WAVE grammar. Although the
wave agent is dynamic, the changes made to the agent are to some extent predictable. If
we have the original copy of the code of an agent, we can test if other wave code is
derived from that original code.

6.3.1 Wave Agent Registration

Every agent running inside the WSS domain will have a registration entry on the local
AC, in which its 'original' and 'correct' copy is recorded. For an agent originating
locally, a complete copy of the agent will be taken and stored into its home AC's Agent
Registration Table (ART) when the injector injects it. When an agent migrates from a
foreign domain, the AC will take a snap shot of it and store the information into the ART,
when the agent asks the domain AC to apply for a visa before it enters the domain. An
example ART table is described in Table 6.1, which has three registered agents running
in the domain. The record will be removed from the ART after the agent finishes its
execution. The enter time is the time when the agent enters the domain and the origin
field is the address of the agent's home host. Because a foreign agent always carries only
a passport, which is issued by its home domain AC instead of its injector's certificate, the origin of a foreign agent is always its home AC. Every agent currently running in a wave domain must have a registration entry in the domain AC's ART, otherwise, the agent is considered malicious, and its execution will be refused.

<table>
<thead>
<tr>
<th>Agent SN</th>
<th>Injector Name</th>
<th>Enter Time</th>
<th>Origin</th>
<th>Agent Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>pfu</td>
<td>08:00 May, 5th 2001</td>
<td>AA1.Local</td>
<td>The original wave code</td>
</tr>
<tr>
<td>14</td>
<td>mphan</td>
<td>09:30 May, 5th 2001</td>
<td>AA2.Local</td>
<td>The original wave code</td>
</tr>
<tr>
<td>7</td>
<td>yuyin</td>
<td>12:00 May, 5th 2001</td>
<td>AC.cs.ubc.ca</td>
<td>The original wave code</td>
</tr>
</tbody>
</table>

Table 6.1 An Example of an Agent Registration Table

6.3.2 Code Integrity Check

The wave agent (or a wave program) is actually a sequential and parallel or unordered composition of space-time actions. These actions are encoded as a string and separated by delimiters (","","" or ";"). All the actions in the first period-separated part together is called head, the rest is called tail. When a wave agent is executed on a wave interpreter, its head is separated and executed. The tail of the agent may stay local or migrate to other wave host depending on the result of the execution of the head. Also, the same process is applied to the tail recursively until the tail is empty. (For a more detailed description of WAVE grammar, please refer to Appendix A.) The execution process of a wave agent indicates that once an action is coded into a wave agent, it cannot be modified but can only be removed as head separation under a legal interpretation. All the actions of a wave agent are the subset of its initial actions when it was injected and the order of the actions cannot be changed. We can use a naïve algorithm to compare the existing agent with its original copy in the local AC's ART and determine whether the actions of the existing agent is a subset of its original actions. Figure 6.4 is the pseudo-code of the code integrity check. When a wave agent with sequence number agentSN, whose current code agentCode needs to be checked, we first obtain its original code from the domain AC's ART. If the agent has a registration entry in the ART and it is still active, we use a
method `contains()` to examine if the current `agentCode` is a sub-string of its original registered code `agentOriginalCode`. If it is, the integrity check is passed, otherwise, the check is failed and the agent is abandoned.

```java
boolean agentCodeIntegrityCheck(int agentSN, String agentCode) {
    String agentOriginalCode =
        agentRegistrationTable.getAgentbySN(agentSN);

    if (agentOriginalCode == null) {
        //no such agent registered in the domain
        return false;
    }

    if (agentRegistrationTable.notActive(agentSN)) {
        //the agent already finished execution
        return false;
    }

    if (contains(agentOriginalCode, agentCode)) {
        //the agentCode is a sub-string of its original code
        return true;
    }

    return false;
}
```

**Figure 6.4 The Code Integrity Check**

### 6.4 Agent Signature Replay

Because the signature signs only over the constant part of an agent, a malicious injector can reuse the signature by assigning his/her agent the same constant part described in the signature. The agent then claims itself as being injected by the signature owner to steal the identity and credit of the owner to obtain the victim's privilege. We describe this kind of attack as the agent signature replay. To prevent such an attack, we should make the signature of the agent one-time-use only. To this end, we added two attributes to a wave agent’s environment and included them into the signature. The two attributes are
Injection Time (IT) and Time To Live (TTL). IT records the time the agent was injected and TTL indicates the lifetime of the agent. In this case, the signature’s effective time is limited to TTL. However, before a signature expires, a malicious agent can still replay it. To solve this problem, we will not remove the agent registration record immediately after the agent finishes execution, we will instead keep the entry in the ART until the signature of the agent expires. Furthermore, we add another attribute to the ART to indicate whether the agent is still running (active) or not. In this case, if another agent replays the signature before the signature expires, it can also be detected by the code integrity check. Table 6.2 gives an example of the refined ART.

Table 6.2 A Refined ART to Prevent Signature Replay

<table>
<thead>
<tr>
<th>Agent SN</th>
<th>TTL (min)</th>
<th>Injector Name</th>
<th>Enter Time</th>
<th>Origin</th>
<th>Active</th>
<th>Agent Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>60</td>
<td>Pfu</td>
<td>08:00 5/5/2001</td>
<td>AA1.Local</td>
<td>Yes</td>
<td>The original wave code</td>
</tr>
<tr>
<td>14</td>
<td>60</td>
<td>Mphan</td>
<td>09:30 5/5/2001</td>
<td>AA2.Local</td>
<td>No</td>
<td>The original wave code</td>
</tr>
<tr>
<td>7</td>
<td>60</td>
<td>Yuyin</td>
<td>12:00 5/5/2001</td>
<td>AC.cs.ubc.ca</td>
<td>Yes</td>
<td>The original wave code</td>
</tr>
</tbody>
</table>

6.5 Discussion

The cryptography, especially public key cryptography, provides powerful tools to protect information on open environments such as the Internet, however this kind of protection works only on constant data, like the passive data package transmitted between parties. For an agent whose state is highly dynamic that can be executed and changed on any agent platform, a naïve application of public key cryptography will not work. We have to differentiate the content of an agent according to time or functionality and treat them separately. We can encrypt or sign the immutable part of the agent so that they become tamperproof. For example, the read only data will remain constant once the agent has been injected, so that the creator can sign it and its integrity can be protected. For the
mutable part of the agent, we have to recode the original and unaltered version of it and perform the integrity check.

There is one more security concern not yet addressed in this chapter. If a wave agent was attacked somewhere along its migration path, how can we determine which host performed the attack? We can use the AOD to solve the question. We use the AOD as a carry-alone and append-only log for a wave agent, and record every action performed by a host to the agent. In this way, the behavior of a host to an agent is accountable in the log. If there is any malicious behavior by the agent, it can be found in the AOD log. However, this log can be a large overhead for a wave agent. If the agent has to pass many hosts, the log maybe too long to be practical.
Chapter 7

Implementation Issues & Application

7.1 Choice of Programming Language

The WAVE concept was invented in the early 90's and its first implementation can be dated back to 1992 [Borst 92]. At that time, C was the dominating programming language for network communication; modern programming languages and concepts such as Java and CORBA for heterogeneous and distributed programming environments were not mature or even existing yet. As a result, C is chosen as the programming language for implementing WAVE. As for developing the WSS, one may assume using C should be the most natural decision; however, the C language has many drawbacks for mobile agent systems and against many of our system design criteria. For example, C is not object oriented, which makes it hard to maintain and scale up; it is platform dependent, and not portable to different operating systems; also, it does not provide any security facilities at language level. The Java programming language [GJS 96], on the other hand, provides all the desired capabilities C lacks. Java is object oriented and type safe, which is more resistant to security threats than C. Also, Java is an interpreting language and the Java program is executed on Java Virtual Machines, which makes it extremely portable. Java provides language level security support [Dageforde] and
provides implementation of almost all major cryptograph primitives [JCE]. Furthermore, the serialization concept and class loader in Java make agent migration easier to implement. Because of all these benefits, most of the mobile agent systems nowadays are implemented in Java, such as AgentSpace, Aglet, Ajanta, Concordia and Mole and so forth. The reimplementation of WAVE using Java is also under consideration. We believe that using Java to implement WSA will best fit the future requirements of WAVE and Java’s built-in security features will greatly reduce the time for implementation. Also using a different programming language will force us to keep the design of the WSA more general, and to separate implementation consideration from the original code of WAVE. Therefore, the resulting system (WSA) will be easily integrated into other mobile agent systems.

7.2 A Secure Protocol for Java RMI Interfaces

In the WSS, ACs and AAs provide services to other principals in the system using interactions through network, such as agent registration, certificate distribution, and so on. These interactions are typically in a client/server style, in which services are requested by issuing requests and results are reported by responses. An RPC style communication mechanism thus fits. For the WSS implementation, we use Java Remote Method Invocation (RMI) [RMI] facilities to implement these communications. By using Java RMI, ACs and AAs register their services as remote invocable methods (RMI services). If any process needs to use these services, it must know the services’ interface, through which it may obtain a reference to these services. Once the process obtains the reference, it can use any service provided through the interface. In this case, no matter where the service process resides, the caller will be able to call the (remote) method in the same way as if calling a local method. Figure 7.1 is an example of how a caller may obtain a reference (interface) to a remote service and invoke a remote method through it. In the example, //goodearth.cs.ubc.ca/GUI is the name of the interface. We can use the Naming.lookup() method of RMI to get a reference to the above interface and in the example we call it myInterface. Through this interface, we can use the services, such as
the `inject()` service which can inject a wave agent by using `myInterface.inject(waveAgent)`. Notice that we call the remote method `inject()` the same as when calling a local method.

Java RMI is very convenient for distributed programming because the network between the caller and callee is transparent and the programmer need not care about the details of communication. However, as everything has two sides, the transparency and convenience provided by Java RMI brings security hole as well. After an RMI service is registered, it becomes available to the public. Any process that knows the service interface will be able to invoke the remote method. An ordinary user can request the use of these remote services only after going through an authentication process; so that appropriate privilege is identified and the access to these services is controlled according to its identity. However, an attacker may skip the authentication process and directly use any of the services provided if she/he only knows the service’s interface. For example, if the attacker knows the interfaces of the services provided to an AA administrator, she/he will be able to perform all the actions that an administrator can do, for example, request a certificate with the highest privilege; or delete a user, and so forth. This could become a disaster in a real life situation. To prevent this kind of attack, we have to authenticate every single remote invocation.

```java
String inject (String waveAgent) {
    AAGuIlnterface myInterface; // The interface to remote method
    // The address (name) of the remote service
    String serverAdd = new String("//" + "goodearth.cs.ubc.ca/GUI");
    try {
        // Obtain a reference to the remote service
        myInterface = (AAGuIlnterface) Naming.lookup(serverAdd);
        // Invoke remote method through the interface
        return myInterface.inject(waveAgent);
    } catch (Exception e) {
        // Do the necessary exception handling
    }
}
```

**Figure 7.1 An Example of RMI Interface Lookup**

67
7.2.1 A TimeStamped Tamperproof RMI Interface

To make a Java RMI service inaccessible to unauthorized users on the Internet, we need to add the identification information of the caller. This can be achieved by requiring the caller to sign the parameters supplied in the interface. The signature can be used to verify the identity of the caller and at the same time it also guarantees that all the parameters are not tampered with. To prevent replay attack of the signature, a time stamp is added to the interface, so that every signature in a remote invocation can be used only once. The time in the time stamp need not be synchronized with the server's clock. A server will set up a timer for every client, which records the client's latest calling time. The timer has a value of 0 if the owner client has never used the services before. Every new call request from the client must have a time stamp larger than its previous calling time recorded. In this case, a time stamp is used as a non-decreasing sequence number for the client/server pair and we do not need any special mechanism to setup such a sequence number between them. The use of time stamps makes the signature time irreversible and non-reusable. By adding the authentication information, every Java RMI interface has three more parameters: the caller name, the time stamp and the signature. A time stamped tamperproof interface allows the injection of a wave agent named waveAgent into the WSS will be as following:

\[
\text{inject(String waveAgent, String callerName, Date timestamp, byte[] signature)}
\]

Where the signature is calculated using the following function:

\[
\text{signature} = \text{RSA_sign(}SHA\text{\_hash(waveAgent + callerName + timestamp)})
\]

When the server receives the call, it will first verify the signature by verify the following Boolean function:

\[
\text{RSA\_decrypt(signature, publicKey) == SHA\_hash(waveAgent + callerName + timestamp)}
\]

If the signature is correct, the server will then verify the time stamp and make sure it is larger than the time of the previous call from the same caller and then update the timer of the caller.
If the signature is correct, the server will then verify the time stamp and make sure it is larger than the time of the previous call from the same caller and then update the timer of the caller.

\[
\text{if (getTimer(callerName) < timestamp) } \{
    \text{updateTimer(callerName, timestamp);} \\
    \text{//perform proper action}
\}
\]

\[
\text{else } \{
    \text{//call is refused}
\}
\]

![Diagram of Wave Host, Authentication Agent, Resource Manager, Wave Interpreter, OS Interface Processor, and Local Resources]

**Figure 7.2** Interface Between Wave Interpreter and the Authentication Agent

### 7.3 The Interface Between Wave Interpreter and WSA

The WSS protects the wave host by applying security rules and policies on the operations that a wave agent can perform. These operations are performed through the wave OS Interface Processor. To restrict these operations we have to intercept the wave agent
interpretation process before the operations get executed, and authenticate the agent who initiates the operations. We need to provide an interface between the wave interpreter's C process, which performs the operation and the WSS's Java process, which has the identification information of the agent. Socket is used to carry out the communication between these two processes. Because the two processes always reside on the same host, we use local loop back IP address 127.0.0.1, which will free us from worrying about the security of the network communication between the C and the Java processes. Before an agent carries out an operation, the wave interpretation process requests permission from the WSA by sending it an operationRequest package, which contains the agent name and the operation name. When the AA Resource Manager receives the request, it will check the policy file and identify the category of the operation. According to the security requirement of the operation, a proper security check is performed, such as checking the privilege of the agent. If the check is passed, the AA Resource Manager will return an OperationPermitted package, and the operation can be legally performed. Otherwise, the AA Resource Manager will return an OperationRefused package, and the operation is refused. Figure 7.2 shows the process described above.

### 7.4 Overhead Analysis

The security mechanisms inevitably introduce overhead to the original Wave system, but this trade-off is only minimal. In this section, we discuss and measure the three major overhead factors introduced in WSS; they are the communication overhead, the authentication overhead, and the access control overhead. Latency is measured using machines with the following hardware and software configuration:

<table>
<thead>
<tr>
<th>Machine</th>
<th>CPU</th>
<th>Memory</th>
<th>Operating System</th>
<th>Java VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 1</td>
<td>Intel PIII 800MHz</td>
<td>256 MB</td>
<td>Linux 7.1.</td>
<td>1.3.1</td>
</tr>
<tr>
<td>Server 2</td>
<td>sparc Ultra-60</td>
<td>1024 MB</td>
<td>SunOS 5.8</td>
<td>1.3.1</td>
</tr>
</tbody>
</table>

**Table 7.1 Server Configurations**
The data calculated below is the average result of ten repeated experiments. Server 2 was used by multiple users, and the experiment was conducted at night when the workload of the server was light.

**Communication Overhead**

Every wave agent carries a signature and a certificate. They become the communication overhead as they attach to the wave agents, when sent between wave hosts. The size of the signature is constant, given the size of the signing key, when we use the SHA-1 with RSA algorithm. In implementation, the key size of 512 bits and the message digest of 160 bits was used for the signature. Therefore, the size of the signature will be 64 bytes. Given the fact that most of the current on-line banking systems use only 128 bits key, the 512 bits key that we are using can be considered very secure, for now. The size of the certificate is not constant; it varies with the length of the issuer and holder’s name. Normally, the length of the name will not exceed 40 characters, in which case the body of the certificate is 715 bytes, plus the 64 bytes signature, which causes the total overhead to be $64 + 715 = 779$ bytes.

**Authentication Overhead**

The authentication process also adds new overhead to the wave system. The wave agent cannot start execution immediately when it arrives at a wave host, because the wave host must identify the wave agent by verifying the signature that the agent carries. This authentication overhead includes the amount of time used to verify the signature and to verify the certificate. Given the average certificate size of 779 bytes (calculated from above), Server 1 takes 20 milliseconds to verify the certificate; and Server 2 takes 35 milliseconds approximately. To verify the signature of the wave agent, Server 1 takes 10 milliseconds and Server 2 takes 20 milliseconds approximately. In total, the authentication process of a wave agent costs 30 milliseconds of overhead for Server 1 and 55 milliseconds of overhead for Server 2.
Access control overhead

When a wave agent tries to access controlled resources on the wave host, that is to execute a program, the WSS must obtain the identification information of the wave agent and test its privilege according to the security rules first. As described in Chapter 7.3, this process involves a local UDP package exchange and the search of the agent’s identity and security policies. The time required to verify the access privilege is 10 milliseconds for Server 1 and 18 milliseconds for Server 2 approximately.

Based on the performance evaluations conducted on various machines, we determined the overheads added to the wave system are minimal. The security techniques we introduce to WAVE will not disadvantage the scalable issues. To conclude, we claim that the security techniques will only bring minimal overhead to the system.

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Communication</th>
<th>Authentication</th>
<th>Access control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 1</td>
<td>779 bytes</td>
<td>30 millisecond</td>
<td>10 millisecond</td>
</tr>
<tr>
<td>Server 2</td>
<td>779 bytes</td>
<td>55 millisecond</td>
<td>18 millisecond</td>
</tr>
</tbody>
</table>

Table 7.2 Overhead Measurements

7.5 An Application

In order to verify the programmability and security of the WSS we want to explore some of the possible end-user applications of it. That is the motivation behind our intention to build a distributed file sharing system called wavetella, which implements a Gnutella [GNUT] like peer-to-peer file sharing system. Mike White had the idea and implemented the first version of the wavetella for his CPSC527 project [White 01], which is based on the original WAVE system. We enhanced his system and ported it to the WSS.
7.5.1 Wavetella

The wavetella is a distributed, peer-to-peer file share system. There is no central control over the file servers inside the system and the servers are peers of one another. A file server is simply an extension of a wave host. In addition to the basic agent hosting capability, each file server selectively shares files with other servers and provides controlled access to these files. Each file server also provides an entry point, through which a user can access the whole wavetella file system and need not know the topology of it. The WSS is used as a middleware that the wavetella user will not be aware of its existing. Figure 7.3 gives a brief description of the structure of the Wavetella with three layers:

![Diagram of Wavetella structure](image)

**Figure 7.3** Structure of the Wavetella
File Transfer Layer
The File Transfer layer provides a file search and retrieval interface to the users. It also provides the entry points for the users, where they can search and download files. It will take the user’s search input and acquire search results from the lower layer. After the search is hit, it also provides the interface for users to download the selected file.

WSS Layer
This provides the abstract network interface to the File Transfer Layer. The file servers are organized and managed as nodes of the wave KN. The WSS is used to perform the distributed file-searching algorithm. It will take the key word string from the File Transfer layer, and use a wave agent to search the key word on its knowledge network. When a search is hit, the result will return to the top layer.

File Server Layer
The file servers are the physical media that store the files. They are also the logical nodes of the WSS KN. A file server provides a database, which has a specific interface that publishes the files that the server wishes to share. The tuples in the database describe the name, size and access requirement of the file. The file names are searchable by the user’s agent, which will then return the search results to the user. A file server provides searchDB facility, which is used by the wave agent to search the database.

7.5.2 Security of the Wavetella
In a real world environment, the file servers in the wavetella may want to provide users with differentiate access abilities to their files. For example, the wavetella can be used in a company to share the files between different departments; each department may have one or more file servers. In one scenario, one may only want to share its annual reports with employees from the same department, disallowing no employees from other departments to read them. In another, one may not want to share a detailed budget file with others, and only the head of the department may read it. The president of the company may want to have the ability to read all the files from all the departments. In
these situations, authentication of the wave agent, which will search the file servers for the user, becomes critical in the system.

A wave agent needs to execute a \textit{searchDB} program to search the database on a file server. When a wave agent tries to execute the \textit{searchDB} operation, it sends an \textit{OperationRequest} to the local AA Resource Manager. The AA Resource Manager examines the privilege of the agent, and returns the approval or refusal of the agent to execute the program. If the access is approved, the agent will search the database for its keywords and bring back the result. Otherwise, the agent will assume nothing is found and always return an empty result. When the user finds the file she/he is interested in, she/he can use \textit{fileServer} daemon to download the file. However, the download operation is also subject to a privilege check, in which the user must have a privilege no less than that described in the fileserver's database.
Chapter 8

Conclusions

Unlike traditional mobile intelligent agent systems where a small set of APIs are provided to support limited agent (code) mobility capabilities, the novel agent system, WAVE, offers a complete, high-level language that is, despite its fairly simple syntax, rich in semantics and mechanisms for integration, control and management for rapid, effective realization of seamless, cooperative distributed applications. However, like many other mobile agent systems, the lack of security in WAVE highly restricts its scope of applications. In this thesis, we propose a security architecture and implement a security system based on this architecture for the Wave Secure System. This security system makes use of a rich security model that gives identification to each principal user and provides access control to a very fine level of granularity. The security system also provides the methods for detecting if the behavior or data of a wave agent has been tampered with. Although the security architecture was developed for WAVE, its applicability can be generally suited to any mobile intelligent system.

8.1 Contributions

In this thesis, we give the framework of a secure architecture for WAVE, along with partial implementation of the system. The design is general, so that it can be easily integrated into other mobile agent systems.
We introduced and implemented the Wave Certification Infrastructure (WCI) to distribute and authenticate the identities of the wave agents and other principals within the system. We analyzed both the totally centralized and totally decentralized methods distributing trust and belief in the system; we also came up with a design that considers the trade-offs between the two extremes, and apply only their advantageous characteristics.

We designed the AA Resource Manager and used it to protect the wave host. Every operation performed by a wave agent is subject to security checks by the AA Resource Manager. Access control of these operations is defined by its security rules and policies.

Three different kinds of data types are introduced to protect the wave agent’s data integrity. A Read Only Data is used to ensure that the information stored in the wave agent is immutable. An Append Only Data is used to allow a wave agent to collect and store data during its migration and these data are immutable and irremovable. A Destination Specified Data allows the wave agent to carry some information that is readable only to specific receivers.

A Code Integrity Check mechanism is proposed to protect the wave agent’s code integrity. We took the advantageous of the WAVE grammar, and used a naive string comparison method.

A time-stamped, tamperproof communication interface is defined for the Java RMI interaction between principals, so that only authorized users can use the services provided by the ACs and the AAs.

In order to test the design of the WSS, especially its secure capabilities, we built a distributed file sharing system application named wavetella. The wavetella uses the WSS as a middleware, which provides an abstract network interface and is
used to locate desired files. Access to shared files is guarded by the security functions of the WSS; therefore it can differentiate access privileges according to user identities.

8.2 Future Work

For the WSS described in the thesis, not all the features have been implemented, such as the three data types used to protect wave agent data integrity. This is partly due to the fact that the WSS is too comprehensive and large to complete in a single master's thesis. Another reason is the limitation of functionalities of the original WAVE system. We would like to continue to implement all of the features in the WSS. Also we would like to enhance the WAVE system as well, for example, make WAVE support different data types, directories, and libraries. A most attractive possibility for enhancing WAVE is to reimplement it using Java, which, we believe, will solve all the problems at once. We propose to upgrade the current WAVE to its second generation, which is called Actigen [VF 01c].

The methods we proposed to protect the wave agent are obtained from the "detection" point of view, where detection means we can detect the altered part of the agent after a wave agent has been tampered with. Although we believe this is the most feasible way to protect the wave agent so far, a more proactive way would be more desirable. We would like to prevent attacks against the wave agent, so that both the code and data integrity protection problems can be solved at the same time.

Passport and visa issuing for the agent travels across domain boundaries is according to the foreign security policy of the relating domain ACs. The current design of these policies is fixed after an off-line policy negotiation between the two ACs. A more flexible design of visa issuing method would be interesting to consider. One possible way, is to have a wave agent negotiate with the foreign AC and obtain its privilege from the foreign domain.
In order to provide greater assurance of the security of the security mechanisms that we provided in this thesis, we would like to define the formal (usually mathematical) models which would describe them. With these models, it would be possible to give each mechanism a mathematically sound proof of its security. However, these models, we believe, are difficult to obtain if not impossible. We also would like to prove the effectiveness and correctness of the trust and belief relationship created by the WCI, to win more theoretical support.

To fully examine the WSS, one application is far from enough; we would like to port other existing WAVE applications to the WSS and implement new applications, so that a larger spectrum of applications can be analyzed and their performance and security can be compared.
Bibliography


[PGP] Website at URL: http://www.pgp.com


[SSL] OpenSSL, URL: http://www.openssl.org/


[X208] ITU-T X.208, URL: http://www.itu.int/itudoc/itu-t/rec/x/x200-499/x208.html


Appendix A

A Brief Description of WAVE Grammar

In Figure A.1, the syntax and primitives in the current WAVE language definition are described. Words in small letters denote syntactic categories. Brackets represent zero or more repetitions with a corresponding delimiter; square brackets embrace an optional construct; and a vertical bar separates alternatives.

```
wave  ->  {{move, .}}
move  ->  unit act unit | [rule] (wave)
rule  ->  SEQUENCE|OR_SEQUENCE|
         OR_PARALLEL|AND SEQUENTIAL|
         AND_PARALLEL|REPEAT|
         WAIT|CREATE
unit  ->  {string; }IN{letter_or_digit}|
         F{letter_or_digit} }environmental_variable
environmental_variable  ->  CONTENT|ADDRESS|
                         PREDECESSOR|LINK|
                         LINK_SIGN|TERMINAL
act   ->  #@|~|~/|==|=|<=|==|=|+|*|**||%|&|\|:|:=|?|!
```

Figure A.1 Wave Grammar

A Wave program or agent, which is called `wave` in a grammar description, can be seen as a sequence of moves separated by dot delimiters. Each move can be an `act` over two `unites` or other waves in parenthesis (with rules), which allows a recursive nesting of a
wave. In the sequence of moves of a wave, the first one is called head and the rest are called the tail. The interpretation process of the wave is always performed on the head. The tail of the wave may be terminated, moved to another KN node, or remain at the current node due to the result of its head. If the tail will continue execution, again its head is interpreted which forms a recursive interpretation of the wave. During the execution, the wave loses its worked part and therefore shrinks in size.

The data components in a wave are defined as a unit, which can be data values or three kinds of variables. Data values are represented as linear vectors of character strings separated by semicolons. The three kinds of variables are frontal variable, nodal variable and environment variable. Frontal variable starts with a capital letter F and the scope of its value is only within the wave. Nodal variables start with a capital letter N and belong to a KN node. Environment variables are used for system environment propose. CONTENT is used to store KN node content; ADDRESS stores KN node addresses; PREDECESSOR is the address of the predecessor node, which the wave just visited; LINK is the link content of the link that the wave just passed; LINK_SIGN is the direction of the link; and TERMINAL is used for user input/output.

A wave can perform operations over units by act. Usually, these operations will have two operand units and the result of the act, if any, will be stored in the first operand. Also, as a general rule, either operand can be empty, which will trigger a special semantics, depending on the act. Now, WAVE provides operations for wave migration, logical and mathematical operations as well as control acts. The hop operation "#" is the major navigation act. The first operand of hop specifies the content of the links over which to move and the second operand is either the address or the content of the node to which the propagation will lead. If "+" or "-" is used directly with a link, it specifies the direction of the link. An @ operation will cause the wave to jump to the node specified by the second operant of @. We will not introduce logical and mathematical acts in detail, as their meanings can be self explained by their names. The last two acts "?" and "!") are control operations. "?" is used to activate any sort of external executable from a
local operating system. "!" is used to stop the current wave in the goal node and returns an echo.

A wave in parenthesis may be optionally preceded by a control rule. The rule splits the wave into a set of individual branches and organizes their coordinated parallel or sequential development. As we have introduced, the execution of a wave is performed on its head. The head of a wave can be composed of several sectors separated by commas. For each sector, the wave splits into a new, independent instance. The sector's string becomes the head of the new program and is followed by the original common tail. The rules then enforce control constraints on these separated wave programs.

- **SEQUENCE, OR_SEQUENCE and AND_SEQUENTIAL**
The branches of the wave's head are developed sequentially. The wave program obtained from one branch has to terminate before the next branch is activated. The "or" and "and" rule add additional conditions on the echoes returned from the terminated waves of the branches.

- **AND_PARALLEL, OR_PARALLEL**
The branches of the wave's head are developed in parallel. The "or" and "and" rules have the same functionalities as that in sequence rules.

- **REPEAT**
Under this rule, the wave will replicate itself. It is used to implement loops.

- **WAIT**
It will wait for the complete termination of all branches. It is used for synchronization.

- **CREATE**
It creates new nodes or links of the Knowledge Network.
## Appendix B

### A List of the Existing Mobile Agent Systems

<table>
<thead>
<tr>
<th>System</th>
<th>Organization</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ajanta</td>
<td>Univ. of Minnesota</td>
<td><a href="http://www.cs.umn.edu/Ajanta/">http://www.cs.umn.edu/Ajanta/</a></td>
</tr>
<tr>
<td>Ara</td>
<td>Univ. of Kaiserslautern</td>
<td><a href="http://wwwagss.informatik.uni-kl.de/Projekte/Ara/index_e.html">http://wwwagss.informatik.uni-kl.de/Projekte/Ara/index_e.html</a></td>
</tr>
<tr>
<td>Hive</td>
<td>MIT</td>
<td><a href="http://www.hivecell.net/">http://www.hivecell.net/</a></td>
</tr>
<tr>
<td>JAFMAS</td>
<td>Univ. of Cincinnati</td>
<td><a href="http://www.eeccs.uc.edu/~abaker/JAFMAS/">http://www.eeccs.uc.edu/~abaker/JAFMAS/</a></td>
</tr>
<tr>
<td>MOLE</td>
<td>Univ. of Stuttgart</td>
<td><a href="http://mole.informatik.uni-stuttgart.de/">http://mole.informatik.uni-stuttgart.de/</a></td>
</tr>
<tr>
<td>NOMAD</td>
<td>Univ. of West Florida</td>
<td><a href="http://nomads.coginst.uwf.edu/">http://nomads.coginst.uwf.edu/</a></td>
</tr>
<tr>
<td>SOMA</td>
<td>Univ. of Bologna</td>
<td><a href="http://www-lia.deis.unibo.it/Research/SOMA/">http://www-lia.deis.unibo.it/Research/SOMA/</a></td>
</tr>
<tr>
<td>WAVE</td>
<td>Univ. of Surrey</td>
<td><a href="http://www-zorn.ira.uka.de/wave/">http://www-zorn.ira.uka.de/wave/</a></td>
</tr>
</tbody>
</table>

A more complete list can be found from: [http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/ma1/preview/preview.html](http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/ma1/preview/preview.html)
## Appendix C

### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>Authentication Agent</td>
</tr>
<tr>
<td>AC</td>
<td>Authentication Center</td>
</tr>
<tr>
<td>AOD</td>
<td>Append Only Data</td>
</tr>
<tr>
<td>ART</td>
<td>Agent Registration Table</td>
</tr>
<tr>
<td>CA</td>
<td>Certification Authority</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CRL</td>
<td>Certification Revocation List</td>
</tr>
<tr>
<td>DSD</td>
<td>Destination Specified Data</td>
</tr>
<tr>
<td>KN</td>
<td>Knowledge Network</td>
</tr>
<tr>
<td>PGP</td>
<td>Pretty Good Privacy</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>ROD</td>
<td>Read Only Data</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Socket Layer</td>
</tr>
<tr>
<td>TTL</td>
<td>Time To Live</td>
</tr>
<tr>
<td>WAVE</td>
<td>The mobile agent paradigm.</td>
</tr>
<tr>
<td>wa</td>
<td>Wave agent</td>
</tr>
<tr>
<td>WI</td>
<td>Wave Interpreter</td>
</tr>
<tr>
<td>WSA</td>
<td>Wave Security Add-on</td>
</tr>
<tr>
<td>WSS</td>
<td>Wave Secure System</td>
</tr>
<tr>
<td>WCI</td>
<td>Wave Certification Infrastructure</td>
</tr>
</tbody>
</table>

89