Department of Computer Science and Engineering

Chinese University of Hong Kong

Final Year Project Report

1999 - 2000

LYU9905

Security in Mobile Agent E-Commerce Systems

Prepared by Wong Tsz Yeung 97623604

Supervisor LYU Rung Tsong Michael

April, 2000

Abstract

Electronic commerce technology offers the opportunity to integrate and optimize global shopping. The computers of various companies will communicate with each other to determine the availability of products and to place as well as confirm orders. Mobile software agent has become an important aspect in electronic commerce. Electronic commerce and information retrieval are two prospective directions for applications of mobile agents. Nevertheless, security is a crucial concern for such systems. In this report, we will discuss about the mobile agent technology for e-commerce system. Also we will describe the system we have built for our final year project – the Shopping Information Agent System (SIAS) based on mobile agent technology. We will discuss the security problem issues for the mobile agents, and particularly we will analyze possible security attacks by malicious hosts to agents in our system, and our solutions to prevent these attacks. Finally, reliability of e-commerce system is also an important aspect. We will discuss the reliability problem for our system, especially we will analyze the fault tolerance design in our system.

Table of Content

Introduction

Nowadays, electronic commerce systems have been established all over the world. Most of them have adopted the traditional client-server model in communication and data transfer. Our project is aimed to apply the use of mobile agent technology on top of e-commerce systems. Moreover, e-commerce systems always need to transfer critical and confidential data over the network, such as credit card number. The main theme of our project is to emphasize on the security problems and corresponding solutions. Last but not least, the reliability of the e-commerce system is also an important topic. In this project, we also investigate and implement solutions upon reliability of agent system.

We will presents the details of **SIAS**, *Shopping Information Agent System*, that we have implemented. SIAS is a web-based e-commerce mobile agent system that provides users with information of products for sale in an electronic marketplace. Advantages of SIAS include such agent's properties as delegation of tasks and reduction of communication costs.

The system is written in Java programming language with support of the Concordia application programming interface (API), which is developed by Mitsubishi Electric Research Lab. In the coming subsections, we will describe the basic design, implementation details as well as the functionality of SIAS. More importantly, we will describe the security design and implementation as well as reliability features of SIAS.

1. SIAS – Shopping Information Agent System

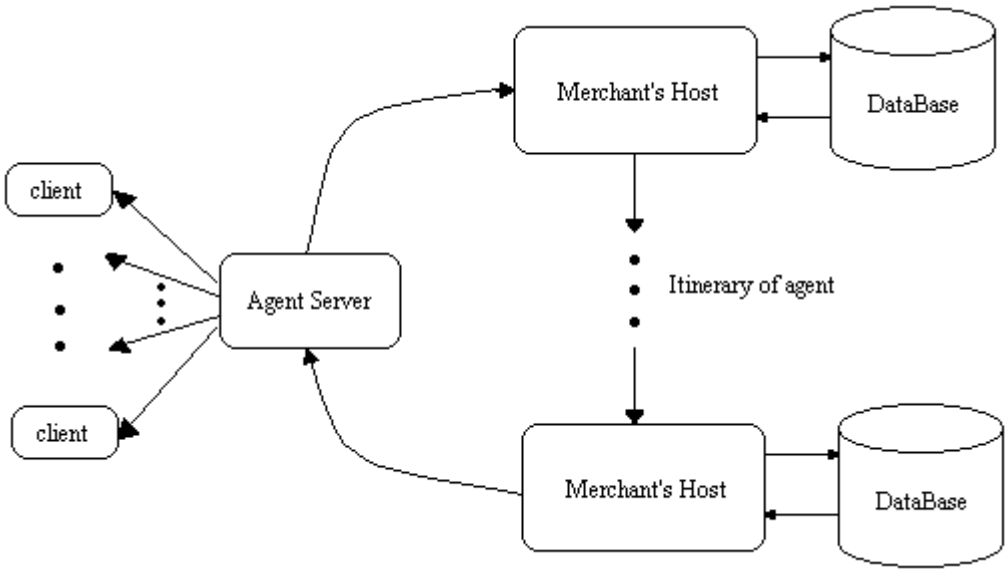


Figure 1. What the System Does

SIAS implements mobile agents to retrieve information of products in an electronic marketplace. An electronic marketplace consists of merchants that sell products on the network. Each merchant maintains a distinct database that consists of prices and the corresponding stocks of different products. Each database uses the same database table in order to achieve consistency in data format and simplicity of data retrieval.

SIAS keeps a roaster of all the merchant hosts on the network. It also keeps a name list of all the products selling on the marketplace. Users can use the client program to choose the products and the corresponding quantities that they desire to buy from the list of products. Whenever a user sends a request to the agent server, the agent server creates an agent for the user. The agent, on behalf of the user, collects the information about the availability and price of each product from merchant hosts in the network. The agent travels through the network according to its itinerary, or path, which is pre-determined by user before it is launched. After the agent has visited all hosts specified in its itinerary, it returns to its sender and reports the lowest prices and corresponding sellers. The design of the system is described in details in the next subsection.

1.1. System Design

SIAS is designed using the object-oriented paradigm because the concept of objects is useful to describe agents. There are four main types of objects in the system, namely **Agents**, **Launch Servers**, **Database Servers** as well as the **client program**. We describe the object details and control flow of the system in this subsection.

1.1.1. Object Description

I, The ***Agent*** object: it keeps a list of product identification numbers (IDs) and a list of corresponding quantities specified by users. It is responsible for travelling around the network and collect product information for users from different hosts. It also keeps an itinerary; it travels through the network according to the order of hosts in its itinerary. Whenever an *Agent* arrives at a host, it retrieves data from it and appends the retrieved data to a list of product entries. It is also responsible for calculating the purchasing combination that is lowest in price.

II, The ***Launch Server*** object: it is responsible for creating *Agents* for users, sending the *Agents* to the network, and receiving the *Agents* when they finish visiting all the hosts specified in their itineraries. It provides a gateway between client programs and the agent environment.

III, The ***Database Server*** object: it stores the information of products available at a particular host (each host has its own instance of this object), and is responsible for retrieving required information for an *Agent* when it arrives at the host. It can receive incoming *Agents*, and executes the codes of the received *Agents*. After executed the request of an incoming *Agent*, it sends the *Agent* to the next host.

IV, The ***Client Program***: it is a Java Applet. It lets users to choose the products and the corresponding quantities. Each instance of the client program can communicate with the *Launch Server* in order to launch *Agents* and receive results reported by *Agents*. It is a multi-threaded application, therefore it enables users to send out several queries at simultaneously.

The following figures show the details of the above objects.

Database Server

Methods:

- **handleAgent**

  This method is invoked when an *Agent* arrives at the *Database Server*. This method will invoke the **queryServer** of the incoming *Agent*, appends the result in the **List of product query result**, and sends out the *Agent*.

Figure 2. Object Details of Database Server.

Agent

Attributes:

- **List of Product ID**

  stores product IDs inputted by user

- **List of Product Quantities**

  stores the quantities of corresponding products

- **List of product query result**

  stores the product entries retrieved from *Database Servers*

Methods

- **queryServer**

  This method is invoked by *Database Servers*. When an *Agent* arrives at a *Database Server*, the *Database Server* invokes this method. The Database Server appends the query result into the **List of product query result**.

- **reportCheapest**

  This method is invoked by *Launch Server*. When an *Agent* finishes travelling all the hosts specified in its itinerary, it returns to the *Launch Server*. The *Launch Server* executes this method, and sends the result to *Client Program*.

Figure 3. Object Details of Agent.

Launch Server

Attributes:

- **Hashtable result**

  It is used to map the *Agent*'s ID to a character string. It stores the report generated by an *Agent*

Methods:

- **launch**

  This method is invoked when users send requests to the *Launch Server*. The *Launch Server* creates an *Agent*, initializes its attributes with the user's input, determine the itinerary, and, at last, launches the *Agent*.

- **handleAgent**

  This method is invoked when an *Agent* arrives at the *Launch Server*. This method will invoke the **reportCheapest** of the incoming *Agent*, and store the result string to the **result** hashtable for users to query.

Figure 4. Object Details of Launch Server

1.1.2. Flow Description

This subsection describes the flow of SIAS. It describes the flow in terms of the interactions between objects and how a user can make a successful request.

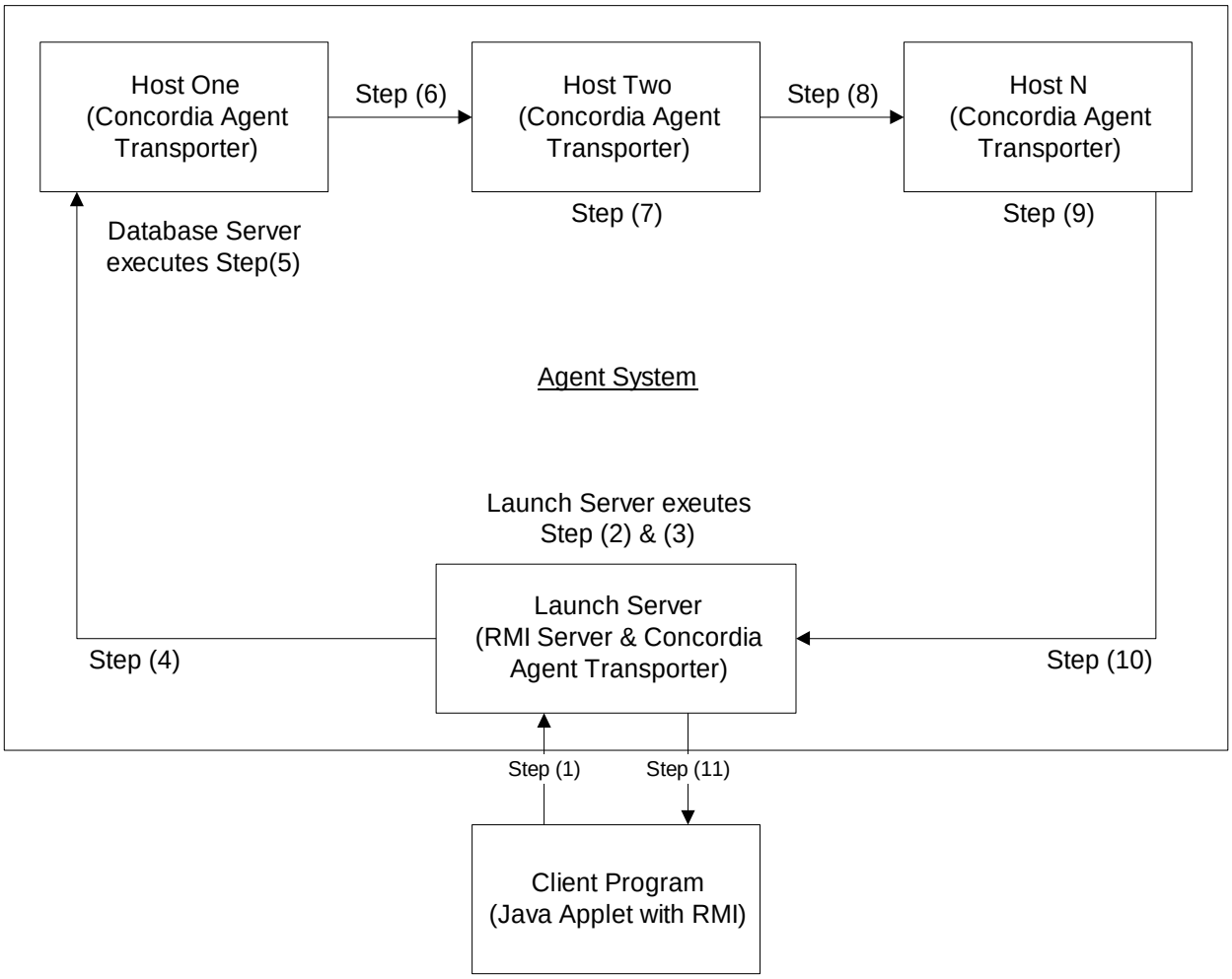
When a user makes a request for product information by using a client program, the request is sent to the *Launch Server*. The *Launch Server* will create a new *Agent* and initialize the variables according to the request made by the user. Next, the itinerary of the *Agent* will be instantiated. The *Agent* is now ready to launch.

After launching, the *Agent* will start visiting all the hosts specified in its itinerary. Whenever it arrives at a *Database Server*, it will stay there. Within the execution environment provided by the *Database Server*, the executions of *Agent* will be governed by the *Database Server*. In a normal case, i.e. the host is a trustful one, the *Database Server* will execute the *queryServer* of the incoming *Agent*, and then goes to another host. But, if the host is a malicious one, the execution of the *Agent* will be different. This will be covered in later sections. After the execution has been finished, the *Agent* will leave the host, and continues to visit other hosts specified in its itinerary in a round-trip manner.

After it has visited all the hosts specified in its itinerary, it will return to the *Launch Server*. The *handleAgent* of the *Launch Server* will control over the execution of the *Agent*. The *Launch Server* executes the *reportCheapest* of the incoming *Agent* and saves the report into its hash table *result*. At last, the report inside *result* will be sent to client program.

On the other hand, the client program will wait for the report while the Agent is travelling among hosts. When the report is ready, it will send a request to the server and receive it.

Figure 5 shows a detail flow of SIAS. Some technical terms that has been used in Figure 5 will be discussed in next subsections.



Explanation of steps:

1, Client program launches a request to the Launch Server object upon user input using Java Remote method Invocation (RMI);

2, Launch Server creates an Agent object;

3, Launch Server initializes the Agent with user-specified products and corresponding quantites, as well as the itinerary;

4, Launch Server sends the Agent to the network;

5, Database Server on Host One retrieves the required information for the incoming Agent;

6, Agent goes to the next destination;

7, Database Server on Host Two repeats Step (5);

8, Agent goes to other hosts in its itinerary;

9, Database Server on every host repeats Step (5);

10, Launch Server receives the returning Agent and calculates the cheapest prices;

11, Launch Server reports the result to client program by Java RMI.

Figure 5. The Flow Description of SIAS.

1.2. Implementation

SIAS is implemented by using Java programming language with the support of Concordia API. The choice of programming language and mobile agent platform, together with other implementation details will be discussed in the following subsections.

1.2.1. Choice of Programming Language

We choose Java programming Language to implement SIAS with three main reasons. First, Java is a platform independent language. This feature enables us to execute our system in different OS. Second, most mobile agent platforms that are currently available, including Concordia, is built on top of Java. Last but not least, Java provides many useful APIs such as **Java Cryptography Extension**, **JCE**, that enables us to build security measures on top of our current system.

1.2.2. Choice of Mobile Agent Platform

We choose Concordia mobile agent platform, among others like IBM Aglets Software Development Kit (ASDK) to implements SIAS because of its simplicity. The Concordia API is much easier than others. This saves a lot of time in developing our agent system.

Another important reason for us to choose Concordia is that it allows full manipulation of execution of agent code. This feature provides us a way to simulate a behavior of a malicious host, which does not execute agent code in an intended way.

1.2.3. Technology used in Objects

In SIAS, not only the Concordia API, but also other APIs are used. In this section, we will discuss the implementation details in the four main objects.

1.2.3.1. Agent

It is implemented by the support of Concordia. It is a subclass of a Concordia object, called **agent**. With the help of **agent**, we can ignore the difficulties concerning networking. The Concordia has implemented the dispatch (or marshal) and retract (or unmarshal) of **agent**. This reduces the difficulties in building mobile agent system.

1.2.3.2. Database Server

The *Database Server* has used objects from Concordia. Concordia provides a class called **Agent Transporter**. The **Agent Transporter** provides the object that has used it the ability to send out agents and receive incoming agents. The **Agent Transporter** provides an execution environment for incoming agents to execute their code. Under such an environment, the agent is fully controlled by the *Database Server*. Therefore, it is a good point for us to use **Agent Transporter** in order to simulate the behavior of malicious hosts.

The *Database Server* has a SQL server behind it. We choose to use *Oracle 8i* SQL server. As we are using Java programming Language, we have to use JDBC (Java Data Base Connectivity) to connect with SQL server. As Oracle 8i fully supports the use of JDBC, so we choose it.

1.2.3.3. Launch Server

The *Launch Server* also uses the **Agent Transporter** since it needs to send out *Agents* when client requests arrive, and receives incoming *Agents* when an *Agent* has completed its tour on the network.

The *Launch Server* uses Java Remote Method Invocation (RMI). RMI is a Java API which helps to build client-server distributed system. We use RMI as the connection between client programs (RMI clients) and *Launch Server* (RMI server). RMI supports multiple client connections. Therefore, the hash table, which maps *Agent*'s IDs to reports, used in the data structure of *Launch Server* is to enable multiple client connections.

A question may be raised: *Why don't use web browser (or client program) to send and receive agents directly?*

At first, our primary design uses client programs to send and receive *Agents* directly; there is no *Launch Server* in the very beginning design. Unfortunately, in order to use web browser to send out *Agent*s, the user has to install the whole package of the Concordia API. Also, the user has to install an additional plug-in, namely Java plug-in, for the web browser. After these, the user also has to do a lot of configurations in order to send and receive successfully. This is not user-friendly for users to do such tedious works.

But, with RMI, user does not need to install or configure any additional things. Just a Java-enable web browser can use to send and receive requests. Therefore, we choose to use RMI and introduce the *Launch Server* in our system.

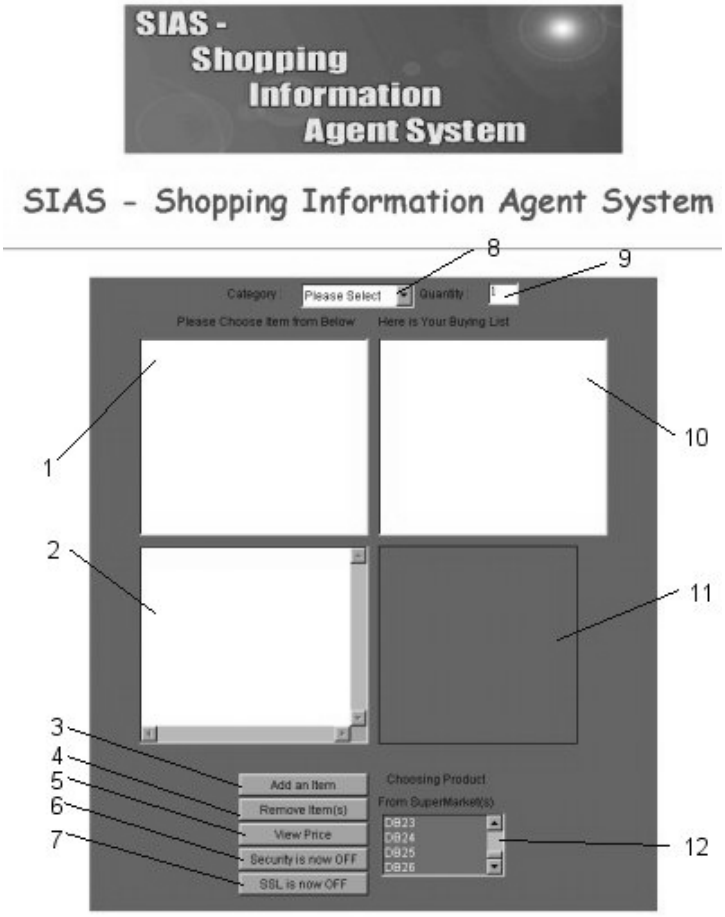
1.2.3.4. Client Program

The *Client Program* is a Java Applet. Every Java-enabled web-browser can run our client program. The *Client Program* uses RMI to connect with the *Launch Server*. Also, it uses *Thread*, a Java object, to develop the multi-threaded feature.

1.3. Snapshots

We have implemented a graphical user interface (GUI) for SIAS. We present some screen shots that demonstrate the use of SIAS in this subsection. Figures 6 to 8 show a typical run of the system. Our current system involves one *Launch Server* and 26 *Database Servers*.

Figure 6 shows the GUI when an user starts the applet on a web browser. On Figure 7, it shows the user is editing his/her buying list. At last, on Figure 8, it shows the user finishes his/her queries. It also shows that the user can make several queries at one time.



Description of the GUI

1, *Item List: This list contains all the products in the system. Users can choose their desired products from it.*

2, *Description Text: This text area is for displaying product description, such as name of the product.*

3, *Add Item Button: This button is to add a select item from Item List to Buying List.*

4, *Remove Item Button: This button is to remove a select item from the Buying List.*

5, *View Price Button: This button is to query information of products listed on Buying List.*

6, *Security Button: This button is to turn the Security attacks detection On or Off.*

7, *SSL Button: This button is to turn the Secure Socket Layer Transmission On or Off.*

8, *Category List: This list contains all the categories of products in the system.*

9, *Quantity: This text field is for user to change quantity when adding a product to Buying List.*

10, *Buying List: This list contains all the products that users have chosen.*

11, *Photo Displaying Area: This area is used for displaying the photo of the selected product.*

12, *Merchant List: This list contains all the merchants in the system. Users can have choices on selecting which supermarkets that users want to query.*

Figure 6. Applet starts, showing the user interface of the system.

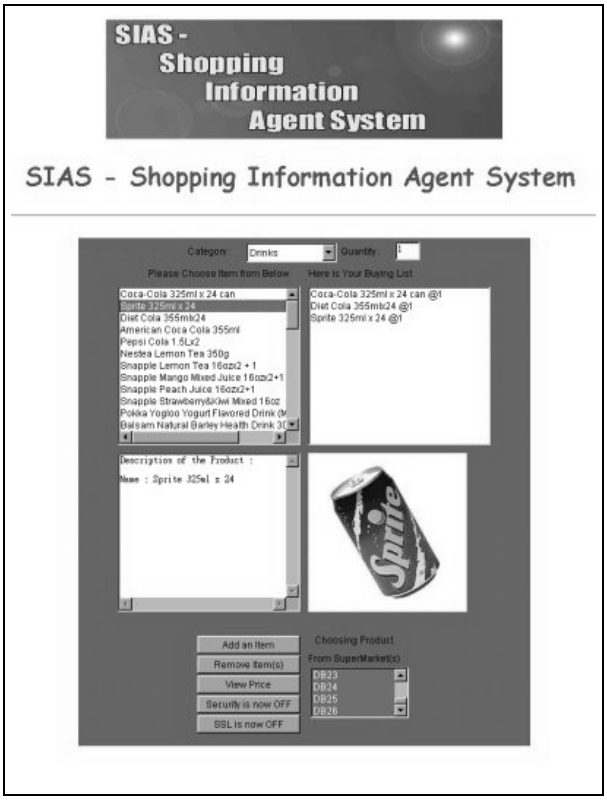
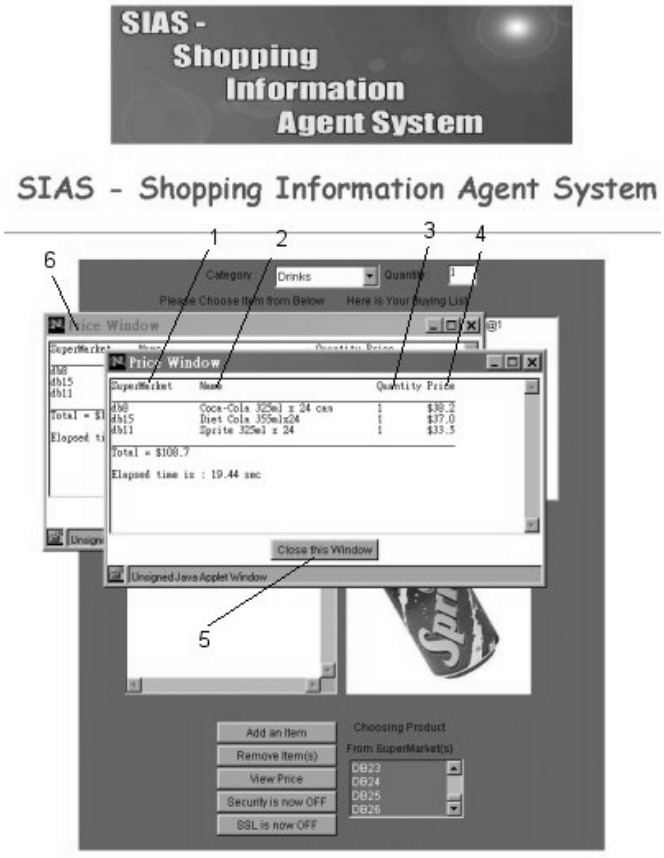


Figure 7. User is now choosing products into his/her Buying List.



Description of Price Window:

1, *Supermarket Column: this column displays, for each product, the merchant that is selling at the lowest price.*

2, *Name Column: this column displays the name of each product.*

3, *Quantity Column: this column displays the quantity of each product that users have specified.*

4, *Price Column: this column displays the price of each product times the quantity specified by users.*

5, *Close Window Button: this button is to close the Price Window*

6, *Another Price Window: this shows the applet can handle multiple queries simultaneously.*

Figure 8. The system reports the query results in the Price Windows.

2. Security Design in SIAS

SIAS is a web-based system, attacks from the Web to the system are likely, and security is an important issue of the system design. Moreover, system security is of crucial importance to applications in an electronic marketplace, where money transaction is concerned. This section describes the security challenges of SIAS, and presents a simple but original approach to solve the problems.

SIAS is a mobile agent system, and is therefore subject to all kinds of attacks described in previous sections. Both host security and agent security would be issues of SIAS. However, since we have built SIAS using the Java programming language, which provides strong security mechanisms to protect hosts against malicious programs or agents through the use of Java Virtual Machine (JVM), host security problem is very much simplified and solved. On the other hand, agent security needs much more concerns. In what follows, only agent security of SIAS against malicious hosts would be discussed.

In the following subsections, we will discuss the security problem in SIAS, describe how a malicious host performs attacks on an *Agent*, as well as the solution to detect such attacks. This will cause changes on the primary design of SIAS.

2.1. Security Problems in SIAS

To start our discussion of security problem in SIAS, we first have to address the security requirements for SIAS. There have three primary requirements:

1, *Integrity*: the query results reported by an *Agent* must truly represent the market prices of the products and at the quantities specified by the user.
2, *Confidentiality*: information collected from a host by an *Agent* should not be disclosed to other hosts or *Agents*.
3, *Authenticity*: an *Agent* must visit and collect information truly from the list of hosts specified by users, i.e. the itinerary.

None of the above 3 requirement should be violated, or the *Agent* is **suffered** from security attack(s).

Without a special design, all these requirements can be easily violated by the attacks of malicious hosts. There are four possible types of such attacks to agents that can compromise the security of the system, namely **modification of query product IDs of an Agent**, **modification of query quantity of query quantities of an Agent**, **spying out and modification of query results**, as well as **modification of itinerary of an Agent**.

To limit the complexity of our discussion, we impose one assumption in SIAS security design: **Only one or no malicious host is on the network**.

2.1.1. Modification of Query Product IDs

The list of product IDs specified by user is stored as the product ID list attribute in the *Agent* (see Figure 2). However, the list is store in **plain text form**. When an *Agent* arrives at a malicious host, the malicious host can easily spy out the list of product IDs. As the *Agent* is under the full control of the host, the host can changes the product IDs very easily. When the *Agent* goes to another host, the later host would not aware that the list of product IDs has been changed. The host would response to the wrong product IDs and report wrong information to users. This violates the **integrity requirement**.

The malicious host can benefit from this kind of attack. When an *Agent* returns back to the *Launch Server*, it will check against the list of product IDs and the retrieve product information in order to find the cheapest purchasing combination. The malicious host can eliminate all the competitors before it if the system is not clever enough to detect such attacks. If the malicious host is the last destination in the itinerary of an *Agent*, it can completely dominate the choices of user. The user can only choose the malicious host, although it probably provides the most expensive goods, as it seems to the user that there is only one store provides such products.

2.1.2. Modification of Query Quantities

Similar to the modification of query products, when an *Agent* goes to a malicious host, the malicious host can change the quantities of products the *Agent* wants to query as the list of query quantity list is simply in **plain text form**. When the *Agent* goes to another host, the later host will respond to the modified quantities of query, and report wrong information. This also violates the integrity of queries.

The malicious host can still benefit from this kind of attack, but less than that of the previous kind of attack. In a scenario that the malicious host is the last one in the itinerary of an *Agent*, the malicious host can force the *Agent* is reported as many quantities as possible. If owner of the victim *Agent* is not careful enough and cannot aware of such changes in the report, the user will be cheated.

2.1.3. Spying Out and Modification of Query Results

Agents carry query results is also in plain text form. Therefore, when an *Agent* goes to a malicious host, the malicious host can spy out and modify the results that the *Agent* has collected from previous hosts in such a way that the changed results would favor the malicious host itself. For example, a malicious host may raise the prices quoted by other hosts, to convince the user that it is selling at the lowest price, which is not the truth. This violates the confidentiality and integrity of

query results.

From the viewpoint of the malicious host, this attack is a very effective one. The malicious host can successfully eliminate competitors that have been visited by the victim *Agent* by raising their quoted prices. If the malicious host is the last one of the itinerary of the *Agent*, it can eliminate all its competitors.

2.1.4. Modification of Itinerary of an Agent

Inside the execution environment of the *Agent Transporter*, which is provided by Concordia API, a malicious host can have fully control over an *Agent*. It can access all its attributes, even the attributes that are forbidden to access. The itinerary, which is a forbidden area, is one of our concerns. When an agent goes to a malicious host, the malicious host can modify the itinerary (or path) of the *Agent* so that the *Agent* will go to a host not specified by user. This violates the authenticity requirement of the system.

This is a very important attack from the viewpoint of a malicious host. A malicious host can change the itinerary of an *Agent* in such a way that its next destination is the *Launch Server*. Then, the *Agent* will be cheated as it has finished its tour on the network. Therefore, the malicious host can get rid of all the hosts that are ranked behind it in the itinerary.

2.1.5. Hybrid Attack

After describing the above four kinds of attacks, one can easily come up with ideas of mixing the attacks.

For example, a malicious host can combine the third and the fourth kinds of attacks to get rid of all its competitors no matter where it is on the itinerary of the victim *Agent*. A malicious host can first carry out the third attack; it can raise all prices inside the query result list. Then, it can proceed with the fourth attack; it can change the next destination to be the *Launch Server*. Eventually, the effect becomes the *Agent* will choose all the products from the malicious host.

One can also comes up with another kind of attack combination that is much more complex (indicated on Figure 9). This would combine three kinds of attacks, except the second one. First, when the agent comes to a malicious host, the host can raise all prices inside the query result list. Then, it will backup the *list of product IDs* inside the *Agent*, at last, it sets the *list of product IDs* to nothing (or null). The effect of this is to let all the hosts that will be visited later to retrieve nothing from the databases. One more step need to do before the *Agent* launches is to append a destination in the itinerary. One can add a destination to be the malicious host itself before the *Agent* returns back to the *Launch Server*. What is the aim of this? The main purpose is to fake the system; when the

victim *Agent* visits the malicious host again, it will restore the *list of product IDs* to be the original list. Then, it seems that the *Agent* has visited all the hosts, but all the products that are cheapest are from the malicious host, however, may be the malicious host is selling the most expensive goods.

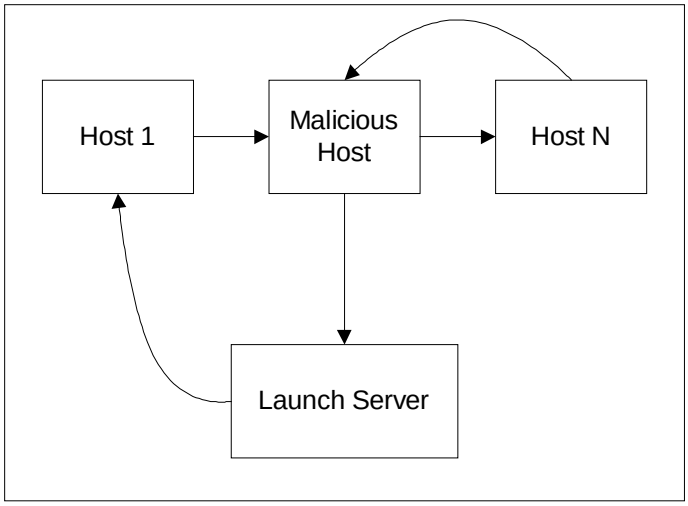


Figure 9. A complex hybrid attack.

The above is only a subset of possible attacks. There are other attacks such as replaying of query results and masquerading of hosts. However, these attacks are more complex, and require more efforts for designing as well as implementing both attacks and defenses. For the time being, we consider the four simple attacks only.

2.2. Design of Solutions to Security Problems of SIAS

Having figured out the four system vulnerabilities described above, we have to implement mechanisms to protect our systems against exploitation of these vulnerabilities. As there is currently no good solution to mobile agent security in general, therefore we have to design our own mechanisms to defend against possible attacks.

We have developed a simple but original approach to protect agents in SIAS against attacks from malicious host, based on public key cryptographic techniques. It is actually a mixed approach of the solutions, i.e., *establishing a closed network*, *agent tampering prevention* and *agent tampering detection*, discussed before.

2.2.1. Closed Network

We introduce a new object, namely **Key Server**, into our system, which provides a public key infrastructure (or PKI) for *Agents* and hosts in the system. Each *Agent* or host should have a public key certificate registered to the *Key Server* for encryption or decryption purposes later on. The *Launch Server* generates a pair of keys for each agent created, and registers the public key of the *Agent* with a unique agent identification number (or *Agent* ID) to the *Key Server* at run-time. On the

other hand, each host must identify itself and register its public key with its unique network address (in our implementation, IP address is used) to the *Key Server* at the time it starts; it is a means as a formal paper writing. Thus, in effect, this establishes a closed set of hosts registered and known to the *Key Server*. *Agents* are then confined to travel among a closed network formed by these hosts. This can get rid of foreign malicious hosts.

2.2.2. Agent Tampering Prevention

In order to protect query integrity, an agent can encrypt its list of products and quantities using its private key before it is launched from the *Launch Server*. Since only the *Launch Server* possess the private key for the agent, malicious hosts would not be able to duplicate the encrypted product and quantity lists.

Moreover, each host should encrypt the query results returned to the agent with the public key of the agent. Therefore, the malicious host cannot modify the query result since it does not have the private keys of other hosts. The Launch Server can decrypt the original query result, and confidentiality of query results is achieved.

I, {Product ID list} changed to: E_A(Product ID list)
II, {Product Quantity list} changed to: E_L(Product Quantity list)
III, {Query result} changed to: E_H(Query result)

Key
A: agent;
H: host;
E_X(Y): the ciphertext of Y encrypted by the private key of X;

Figure 10. Agent Tampering Detection.

2.2.3. Agent Tampering Detection

The itinerary of an *Agent* is a variable hidden by the Concordia system and normally not accessible. However, hosts can actually have access to the itinerary of an incoming agent by controlling the execution of the Concordia *Agent Transporter*. A malicious host would be able to change the itinerary of the agent. As before, the straightforward method of protecting the itinerary is to encrypt it. However, this requires modification of the agent transporter of Concordia, i.e. hacking the source code of Concordia API, which is not desirable and also not feasible to do so.

We work around the problem by introducing a new variable, namely **Encrypted_Itinerary**, which makes use of the itinerary of an *Agent*. When an agent arrives at a host, the host should read the itinerary of the agent, and encrypt the host name (or IP address) of the next host using its own private key to form an encrypted itinerary EI_1. EI_1 will be assigned to *Encrypted_Itinerary*. Then,

when the agent arrives at a second host, the itinerary of the *Agent* is changed as the entry for the visited host has been removed. The second host should encrypt the new itinerary, with its own private key. However, before this happens, EI_1 should concatenate with the itinerary it reads from the *Agent*. At last, the concatenated list will be assigned to *Encrypted_Itinerary*. This keeps on to form a chain of encrypted itineraries. When the agent returns to the *Launch Server*, the *Launch Server* will decrypt the chain of encrypted itineraries layer by layer using the public keys of the hosts. This enable our system to check the consistency of all itineraries and check with a copy of the original itinerary it saves before launching the *Agent*. If a malicious host ever changes the itinerary of the agent, it is likely to be reflected in the encrypted itinerary chain and detected finally.

New attribute for Agent: Encrypted_Itinerary
Encrypted_Itinerary = E_{H1}*(Next Host at Host H1) +* E_{H2}*(Next Host at Host H2)+ ... +* E_{Hn}*(Next Host at Host Hn);*

At Launch Server, we compare original itinerary to :
$D_{H1}(E_{H1}$*(Next Host at Host H1)) +* $D_{H2}(E_{H2}$*(Next Host at Host H2))+ ...+*
$D_{Hn}(E_{Hn}$*(Next Dost at Host Hn))*
;
If they are not equal, we can detect which host is the malicious one by detecting which decrypted part goes wrong.

Key
H: host;
$E_X(Y)$*: the ciphertext of Y encrypted by the private key of X;*
$D_X(Y)$*: the plaintext of Y decrypted by the public key of X;*

Figure 11. Agent Tempering Detection.

2.3. Implementation

We have introduced 3 main modifications to SIAS, namely the **RSA** object, **Key Server** object and cryptographic measures.

2.3.1. RSA Object

The **RSA** object is the core part of our public key infrastructure (PKI).

This object implements the famous asymmetric cryptographic algorithm, RSA. When a *RSA* is constructed, a pair of private and public keys will be generated. With the keys, 2 main operations can be carried out, which are **encrypt** and **decrypt**. These two operations change a message (a character string only) into a ciphertext (a character string), and vice versa.

We implement this *RSA* object with a great help from the Java class called **BigInteger**, in the package **java.math**. It provides such helpful methods as modulus arithmetic that enable us to implement the *RSA* algorithm a lot easier. Figure 12 shows the object details of *RSA*.

RSA

Attributes:

- **private key & public key**

  This key pair is generated when the object is constructed. The length of a key is 128 bit.

Methods:

- **getPrivateKey**

  It is used to retrieve the private key

- **getPublicKey**

  It is used to retrieve the public key

- **encrypt**

  It accepts a string and a key. It returns a ciphertext as a string.

- **decrypt**

  It accepts a ciphertext and a key. It returns a plain text as a string.

Figure 12. Object details of RSA.

2.3.2. Key Server

The *Key Server* acts as a **Certificate Authority (CA)** in our PKI. The *Key Server* is used to store and distribute public keys that belong to *Agents* or hosts. Whenever a *Database Server* starts up or an *Agent* is created, a public key is stored to the *Key Server* by calling the *putPublicKey* method of *Key Server*. Whenever the *Launch Server* needs a host's public key or a *Database Server* needs an *Agent*'s public key, it calls the *getPublicKey* method of *Key Server*. The connection between *Database Servers* or *Launch Server* and the *Key Server* is done by Java Remote Method Invocation (RMI).

A question may be asked: *Why use RMI instead of agents to transmit the public keys?*

The answer is simple. We use PKI to solve the security problem in our agent system. But, if the PKI involves agents, a new security problem concerning agents will be raised. Then, we may need to use another security measure to solve the induced problem. This will start an endless loop in building the PKI. So, we choose to use RMI in transmission of keys in our PKI. Although RMI may induce other security problems, it is more feasible than using agents. A solution to RMI security problem is to use *Secure Socket Layer* (SSL) transmission. This can prevent intruders from stealing the public key or doing replaying attack.

Figure 13 shows the object details of *Key Server*.

Key Server

Attributes:

- **Hashtable key**

  It is used to map *Agent's* ID as well as host's IP address to a RSA public key.

Methods:

- **getPublicKey**

  It is used to retrieve a public key by supplying a host's IP address or an *Agent's* ID as the method argument.

- **getPrivateKey**

  It is used to hash a public key to the **key** hashtable with an *Agent's* ID or a host's IP address as the hash key.

Figure 13. Object Details of Key Server.

2.3.3. How Cryptographic Measures Work

With the introduction of the above objects, our security system still needs a mechanism in order to detect or prevent security attacks. Our design is to use *Launch Server* to detect whether attacks happen or not.

Before an *Agent* leaves the *Launch Server*, the *Launch Server* will do several security measures first.

- First, the *Launch Server* will back up the original **list of product IDs** and **list of product quantities**. This is used for validating the 2 lists when the *Agent* comes back.
- Second, the *Launch Server* will back up the original **itinerary**. This is used to detect malicious host if it alters the itinerary of the *Agent*.
- At last, the *Launch Server* will encrypt the **list of product IDs** and **list of product quantities**. Then, the *Agent* can be launched.

After an *Agent* has traveled through the network and goes back to the *Launch Server*, the *Launch Server* will check the attributes inside the *Agents* against its backups.

- First, the *Launch Server* will decrypt the **list of product IDs** and **list of product quantities**. Then, it will check them against its backup versions. If this fails, it can report to the user that security requirement is violated.
- Second, the *Launch Server* will decrypt the **Encrypted_Itinerary** in the *Agent* in the way that is specified in Figure 11. It will also check for the violation of security requirements.
- If none of the above check is failed, the *Launch Server* can continue to report the cheapest purchasing combination to user.

These measures have introduced additional attributes to *Launch Server*. The additional attributes are

all hashtables, so this can efficiently map *Agent's* ID to the backup data. Figure 14 shows the additional attributes to *Launch Server*.

Additional Attributes to Launch Server:

Hashtable originalProductID

This maps *Agent's ID* to the backup list of product IDs

Hashtable originalQuantity

This maps *Agent's ID* to the backup list of product quantities

Hashtable originalItinerary

This maps *Agent's ID* to the backup itinerary

Figure 14. Addition Attributes to Launch Server.

2.4. Flow Description

When a user makes a request for product information by using a client program, the request is received by the *Launch Server*. The Launch Server will create a new *Agent* and initialize the variables according to the request made by the user. Next, the itinerary of the *Agent* will be instantiated. But, before it launches, we create an *RSA* object for the created *Agent*. The *Launch Server* then sends the public key of the *Agent* to the *Key Server*, and the private key is stored in the *Launch Server*. The purpose of saving the private key in *Launch Server* is to avoid the tempering as well as modification of the private key if it is stored inside an *Agent*.

Whenever it arrives at a *Database Server*, the *Database Server* will retrieve the public key of the incoming *Agent* from the *Key Server*. The *Database Server* then uses the public key to decrypt the *list of product IDs*. The query results are protected by using cryptographic techniques, so the *Agent* will leave after all the query results are retrieved from the database. The *Agent* continues to visit other hosts specified in its itinerary in a sequential way.

After it has visited all the hosts specified in its itinerary, it will return to the *Launch Server*. The *Launch Server* will retrieve public keys of all the hosts that are specified in the itinerary of the incoming *Agent*. Then, the *Launch Server* will check the integrity of the *list of product IDs*, also the *list of product quantities*. Moreover, the *Launch Server* will check whether the itinerary is changed or not. If all the security checks are passed, the *Launch Server* executes the *reportCheapest* of the incoming *Agent* and saves the report into its hash table *result*. Or, the Launch Server will issue error messages to the client saying that the *Agent* has been modified.

In Figure 15, we will describe the new flow of SIAS. Our current implement uses one *Launch Server*, one *Key Server* and 26 *Database Servers*.

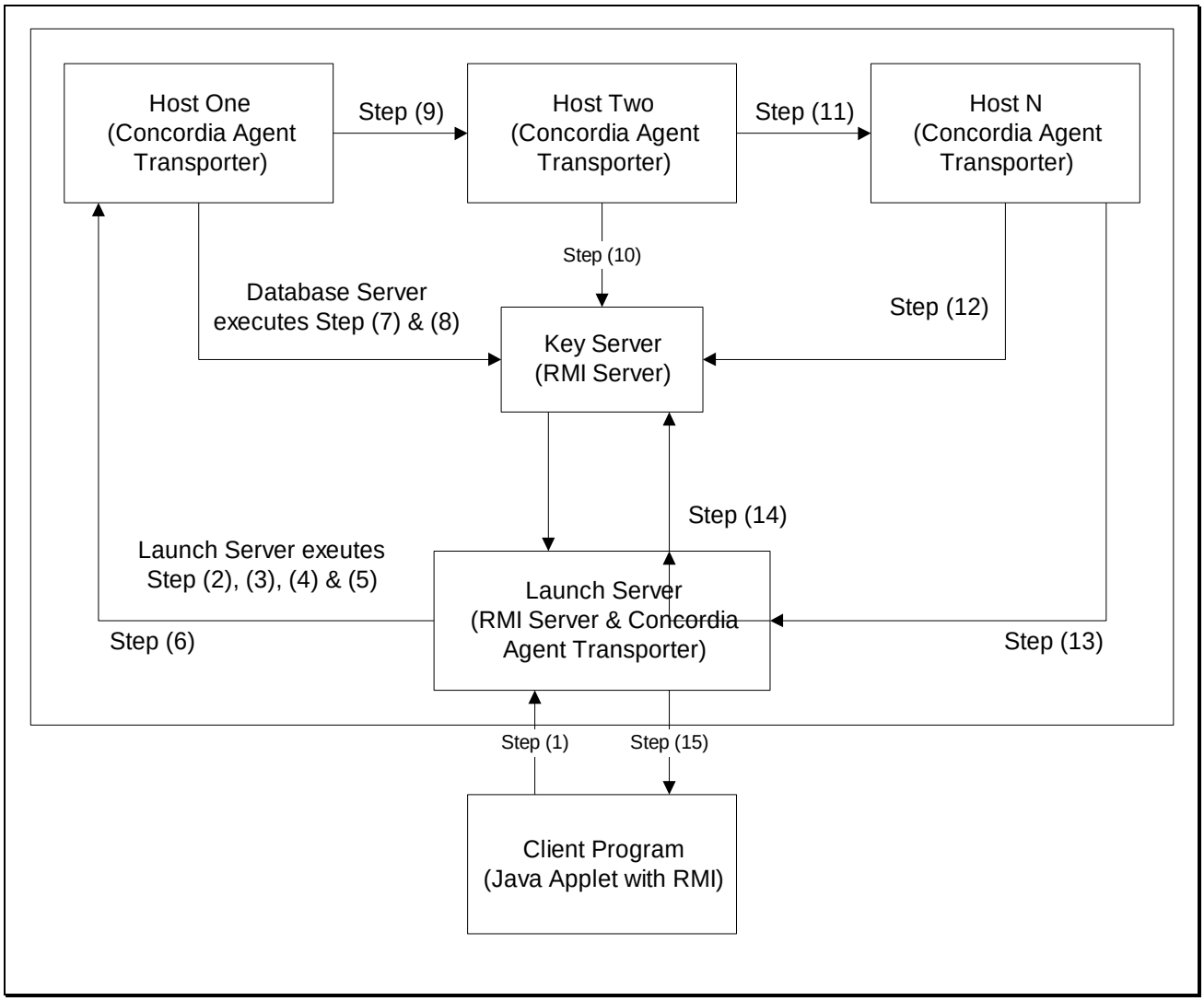


Figure 15. Flow Description of Secure SIAS. (1 of 2)

Explanation of steps:

1, Client program launches a request to the Launch Server upon user input by using RMI;

2, Launch Server creates an Agent object;

3, Launch Server initializes the Agent with user-specified products and quantities, and the itinerary of agent;

4, Launch Server generates a RSA key pair for Agent;

5, Launch Server encrypts product IDs and quantities lists for Agents and registers the public key of agent to Key Server;

6, Launch Server sends the Agent to the network;

7, Database Server on Host One retrieves public key of Agent from Key Server;

8, Database Server retrieves the required information for the incoming Agent, encrypts the result using its own private key, and encrypt the itinerary of Agent.

9, Agent goes to the next destination;

10, Database Server on Host Two repeats Steps (7) & (8);

11, Agent goes to other hosts in its itinerary;

12, Database Server on every host repeats Steps (7) & (8);

13, Launch Server receives the returning Agent and calculates the cheapest purchasing combination;

14, Launch Server decrypts the query product IDs and quantities to verify its integrity. It also decrypts the encrypted itinerary to detect malicious host. After checking is finished, the Launch Server sends a request to Key Server to destroy the public key of the Agent.

15, Launch Server reports the cheapest purchasing combination to client program.

Figure 16. Flow Description of the Secure SIAS. (2 of 2)

2.5. Secure Agent Transmission in SIAS

In this section, our focus is not on agent security. We focus on the security problem dealing with transmission of agents between hosts.

As agent is travelling in the network, it is easy for people to tap the network and steal the agent while transmission. The purpose of stealing the agent can be, for example, doing replaying attack; the hacker can save the agents, and then sends it out again. In this way, the system cannot distinguish whether the replaying agent is created by the user or not. This can violates the *Authenticity* of the system. In order to prevent this kind of attack, we choose to use cryptographic technique to deal with the problem.

2.5.1. Design

We adopt the idea of Secure Socket Layer (SSL) transmission in designing **Secure Agent Transmission**. In SSL protocol, it uses RSA algorithm to achieve a secure transmission channel.

However, our original RSA implementation cannot handle encryption and decryption of binary data. Therefore, we choose to use symmetric key cryptographic techniques to encrypt the transmitted data. Nevertheless, we cannot exchange a symmetric key on the network as it is not secure to do so. Therefore, we use the **Diffie-Hellman Key Exchange** for exchanging the symmetric key between 2 hosts securely. With two hosts sharing the same symmetric key, we can encrypt all the data that is need to transmit on the network.

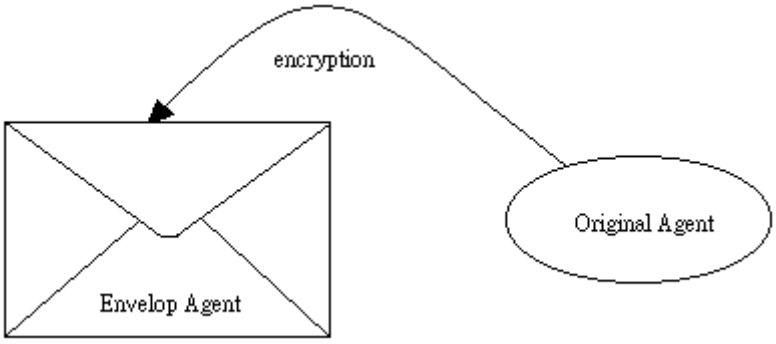


Figure 17. Envelop Agent Approach. (1 of 2)

Our approach is to use an **Envelop Agent Approach** to enclose the encrypted agent.

- First, we have to use **Diffie-Hellman Key Exchange** algorithm to exchange the symmetric key between the two communicating hosts.
- Then, we use the exchanged key to encrypt the agent on the sending host. We use bitwise XOR to encrypt the whole agent.
- As the object is messed up after encryption, the Agent Transporter at the receiving host cannot distinguish whether it is an agent or a pile of garbage. Therefore, we introduce an **Envelop Agent** in order to transmit the encrypted agent.
- We put the encrypted agent inside the **Envelop Agent**, then we send the **Envelop Agent** out.

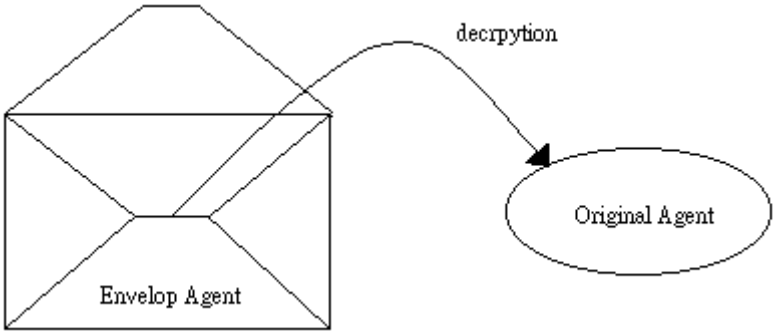


Figure 18. Envelop Agent Approach. (2 of 2)

- When receiving the **Envelop Agent**, the receiving host will decrypt the encrypted agent by

using the symmetric key exchanged between it and the sending host.

2.5.2. Implementation

We have introduced 3 new objects in order to achieve secure agent transmission. The approach is very similar the *RSA* and *Key Server*.

2.5.2.1. DHKey Object

The **DHKey** object contains a key pair, one is public and another is private. This object implements the key generation of **Diffie-Hellman Key Exchange** algorithm. It also implements the method for changing object to bytes, and changing bytes to object. The **encrypt** and **decrypt** method is implemented by using bitwise *Exclusive OR*.

DHKey

Attributes:

- **private key & public key**
  This key pair is generated when the object is constructed.

Methods:

- **getPrivateKey**
  It is used to retrieve the private key.
- **getPublicKey**
  It is used to retrieve the public key.
- **toByteArea**
  It is used to transform an object to an array of bytes.
- **toObject**
  It is used to transform an array of bytes to an object.
- **encrypt**
  It is used to transform an object into an encrypted byte array
- **decrypt**
  It is used to transform an encrypted byte array to the original object

Figure 19. Object Details of DHKey.

2.5.2.2. DHKey Server

DHKey Server plays a similar role as the *Key Server*. It is used to store and distribute public keys that belong to hosts. Whenever a *Database Server* or *Launch Server* starts, a public key of the host is sent to and stored in *DHKey Server* by calling the *putPublicKey* method of *DHKey Server*. Whenever the *Launch Server* or a *Database Server* needs a host's public key, it calls the

getPublicKey method of *DHKey Server*.The connection between *Database Servers* or *Launch Server* and the DHKey Server is done by Java RMI.

The reason for using Java RMI in connection with *DHKey Server* is the same as the *Key Server*. The following figure (Figure 20) shows the object details of *DHKey Server*.

DHKey Server

Attributes:

- **Hashtable key**

  It is used to map host's IP address to a Diffie-Hellman public key.

Methods:

- **getPublicKey**

  It is used to retrieve a public key by supplying a host's IP address as the method argument.

- **getPrivateKey**

  It is used to hash a public key to the **key** hashtable with a host's IP address as the hash key.

Figure 20. Object Details of DHKey Server.

2.5.2.3. Envelop Agent

Envelop Agent is an agent that is capable to carry an encrypted *Agent*. It doesn't have any methods as it is just used for transportation.

Envelop Agent

Attributes:

Byte Array Encrypted_Agent

- It is an array of bytes for carrying an encrypted *Agent*.

Figure 21. Object Details of Envelop Agent.

2.5.3. Flow Description

When a user makes a request for product information by using a client program, the request is received by the *Launch Server*. The Launch Server will create a new *Agent* and initialize the variables and itinerary according to the request made by the user. But, before it launches, we retrieve the public key of its next destination from *DHKey Server*. The *Launch Server* then computes the symmetric key by using the private key stored in the *Launch Server* and also the retrieved public key. We then encrypt the *Agent* and put it into the *Envelop Agent*. At last, we send out the *Envelop Agent* to the network.

Whenever the *Envelop Agent* arrives at a *Database Server*, the *Database Server* will retrieve the public key of its previous host from the *DHKey Server*. The *Database Server* then uses the retrieved public key and its private key to generate a symmetric key to decrypt the encrypted *Agent*. Then, the *Database Server* can use the original *Agent* to make query. After the query has been finished, the original *Agent* must be encrypted again. However, the *Database Server* has to generate another key because the target host has changed. Again, it retrieves public key, generate a symmetric key with the public key and its private key, encrypt the *Agent*, put the encrypted *Agent* back to the *Envelop Agent*, and at last send it to the next host.

After it has visited all the hosts specified in its itinerary, it will return to the *Launch Server*. The *Launch Server* will retrieve public keys of the previous host that the *Agent* has travelled. Then, the *Launch Server* will decrypt the encrypted *Agent*, and carry out *reportCheapest* of the original Agent. At last, the result will be sent to the user.

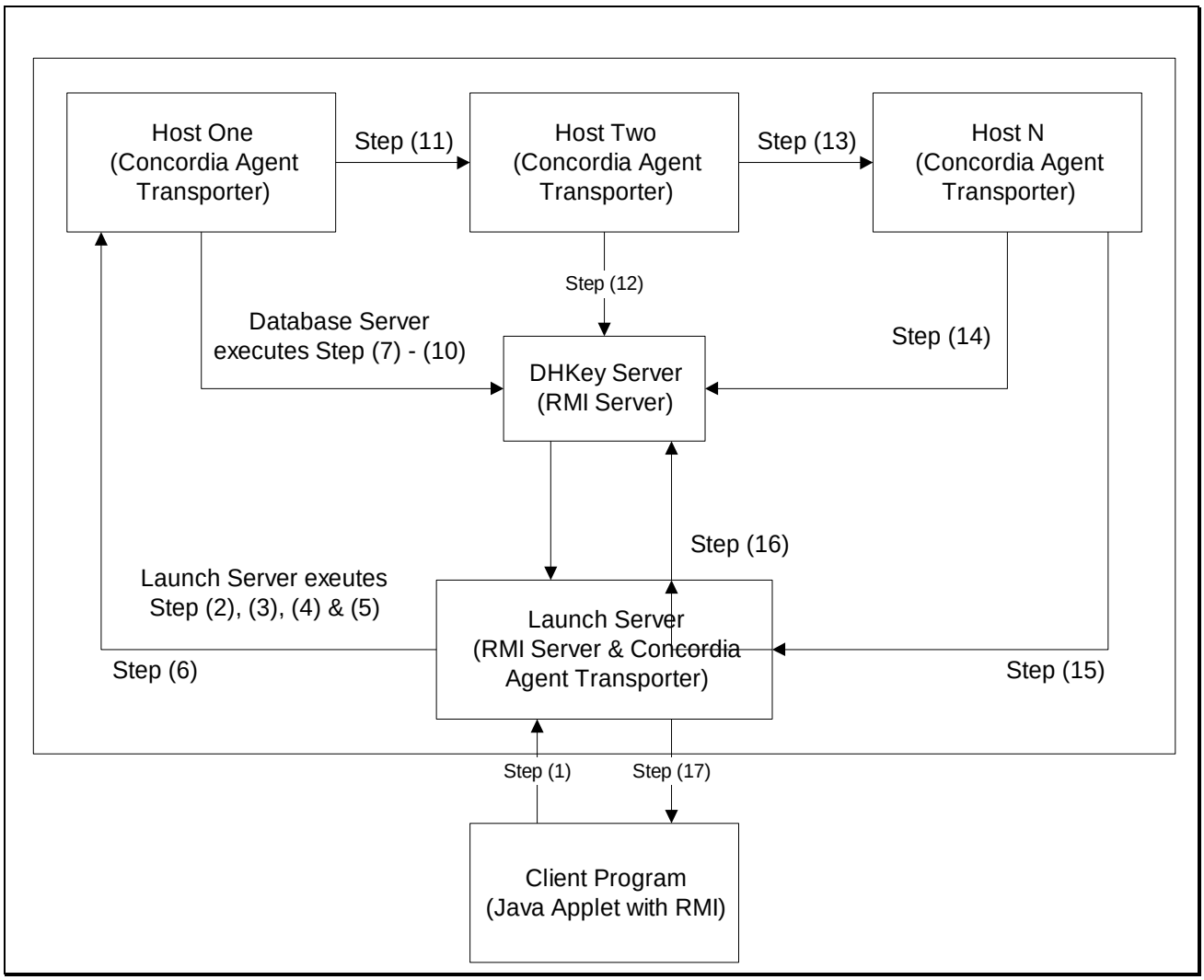


Figure 22. Flow Description of Secure Agent Transmission. (1 of 2)

Explanation of steps:

1, Client program launches a request to the Launch Server upon user input by using RMI;

2, Launch Server creates an Agent object;

3, Launch Server initializes the Agent with user-specified products and quantities, and the itinerary of agent;

4, Launch Server retrieves public key of next host of the Agent to generate a symmetric key;

5, Launch Server encrypts the Agent using the symmetric key, and puts the encrypted Agent into the Envelop Agent;

6, Launch Server sends the Envelop Agent to the network;

7, Database Server on Host One retrieves public key of Launch Server from DHKey Server;

8, Database Server generates a symmetric key and decrypts the encrypted Agent inside the Envelop Agent.

9, Database Server retrieves the required information for the incoming Agent

10, Database Server retrieves the public key of Host Two and generates another symmetric key for encrypting the Agent. After the Agent is encrypted, it is put back into the Envelop Agent.

11, Envelop Agent goes to the next destination;

12, Database Server on Host Two repeats Steps (7) to (10);

13, Agent goes to other hosts in its itinerary;

14, Database Server on every host repeats Steps (7) to (10);

15, Launch Server receives the returning Agent and retrieves the public key of Host N. It generates a symmetric key and decrypts the encrypted Agent in the Envelop Agent;

16, Launch Server calculates the cheapest purchasing combination after decrypting the Agent;

17, Launch Server reports the cheapest purchasing combination to client program.

Figure 23. Flow Description of Secure Agent Transmission. (2 of 2)

3. Reliability in SIAS

SIAS is a web-based e-commerce system. Not only security is an important aspect, but also reliability is a vital feature of a successful system. As an e-commerce system always involved such critical data as money transaction, frequent data lost is not acceptable. Therefore, we have to design measures in order that when a component fails, it can be restarted within a certain amount of time.

As our system is highly dependent on the performance of Concordia API, the faults that are raising from the Concordia API are not avoidable. Therefore, we cannot do anything to stop the failure of anyone of the Concordia components. Hence, we have to design measures to restart a Concordia component whenever it fails.

3.1. Design

We have designed an original approach that can efficiently restart Concordia components whenever a failure is detected. We will also discuss its advantages and weaknesses.

3.1.1. Faulty Components

SIAS is built on top of the Concordia API, therefore many components are having chances to fail.

- *Database Server* is one of the *Faulty Components*. In Concordia architecture, whenever an agent wants to travel to next destination, but, in fact, the expected host is down or does not exist, an exception will be raised, and the agent will stop travelling on the network. More than that, the agent will be destroyed also. This is an undesirable consequence for a data-critical mobile agent system. As mobile agent carries data or result with it while it is travelling on the network; such an exception will lead a **total data loss**.

  Moreover, as the *Database Server* provides an execution environment for *Agent* to execute its code, the failure of the *Database Server* will also destroy the *Agent* that is executing on the host. This will also lead to a **total data loss**.

- *Launch Server* is also one of the *Faulty Components*. As one of the uses of *Launch Server* is to send and receive *Agents*, it also has the same problem as the *Database Server*.

  The best case is:

  Launch Server fails when an *Agent* is just created. This loss will have the least effect. No data will be lost in this case.

  The worst case is:

  Launch Server fails when an *Agent* returns / is returning to *Launch Server*. This will lose all the data that the *Agent* has collected on the network.

However, the *Launch Server* has stored many important data that is employed in the security measures of SIAS, such as private keys of *Agents*. The failure of the *Launch Server* will eventually lead to the recovery of the whole system. One simple reason is that we cannot recover the private keys of *Agents* when the *Agents* come back after the recovery; those corresponding public keys will become useless, and the *Agents* that are on the network will fail the security check with the lack of private keys. Therefore, we have to recover the whole system.

- The *Key Server* and *DHKey Server* are not *Faulty Components* since they do not involve Concordia component. However, if they fail, for other reasons such as failure on power supply, their recovery processes will lead to a whole system recovery. It is because their failures will lose all the public keys stored in them. Without the public keys, the **public key infrastructure** as well as the **secure agent transmission** will be failed. The only way is to restart the whole system.

The above context has listed the situations of failures as well as the consequences when such failures do happened. The following subsections will describe the measure to tackle the above failure behaviors.

3.1.2. Logging System

One of our approaches is to use **LOG** to monitor the system status. With a carefully designed **Logging System**, we can determine the state as well as the availability of a component by inspecting the **Log File** of each component. We can search for **Error Messages** and **Abnormal States** in order to determine whether a component is needed to restart or not.

In our design, we have one *Launch Server*, one *Key Server,* one *DHKey Server* as well as 26 *Database Servers*. Totally, we have 29 *Log Files* to inspect. Each *Log File* has at most two states to be indicated, one is **Initialization** and another one is **Handle Agent**.

3.1.2.1. Logging in Database Server

For *Database Server*, the **Initialization** stage involves the following processes:

- Starting an *Agent Transporter*, which is a Concordia object. If the *Database Server* fails to start the *Agent Transporter*, it should leaves an *Error Message* on the *Log File* stating that it fails in the *Initialization Stage*, and the *Database Server* needs to be restarted.

- Sending a public key to *Key Server* and *DHKey Server*. If the *Database Server* fails to send the keys to *Key Serve* or *DHKey Server*, this indicates that one of the servers, or both of them, is out

of order. The *Log File* of *Database Server* should write an *Error Message* on its *Log File* to indicate the failure in the *Initialization Stage*. However, no need to restart the *Database Server* this time, as the failure of *Key Server* or *DHKey Server* will lead to a whole system recovery. This will be discussed soon.

```
[initialization] Local Host : 137.189.88.211
[initialization] Public Key is sent to KeyServer
[initialization] Public Key is sent to DHKeyServer
[initialization] Initialization Done.
[initialization] Listening to Incoming Agents
[handle agent] Agent arrived
[handle agent] finish query server
[handle agent] send agent to next host
.....
```

Figure 24. A Database Server Log File.

For the **Handle Agent** stage, it involves the following processes:

- The notification of the arrival of the *Agent*. This process will not involve any exceptions.

- Querying the SQL server. In this process, mainly SQL server exception will be caught. For example, the SQL server is down or the products do not exist in the SQL server. In this case, the *Error Message* does not need to write on the *Log File*. It is because the failures are coming from the SQL server, not the *Database Server*.

- Sending *Agent* to the next host. This process will involve failures, especially this will cause the whole *Agent* to be destroyed because of the failure of the next host. This failure is not worth writing *Error Message* to *Log File* since the failure is not happened on the local *Database Server*, rather it happens on the next *Database Server*. However, we can write the *Error Message* to another host. This situation will be discussed and solved in the coming subsection.

3.1.2.2. Logging in Launch Server

For *Launch Server*, the **Initialization** stage involves the following processes:

- Starting an *Agent Transporter*. The situation is similar to that of *Database Server*. If the *Launch Server* fails to start the *Agent Transporter*, it should write an *Error Message* on the *Log File* stating that it fails in the *Initialization Stage*, and the *Launch Server* needs to be restarted.

- Sending a public key to *DHKey Server*. It is the same case as the *Database Server*. If the *Launch Server* fails to send the key *DHKey Server*, this indicates that the server is down. This needs a

whole system recovery.

```
Mon Apr 17 05:46:34 CST 2000 >> [initialization] Public Key is sent to DHKeyServer
Mon Apr 17 05:46:34 CST 2000 >> [initialization] Initialization Done.
Mon Apr 17 05:50:05 CST 2000 >> [agent creation] An agent is created
Mon Apr 17 05:50:08 CST 2000 >> [agent creation] agent is launched
Mon Apr 17 05:51:50 CST 2000 >> [handle agent] received an agent
Mon Apr 17 05:51:51 CST 2000 >> [agent creation] An agent is created
Mon Apr 17 05:51:52 CST 2000 >> [agent creation] agent is launched
Mon Apr 17 05:54:40 CST 2000 >> [handle agent] received an agent
```

Figure 25. A Launch Server Log File.

The **Handle Agent** stages involves several processes:

- In an *Agent* creation process, it needs to send public key to *Key Server*. This may fail if the *Key Server* is down. This needs to write an *Error Message* to the *Log File* of the *Launch Server*. Also, this *Error Message* indicates that the whole system needs to be recovered. Also, the same situation happens when the *Agent* needs to retrieve the public key from *DHKey Server* in order to travel to the next host.

- When the *Agent* comes back to the *Launch Server*, the only point that it will fail is to get public keys from *Key Server* in order to handle the implemented security measures. If the *Key Server* fails, this will also needs a whole system recovery.

3.1.2.3. Logging in Key Server and DHKey Server

For *Key Server* and *DHKey Server*, their stages are different from the previous components. Although they still have the **initialization** stage, after that, they are waiting for request to store and retrieve keys. Therefore, the failure can only be reflected in the **initialization** stage. The failure of these servers will cause a whole system recovery.

After discussions of the *Logging System*, the data inside *Agent* is still not being protected. We cannot stop *Agent* from travelling to fail servers. We still need another mechanism to improve the reliability of the current design.

3.1.3. Connection Availability Detection

We introduce another mechanism to protect the data inside *Agent*. It is called **Connection Availability Detection**, **CAD**. By using **CAD**, we can successfully stop *Agent* from travelling to fail servers until those servers are being restarted.

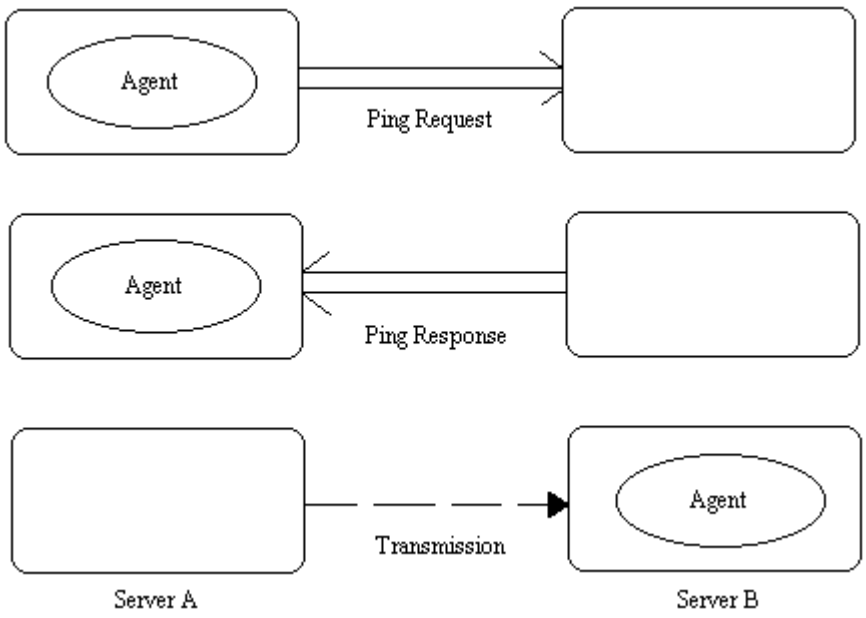


Figure 26. CAD, Connection Availability Detection.

The idea behind **CAD** is simple:

- Before an *Agent* leaves a host, it has to detect whether the connection to the next host is available or not. The method is analogous to the UNIX command **PING**. Before the *Agent* goes on, we *PING* the next host and wait for the next host to response.
- If next host is alive, it should response to the request by the *Agent*. Once the *Agent* receives such a reply, it is guarantee that the connection the next host is available. Then, the *Agent* can travel to the next host without data loss.
- If the next host is dead, it will not be able to reply such a request. In this case, we have come up with 2 solutions:

  1, The *Agent* should continue to send requests to the next host until the next host replies the message. This involves a **busy wait**. This wastes a lot of resources and is not efficient to do so.

  2, The *Agent* should give up sending requests to the next host. Instead of travelling to the next host in its itinerary, the *Agent* puts the current next host to be the last position in its itinerary, i.e. **skip that host**. Then, the *Agent* detects the connection availability of the new next host. This method is more efficient than the previous one.

- However, solution 2 brings about a fatal problem to our design: **it does not fit our security requirement**. As, in solution 2, the *Agent* is able to change its itinerary, this will violate the *Agent Tempering Detection* design. With this mechanism, the security measure is not able to distinguish whether the *Agent* suffers from a security attack or not because the itinerary is being

changed and the security requirement will report a failure by decrypting the *Encrypted_Itinerary* inside *Agent*. Although solution 1 has a hit on system performance, solution 2 violates the security requirement. Therefore, we choose solution 1 to be our **CAD** design.

With the **CAD** and **Logging System** working together, we can guarantee that the new SIAS design is a **Fault-Tolerance** design. However, the system still cannot prevent data loss in some situations.

3.1.4. Weakness in Fault-Tolerance Design

In our current design, we still cannot stop data loss in some failure situations.

- When the failures are happened in *Launch Server*, *Key Server* or *DHKey Server*, based on the previous analysis, the whole system needs to be recovered. All the processes that the *Agents* are running on will be terminated. Hence, a **total data loss** will be resulted. This kind of data loss cannot be avoided as only the recovery of the failed component will violate the security design and also the secure agent transmission design.

- When the failure occurs in the server, mainly *Database Server* and sometimes *Launch Server*, that the *Agent* resides, the *Agent* will be destroyed as its execution environment is destroyed also. This kind of data loss is also not avoidable.

Having discussed the whole fault-tolerance design, we need a new component to take the responsibility to restart the faulty component(s). In the coming section, we will discuss the implementation of the fault-tolerance design.

3.2. Implementation

We have introduced a new component called **Monitor Program** to handle the *Component Recovery* and *CAD, Connection Availability Detection*. The *Monitor Program* is written Perl, which is available in all the UNIX workstations.

A question is may be raised: *Why don't we use Java instead of Perl to implement the Monitor Program?*

The reason is that the whole design of SIAS is built on top of Java. It seems that Java is an obvious choice of the implementation of *Monitor Program*. However, what if the failure is not resulted in Concordia components, instead, it happens in the Java Virtual Machine. If our *Monitor Program* is also implemented by Java, it will fail too. Therefore, we choose a separate platform to implement the *Monitor Program*. The *Monitor Program* will fail only if the Perl interpreter fails, which is seldom happens.

3.2.1. Implementation of Logging System

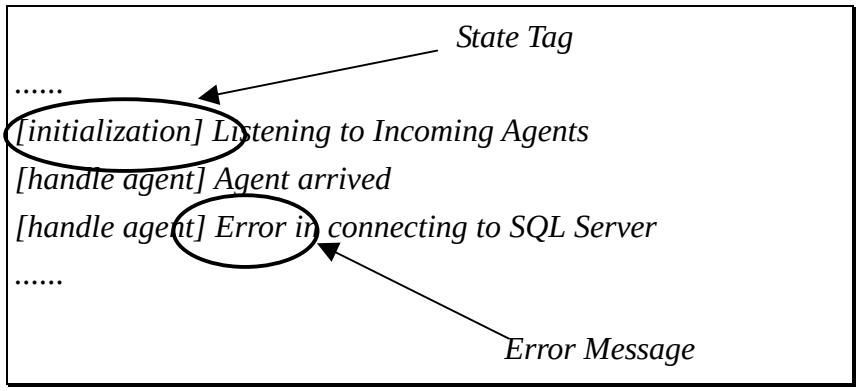


Figure 27. Log Analysis.

The *Logging System* is implemented in the *Monitor System*.

- First, the *Monitor Program* opens the *Log File*, and reads the *Log File* line by line. By reading the *Log File*, the *Monitor Program* knows what the current state of the component is by reading the **state tag** at the start of each line. The *Monitor Program* will look for *Error Message* throughout the whole *Log File*.

- If no *Error Message* is found, the *Monitor Program* will search for *Error Message* in *Log Files* of other components.

- If *Error Message* is found, the *Monitor Program* will send a **restart signal** to the target component. The target component will be killed and start running again. If the error needs a whole system recovery, the *Monitor Program* will send **restart signals** to all components in the system. After sending the restart signal(s), the *Monitor Program* will continue to search for *Error Message* in *Log Files*.

- A forever loop is used in *Monitor Program* to continuously check all the components in the system.

The **restart signal** is another Perl program. The **restart signal** is sent to the target machines by using **RSH**, which is a UNIX command. With the *Monitor Program* and the *restart signal program*, we can implements the *Logging System*.

3.2.2. Implementation of Connection Availability Detection

The implementation of *CAD* involves the modification of *Launch Server* and *Database Servers*. Also, the *Monitor Program* has also taken part in the *CAD* implementation.

3.2.2.1. Modification in Launch Server and Database Servers

In *Launch Server* and *Database Servers*, we have to add a code segment in order to **PING** the next destination. The **PING** program is written in Java by using the RMI. The idea is to check whether a *Logic Address, RMI Address,* is bind to a *Physical Address, IP Address,* or not.

If the *RMI Address* does not exist, the *PING* program will catch an exception. Then, the *Agent* should issue another *PING* request to the target machine until the *Monitor Program* restarts the server. If the *RMI Address* exists, the *PING* program will return true and the *Agent* can proceed to the next host.

With the *Logging System* as well as the *CAD* implementation inside *Launch Server* and *Database Servers*, the *Fault-Tolerance* implementation is completed. However, we can enhance the efficiency of the current *Fault-Tolerance* by extra implementation in *Monitor Program*.

3.2.2.2. Modification in Monitor Program

In our current implementation, whenever the *CAD* mechanism detects a broken connection, the *Agent* has to wait until the *Monitor Program* restarts the target component. However, we can have the detection to happen before the *Agent* does such job. The trick is to put the *CAD* mechanism inside the *Monitor Program*.

Whenever the *Monitor Program* is inspecting the *Log File* of a server, the *Monitor Program* can also *PING* that server. If the disconnection cannot be reflected in the *Log File*, we can still uses *CAD* mechanism to recover such failures.

3.2.3. Flow Description of Monitor Program

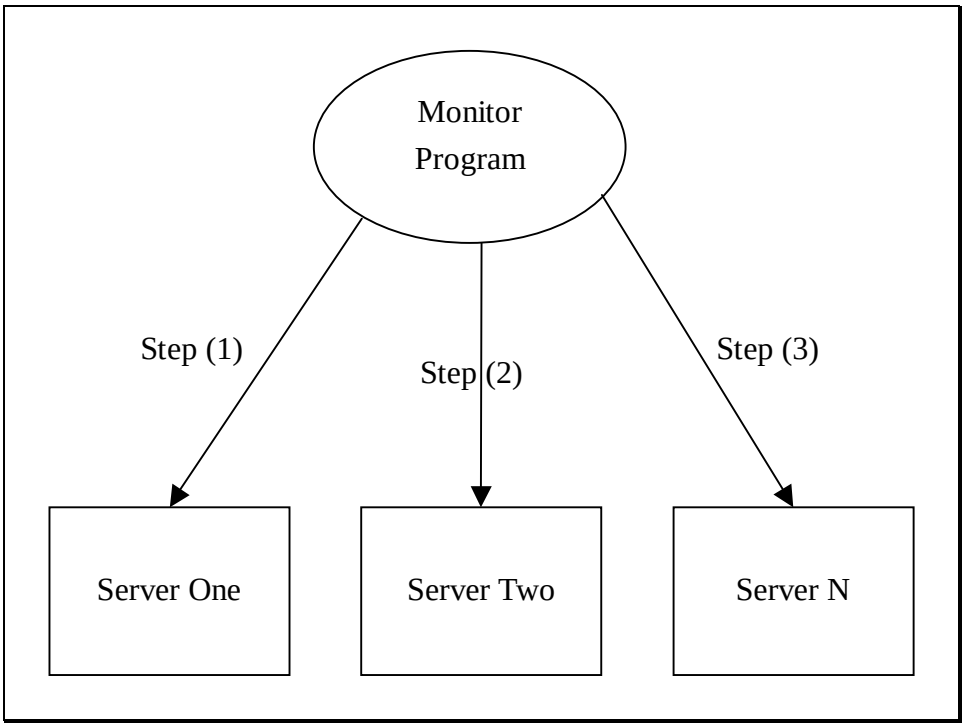


Figure 28. Flow Description of Monitor Program.

- Step (1)

  When the *Monitor Program* starts, it first inspects Server One, it may be a *Launch Server, Key Server, DHKey Server or Database Server*. It inspects the *Log File* of Server One in order to search for *Error Messages*. If *Error Message* is found, it will send a restart signal to Server One. Then, it carries out *CAD, Connection Availability Detection,* by the *Ping Program*. If Server One fails in *CAD,* the *Monitor Program* will send restart signal to Server One.
- Step (2) repeats the processes in Step (1).
- The same step is done throughout all the servers in the network.
- Let Server N is the last server in the network. After Step (3) has finished the processes, it goes back to Step (1). This cycle will go on forever until the *Monitor Program* dies.

4. Evaluation of Secure SIAS

In this section, we evaluate the security design we implemented. There are two aspects to evaluate. First, we analyze the security provided to SIAS by the additional measures. Then, we measure the performance overhead introduced to the system by such measures.

4.1. Security Analysis

The security of the additional measures lies mainly on the introduction of the *Key Server* that facilitates the use of public key cryptography. Assuming the *Key Server* and the communication channel with it is secure enough, which can be justified by the Secure Socket Layer, the closed network we want can be built effectively.

Furthermore, if the keys of *Agents* are managed properly, the prevention of modification of the encrypted product and quantity lists of an *Agent* by a malicious host is supported by the RSA encryption algorithm, of which the difficulty to break is equivalent to the factoring problem. The time complexity for breaking the system depends on the length of the key in number of bits. The longer the key is, the more secure would be the system. In our implementation, we have chosen a key length of 128 bits. This would be sufficiently secure for domestic purpose.

Similarly, a malicious host would understand to modify the encrypted query results collected by an *Agent* from another host at the same complexity. Therefore, integrity of queries, and confidentiality and integrity of query results can be achieved by prevention of tampering.

For the detection of modification to itinerary of an agent by a malicious host, suppose there is only a single malicious host, out of N hosts, that wants to modify the itinerary of an agent. Since the encrypted itineraries are chained together, the malicious host would need to fake all the (N-1) encrypted itineraries from other hosts to avoid being detected, which would be too complex to an ordinary attacker. Therefore, the itinerary of the agent can be assured, and authenticity achieved.

However, as mentioned before, there do exist other attacks that we have not considered completely, such as replaying attacks, timing attacks, and repeated cipher-text attacks.

4.2. Performance Measurements

We have tested the times for SIAS to launch a single agent before and after implementation of the security mechanisms described in Section 6. Round trip times (RTTs) required for an agent to travel around an electronic market, consisting of 26 hosts, are measured under different situations. We have chosen 26 Sun SPARC workstations, listed in Figure 29. They have similar hardware configuration, so the overhead introduced by each machine is more or less the same.

137.189.88.173 (sparc73)
137.189.88.174 (sparc74)
137.189.88.175 (sparc75)
137.189.88.176 (sparc76)
137.189.88.177 (sparc77)
137.189.88.181 (sparc81)
137.189.88.182 (sparc82)
137.189.88.183 (sparc83)
137.189.88.184 (sparc84)
137.189.88.185 (sparc85)
137.189.88.186 (sparc86)
137.189.88.187 (sparc87)
137.189.88.188 (sparc88)
137.189.88.189 (sparc89)
137.189.88.190 (sparc90)
137.189.88.191 (sparc91)
137.189.88.211 (hpc1)
137.189.88.212 (hpc2)
137.189.88.213 (hpc3)
137.189.88.214 (hpc4)
137.189.88.215 (hpc5)
137.189.88.216 (hpc6)
137.189.88.217 (hpc7)
137.189.88.218 (hpc8)
137.189.88.219 (hpc9)
137.189.88.220 (hpc10)

Figure 29. Database Server Locations

To evaluate the performance overhead introduced, we have tested the times for SIAS to launch a single agent with and without security measures. Round trip times (RTTs) required for an agent to travel around an electronic market of different number of hosts, with and without security enforcement, are measured respectively. Queries of different sizes (number of product items) have been tested. The results are plotted in Figures 30 (without security) and 31 (with security) below.

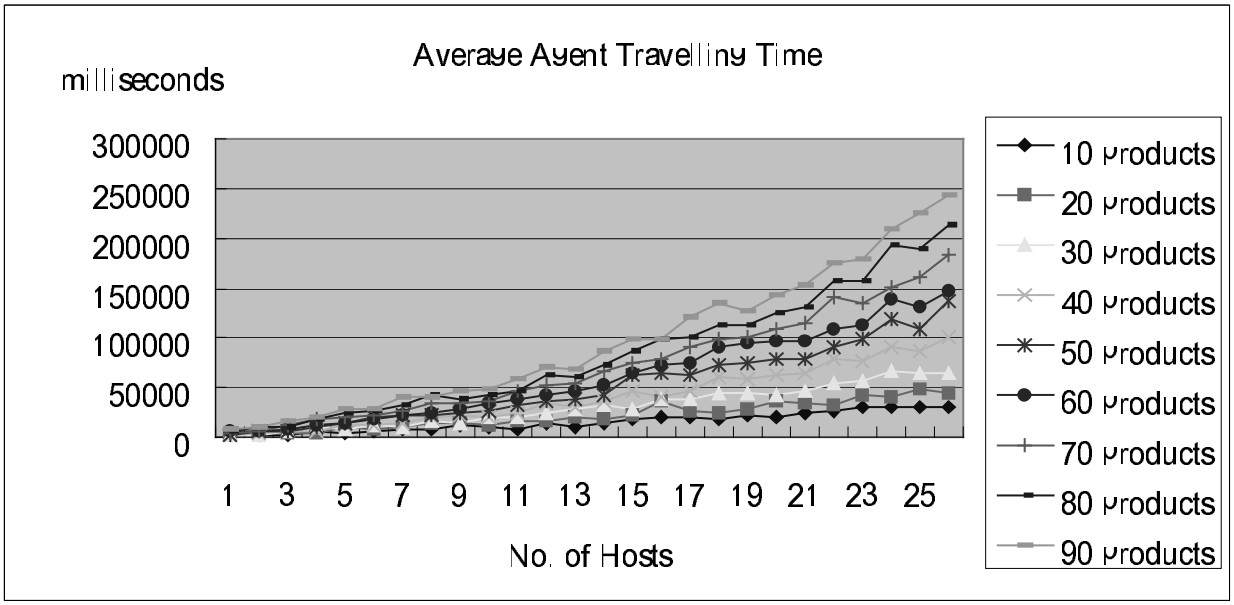


Figure 30. Average Agent Travelling Time (without Security).

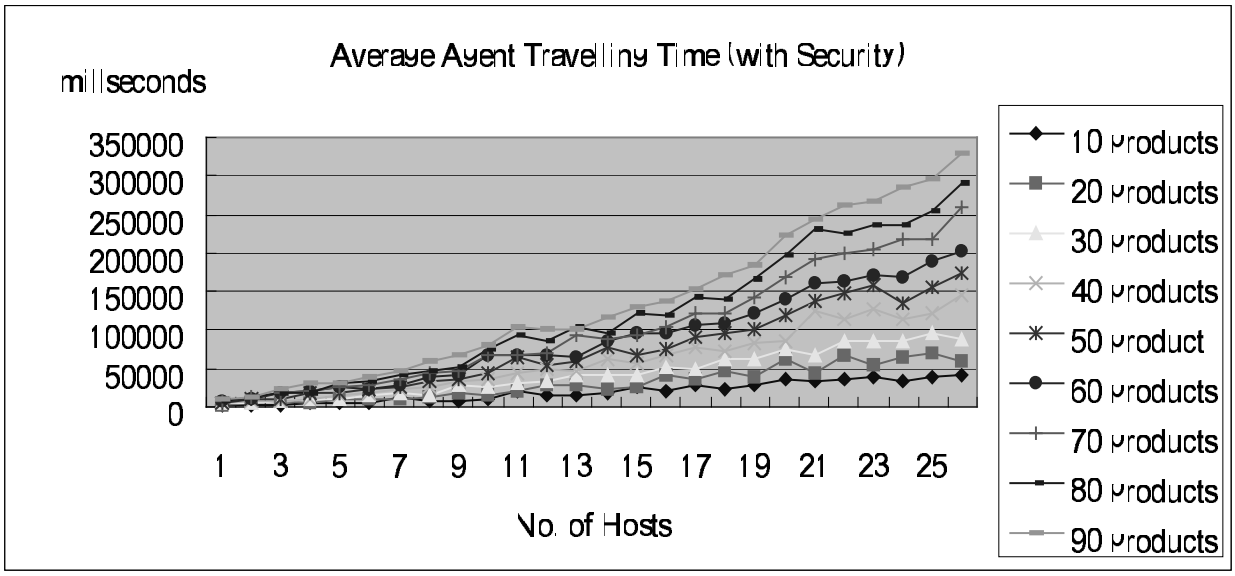


Figure 31. Average Agent Travelling Time (with Security).

Results show that, the RTT for an agent to travel in SIAS changes more or less linearly over the number of hosts in the system. This is due to the additional time to travel an additional host, and the overhead for each additional host is more or less the same. Moreover, the RTT is also linearly increasing as the number of products of the query increases. This can be explained by the increases in number of database transactions and time to transport an agent. When security is enforced, the RTT increases in general. For the maximum number of hosts of 26, and maximum size of query of 90 products, the RTT increases by 100 seconds, from 230 seconds to 350 seconds. This can be explained by the extensive use of the RSA algorithm to encrypt and decrypt each item, which is time consuming, especially when the key is long. Therefore, we see a trade-off between security and performance in SIAS.

Conclusion

We studied the technology of autonomous mobile agents and discussed the problem of malicious hosts in a mobile agent system. We implemented SIAS as a sample application of mobile agents, which reduces communication cost and allows delegation of tasks. We addressed some security problems of malicious hosts in SIAS, and developed a primitive approach to protect the agents. We analyzed the security of our approach, and believe it is strong enough for domestic purpose. We measured the performance overhead of the security measures, saw a trade-off between performance and security for SIAS, and learned that it takes time for a malicious host to attack an agent. We analyzed the reliability of SIAS and implement a fault-tolerance design of SIAS. We believe that mobile agent technology will be a new trend in electronic commerce technology.

Acknowledgement

We would like to express our gratitude to Michael Rung Tsong LYU, our project supervisor. He has provided many valuable suggestions and comments to us throughout this project. We are much appreciated by his patience and kindness in advising us.
Moreover, we would like to thanks Anthoy, H.W. Chan, CSE/CUHK M.Phil Year 2 student, which helps us a lot in discussing the security of mobile agent.

Reference

[1] Danny B. Lange and Mitsuru Oshima, “Programming and Deploying Java(TM) Mobile Agents with Aglets(TM)”.

[2] IBM Aglets Software Development Kit Home Page, URL: http://www.trl.ibm.co.jp/aglets/

[3] Concordia - Java Mobile Agent Technology, URL: http://www.meitca.com/HSL/Projects/Concordia/

[4] ObjectSpace Voyager, URL: http://www.objectspace.com/products/prodVoyager.asp

[5] Java(TM) Cryptography Extension, URL: http://java.sun.com/products/jce/index.html

[6] Java(TM) Remote Method Invocation (RMI), URL: http://java.sun.com/products/jdk/1.3/docs/guide/rmi/index.html

[7] Fritz Hohl, "A Model of Attacks of Malicious Hosts Against Mobile Agents", 4th Workshop on Mobile Object Systems (MOS'98): Secure Internet Mobile Computations

[8] "SIAS: A Secure Shopping Information Agent System", Anthony H. W. Chan, T. Y. Wong, Caris K. M. Wong, and Michael R. Lyu accepted by Fourth International Conferences on Autonomous Agents (AGENTS 2000), Spain, June 3-7, 2000

[9] “SIAS: A Secure Shopping Information Agent System”, Anthony H. W. Chan, T. Y. Wong, Caris K. M. Wong, and Michael R. Lyu accepted by the 15th International Conference on Information Security, Beijing, China,

[10] “Design, Implementation, and Experimentation on Mobile Agent Security for Electronic Commerce Applications” Anthony H. W. Chan, T. Y. Wong, Caris K. M. Wong, and Michael R. Lyu accepted by The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, June 26 - 29, 2000 Monte Carlo Resort, Las Vegas, Nevada, USA

Appendix

A Statistical Comparison of different designs of SIAS

Primary Design

Module Name	Lines of Code
Agent	318
Database Server	134
Launch Server	186
Client Program	533

Total number of lines of code = 1171

Secure Design

Module Name	Lines of Code
Agent	344
Database Server	208
Launch Server	432
Client Program	585
RSA	276
Key Server	198
Simulation of Malicious Host	204

Total number of lines of code = 2247

Secure Agent Transmission Design

Module Name	Lines of Code
Agent	344
Database Server	325
Launch Server	553
Client Program	791
DHKey	183
DHKey Server	184
Envelop Agent	54

Total number of lines of code = 2380

Fault Tolerance Design

Module Name	Lines of Code
Agent	344
Database Server	325
Launch Server	553
Client Program	791
Monitor Program	143
Ping Program	29
Restart Program	96

Total number of lines of code = 2185

Final Design (Basic Design + Secure Design + Secure Agent Transmission + Fault Tolerance Design, with optimization of code)

Module Name	Lines of Code
Agent	280
Database Server	334
Launch Server	677
Client Program	761
RSA	278
Key Server	117
Simulation of Malicious Host	403
Envelop Agent	59
DHKey	179
DHKey Server	108
Monitor Program	143
Ping Program	29
Restart Program	96

Total number of lines of code = 3464