ONEPORT RMI: RMI PROTECTING INTEGRITY AND CONFIDENTIALITY FOR MOBILE AGENTS

Kazunari MEGURO
The Graduate School of Engineering
Tottori University
4-101, Koyama-Minami
Tottori, JAPAN
email: meguro@tottori-u.ac.jp

Shinichi MOTOMRUA, Takao KAWAMURA and Kazunori SUGAHARA
Information Media Center, Faculty of Engineering
Tottori University
4-101, Koyama-Minami
Tottori, JAPAN
email: motomura@tottori-u.ac.jp, {kawamura, sugahara}@ike.tottori-u.ac.jp

ABSTRACT
In this paper, a new implementation of RMI named OnePort RMI is proposed. OnePort RMI consists of new RMI runtime, classes which are implemented interfaces by RMI specification and MultiChannelSocketFactory. Using OnePort RMI, when an object on a client invokes methods of remote objects on a server, the client can use sockets of different types to connect one destination port at the same time, and the server can accept incoming call from the sockets on only the port. In order to protect integrity and confidentiality of our mobile agent framework named Maglog, OnePort RMI is introduced into Maglog. As a result, each agent can select a socket depending on importance of data and programs which are contained in their agents. We emphasize that the proposed OnePort RMI is not only for mobile agent frameworks such as our Maglog but also for any RMI applications.

KEY WORDS
Security and Reliability, RMI, Integrity, Confidentiality, Mobile Agents

1 Introduction

In the construction of network application systems, distributed models are widely adopted. In particular, mobile agent technologies are attracting attention as a key technology for developing distributed systems. Several mobile agent frameworks have been proposed, such as Aglets[1], Jinni[2], Mobilespaces[3], and Telescript[4]. Measures against security threats for mobile agent frameworks have been studied[5][6][7]. The security measures can be classified by following aspects:

Authentication is a process which identifies agents. Authentication is necessary for the below aspects.

Authorization is a process to determine what types of grants an agent has. A computer must be protected from malicious agents, and an agent must be protected from malicious computers and agents.

Integrity means the property that agents have not been altered or destroyed in an unauthorized manner. Agents may be tampered with while agents are on computers and migrates to other computers.

Confidentiality means the property that agents are not made available or disclosed in an unauthorized manner. Data and programs which are contained in agents must be accessible only to agents which are authorized to have access.

In this paper, we concentrate our discussions on integrity and confidentiality when agents migrate across a network. Generally, secure channels between computers via encryption of network packets are used to protect integrity and confidentiality. SSL (Secure Sockets Layer) and IPSec (Security Architecture for Internet Protocol) are illustrative examples of such secure channels. If a secure channel is used in mobile agent frameworks, all network packets are encrypted while agents migrate. However, an encryption process consumes a lot of resources and causes performance degradation. Therefore, selective encryptions of agents are preferable, i.e., considering costs of encrypting, some agents have to be encrypted and others do not. In mobile agent frameworks, every agent should be able to select a communication channel, i.e., secure type or not secure type. Moreover, each agents should select different secure channel so that encryption strength can be selected according to the importance of agents.

We consider that the above manner for using secure channels is introduced into mobile agent frameworks which are implemented in a Java environment. Because, most mobile agent frameworks, such as the above mentioned Aglets, Jinni and Mobilespaces, have been implemented in a Java environment. And these mobile agent frameworks use Java Remote Method Invocation (hereafter referred to as RMI)[8] or XML-RPC[9] as transport mechanisms. In mobile agent frameworks based on RMI, when several channels are used at the same time, the same number of sockets are required, and as the result, the same number of ports are required. In most networks, a firewall is used to prevent unauthorized access to a network, therefore the number of open ports are limited to be minimum. Therefore, it is necessary that one port can be associated with multiple sockets. However, Sun's implementation of RMI cannot realize the requirement mentioned above. For

this reason, we propose new implementation of RMI named OnePort RMI that multiple sockets can be associated with one port. On the other hand, XML-RPC uses HTTP as the transport protocol. Therefore, we build an HTTP server and an HTTP client with MultiChannelSocketFactory is used in the inside of OnePort RMI so that the HTTP server can handle multiple sockets on one port.

To confirm their behaviors, we implement OnePort RMI and our HTTP client/server on our mobile agent framework Maglog[10][11]. However, we emphasize that the proposed OnePort RMI is not only for mobile agent frameworks such as our Maglog but also for any RMI applications.

2 Why we cannot use Sun's implementation of RMI?

In this section, the behavior of a client communicating with a server using RMI is described. And, it is explained that Sun's implementation of RMI cannot realize the behavior of RMI which we require.

2.1 Behavior of RMI

RMI enables programmers to create distributed Java technology-based on Java technology-based applications, in which the methods of remote Java objects can be invoked from other JVM on different hosts. When an object on a client tries to invoke a method of a remote object on a server, the object communicates with the stub object on client's JVM which is corresponding with the remote object. A stub object is client's proxy for remote objects, and its roles are to hide network connections and serialization of parameters. A stub object has an RMIClientSocketFactory object which creates a socket to communicate with a server. A skeleton object which is corresponding with a remote object is on server's JVM, and the skeleton object invokes methods of the remote object in effect. Roles of skeleton objects are to hide network connections and deserialization of parameters. Stub objects and skeleton objects are managed by an RMI runtime on each of the JVMs, moreover the objects are called automatically by each RMI runtime if necessary. An RMI runtime has an RMIServerSocketFactory object which creates a server socket to wait for incoming calls from clients. RMIClientSocketFactory and RMIServerSocketFactory are provided by java's core library. Figure 1 shows a model of relations among an RMI runtime, a stub object and a skeleton object.

A server executes the following steps to export a remote object so that an object on a client can invoke methods of the remote object on the server.

1. An RMIClientSocketFactory object and an RMIServerSocketFactory object are created.
2. A stub object and a skeleton object are generated using above objects and the port number which is used to wait for incoming calls from clients.
3. The stub object and the skeleton object are registered in server's RMI runtime.
4. The stub object is registered in server's RMI registry which allows remote objects on the server to register themselves as available to objects on the client.

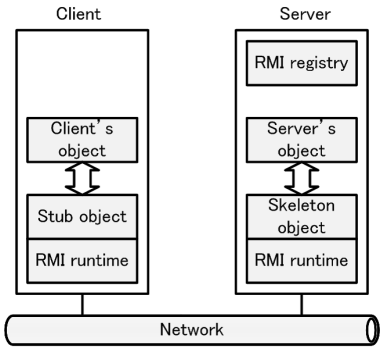

Figure 1. A model of relation among an RMI runtime, a stub object, and a skeleton object when a object on a client invokes a method of a remote objects on a server.

When an object on a client tries to invoke a method of a remote object on a server, the object gets the stub object which is corresponding with the remote object from server's RMI registry. After that, the object invokes the method for the stub object. Client's RMI runtime creates a socket by the RMIClientSocketFactory object which is contained in the stub object. Next, client's RMI runtime communicates with server's RMI runtime by the socket. Furthermore, server's RMI runtime creates a server socket by the RMIServerSocketFactory object which is registered in server's RMI runtime, after that the server socket communicates with the socket.

2.2 The reason that we cannot use Sun's implementation of RMI

The behaviors of RMI which we require are that a client can use sockets of different types at the same time and a server can accept incoming call from the sockets on only one port. In order to realize the behaviors, the following mechanisms are necessary.

1. An RMIServerSocketFactory object must be able to create multiple server sockets which are corresponding with client's sockets.
2. A client must be able to select sockets from different types which are created by an RMIClientSocketFactory object.

In order to implement first mechanism, behaviors of RMIServerSocketFactory and RMIClientSocketFactory are customized. And, it is necessary to solve either of the following two problems to realize the second mechanism.

1. An RMIClientSocketFactory object is created by a server, after that when a client tries to use the object, the object is managed by the RMI runtime on the client. Therefore, an object on the client cannot invoke methods of the RMIClientSocketFactory object. Namely, the client cannot create sockets of different types by the RMIClientSocketFactory object.

2. A server exports an object by using a set of an RMIClientSocketFactory object, an RMIServerSocketFactory object and a port. If a server exports an object by using pairs of multiple RMIClientSocketFactory objects and the same port, a client selects requiring RMIClientSocketFactory object.

It is impossible to solve the first problem since RMI specification does not define the manner to access an RMI runtime. Moreover, Sun's implementation of RMI does not provide the manner to solve the second problem.

3 OnePort RMI

In section 2, we mentioned about the reason why a client cannot use sockets of different types at the same time by using Sun's implementation of RMI. Therefore, we develop new implementation of RMI named OnePort RMI. It consists of MultiChannelSocketFactory, new RMI runtime and classes which are implemented interfaces defined by RMI specification. In the following sections, first, MultiChannelSocketFactory is described, after that proposed RMI runtime is described.

3.1 MultiChannel Socket Factory

In a system which uses Sun's implementation of RMI, when a server receives a request from a client, the server creates a ServerSocket object by an RMIServerSocketFactory object. ServerSocket class is provided by java's core library. Next, the ServerSocket object creates a server socket, after that the server socket waits for incoming calls. In order to create multiple server sockets which are corresponding with connecting sockets, steps of above execution must be changed as follows:

1. A server socket which is created by a ServerSocket object receives the type of a socket from a client.

2. Another server socket which is corresponding with the type of the socket is created, after that the server socket communicates with client's socket.

For sending and receiving the type of a socket, MultiChannelClientSocketFactory class and MultiChannelServerSocketFactory class are developed. MultiChannelClientSocketFactory class implements RMIClientSocketFactory interface, and MultiChannelServerSocketFactory class implements RMIServerSocketFactory interface. Furthermore, MultiChannelServerSocket class is developed so that a server socket which is corresponding with the type of a socket is created. Figure 2 shows a relation of above classes. The above classes are named as MultiChannelSocketFactory.

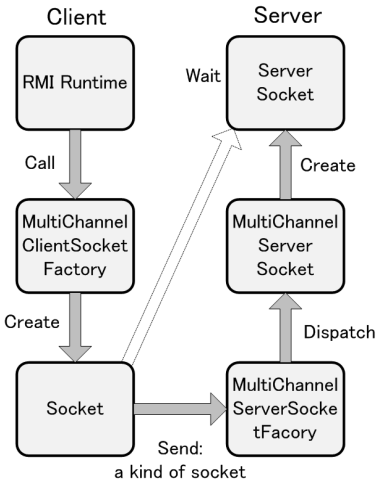

Figure 2. A relation of classes which are contained in MultiChannelSocketFactory.

3.2 Proposed RMI runtime

In order to export remote objects in which identical port number is associated with multiple RMIClientSocketFactory, we develop new RMI runtime. Our RMI runtime is based on an Object Request Broker (hereafter referred to as ORB) which we have developed. The main components of our ORB are described as follows:

ORBServer plays a server role in our RMI runtime. It accepts requests from server sockets.

ORBClient plays a client role in our RMI runtime. When an object on a client invokes methods of a remote object on a server, ORBClient communicates with the server instead of the object.

ORBStubFactory generates a stub object, a skeleton object, and a reference which are corresponding with a remote object.

ORBDirectory provides the following two functions. The first is the function to register objects which are generated by ORBStubFactory. The second is the function to search the registered objects.

In our ORB, the following steps are executed when a server exports a remote object. First, ORBServer receives an RMIServerSocketFactory object. Next, ORBStubFactory generates a stub object using a pair of an RMIClientSocketFactory and a port. Furthermore, ORBServer creates an unique identifier which is corresponding with the stub object. Finally, ORBServer registers the identifier and the

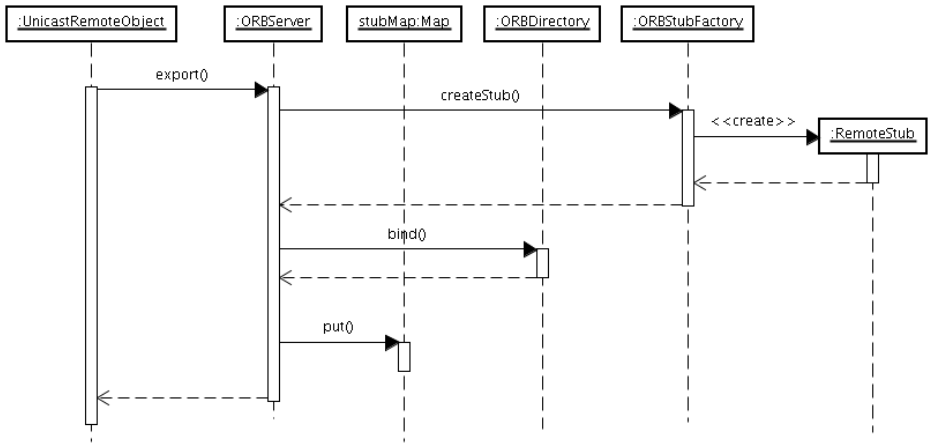

Figure 3. A UML sequence diagram of which a server exports a remote object using our RMI runtime.

stub object in ORBDirectory. In consequence of the above steps, a stub object can be generated using a pair of the other RMIClientSocketFactory object and the same port and be registered in ORBDirectory. Namely, OnePort RMI can realize that a server exports a remote object using pairs of multiple RMIClientSocketFactory objects and the same port. Figure 3 shows a UML sequence diagram of which a server exports a remote object using our RMI runtime. The UnicastRemoteObject class has a static method named export for exporting a remote object. The UnicastRemoteObject class is defined by RMI Specification.

In practice, MultiChannelClientSocketFactory class and MultiChannelServerSocketFactory class are used instead of RMIClientSocketFactory class and RMIServerSocketFactory class.

4 Implementation

OnePort RMI implements interfaces defined by RMI specification, and it includes above RMI runtime. Figure 4 shows an overview of their classes.

4.1 Secure Channels

We develop the following two secure channels into MultiChannelSocketFactory. One is named DES_Channel in which a socket is encrypted using the Data Encryption Standard (hereafter referred to as DES) which is a cryptographic algorithm. The other is SSL_Channel in which a socket is implemented using SSL. SSL_Channel has the following security measures, therefore its security is stronger than DES_Channel. On the other hand, DES_Channel is not necessary to have a digital certification which is needed by SSL_Channel, therefore DES_Channel provides simple manner for utilizing.

Endpoint authentication Two computer's identities can be authenticated using asymmetric cryptography such as Public Key Infrastructure.

Integrity checking Message transport includes a message integrity check using a keyed message authentication code.

Key exchange A symmetric cipher which is used for encryption is exchanged between computers on periodic basis.

DES_Channel is realized by DESSocket class and DESServerSocket class which extend Socket class and ServerSocket class and implement DES encryption. SSL_Channel is realized by SSLSocket class and SSLServerSocket class which are provided by Java Secure Socket Extension. The socket which is not encrypted is defined as RAW_Channel.

4.2 Applying to Mobile Agent Framework

We have proposed a mobile agent framework named Maglog which is based on Prolog and is implemented in a Java environment. In Maglog, the following predicate is introduced so that each agent can select a channel.

```
change_channel(PrevChannel,NewChannel)
```

After an agent is executed the above predicate, the agent uses `NewChannel` to migrate to other computers. A kind of channels before changing is bound to `PrevChannel`. The following three channels are defined.

1. RAW: Above RAW_Channel.
2. DES: Above DES_Channel.
3. SSL: Above SSL_Channel.

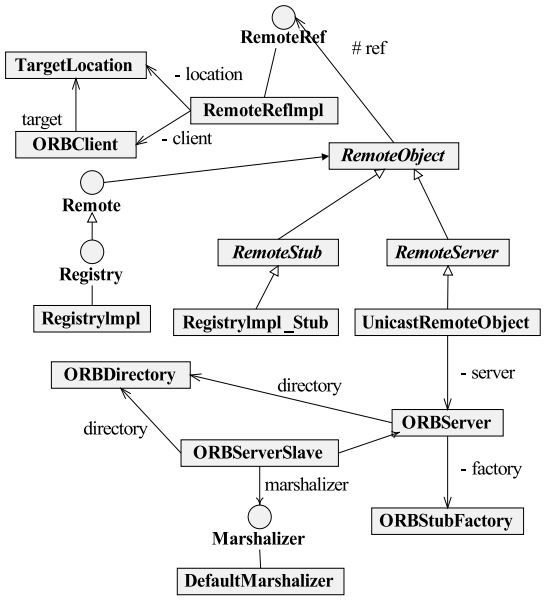

Figure 4. A UML diagram which is an overview of classes of OnePort RMI.

5 Experiments

In this section, sample codes using OnePort RMI are shown. And the experimental results for comparison of the round trip time between OnePort RMI and Sun's implementation of RMI are shown.

5.1 Sample code

Figure 5 shows a part of a sample code when a server provides HelloImpl objects using DES_Channel and SSL_Channel for clients. In this code, HelloImpl objects are created, and the objects are exported by using DES_Channel and using SSL_Channel. After that the objects are registered in server's RMI registry with names of "//server/HelloDES" and "//server/HelloSSL".

Figure 6 shows a part of a sample code when a client invokes the HelloImpl objects on the server using DES_Channel and SSL_Channel. First, the client takes stub objects from server's RMI registry by invoking lookup method with the above names. Next, the client invokes a method of the stub objects. Incidentally, HelloImpl class implements Hello interface.

5.2 Comparison of round trip time between OnePort RMI and Sun's implementation of RMI

This section shows the experimental results for comparison of the round trip time for a remote method invocation between OnePort RMI and Sun's implementation of RMI. In the experiments, two PCs are connected via a 100Base-T network. Under each implementation, the experiments

```
HelloImpl server_des  = new HelloImpl();
HelloImpl server_ssl  = new HelloImpl();

UnicastRemoteObject.exportObject(server_des,
  new MultiChannelClientSocketFactory('des'),
  new MultiChannelServerSocketFactory());

UnicastRemoteObject.exportObject(server_raw,
  new MultiChannelClientSocketFactory('ssl'),
  new MultiChannelServerSocketFactory());

Registory reg =
  LocateRegistry.createRegistry(REGISTRY_PORT);

reg.bind("//server/HelloDES",server_des);
reg.bind("//server/HelloSSL",server_ssl);
```

Figure 5. HelloImpl objects are provided using DES_Channel and SSL_Channel for clients.

```
Hello server_DES  =  null;
Hello server _SSL =  null;

Registory reg =
  LocateRegistry.getRegistry
    ("server",REGISTRY_PORT);

server_DES = (Hello)reg.lookup("//server/HelloDES");
server_DES.exec();

server_SSL = (Hello)reg.lookup("//server/HelloSSL");
server_SSL.exec();
```

Figure 6. A client invokes the HelloImpl objects using DES_Channel and SSL_Channel.

are performed 100 times using three channels in the following condition. A client invokes a method of a remote object on a server with an argument of a byte array. The data sizes of the argument are 1KB, 5KB, 10KB, 50KB, 100KB, 500KB, and 1000KB. The total times are shown in Figs. 7, 8 and 9. The differences of the round trip time between OnePort RMI and Sun's implementation of RMI are small at all channels. We can confirm the execution speed of OnePort RMI is practical.

6 Conclusion

We have developed new implementation of RMI named OnePort RMI. OnePort RMI consists of new RMI runtime, classes which are implemented RMI specification and MultiChannelSocketFactory. Using OnePort RMI, when an object on a client invokes methods of remote objects on a server, the client can use sockets of different types to connect one destination port at the same time, and the server can accept incoming calls from the sockets on only the port. We have developed two secure channels such as DES_Channel and SSL_Channel into MultiChannelSocketFactory. When other secure channels are needed, they can be added to MultiChannelSocketFactory easily.

Note that though we have confirmed the effectiveness of OnePort RMI on our mobile agent framework Maglog,

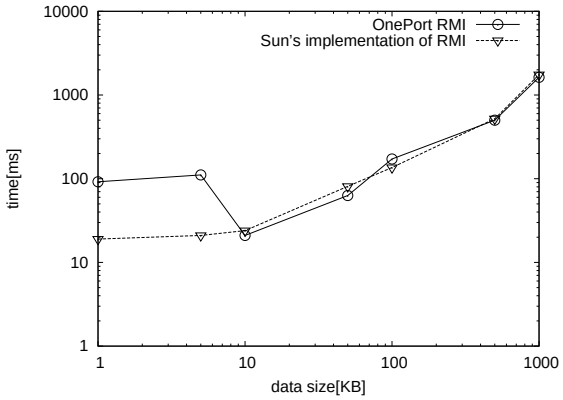

Figure 7. Comparison of the round trip time between OnePort RMI and Sun's implementation of RMI when RAW_Channel is used.

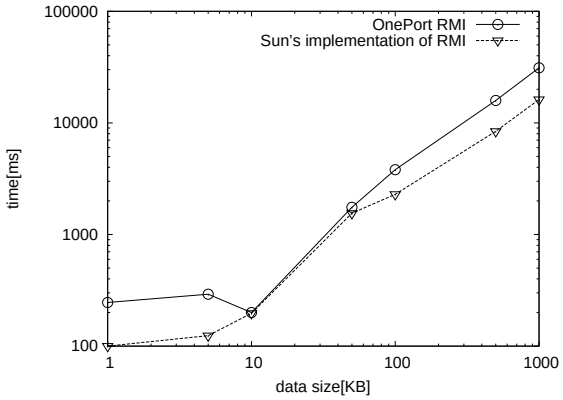

Figure 9. Comparison of the round trip time between OnePort RMI and Sun's implementation of RMI when SSL_Channel is used.

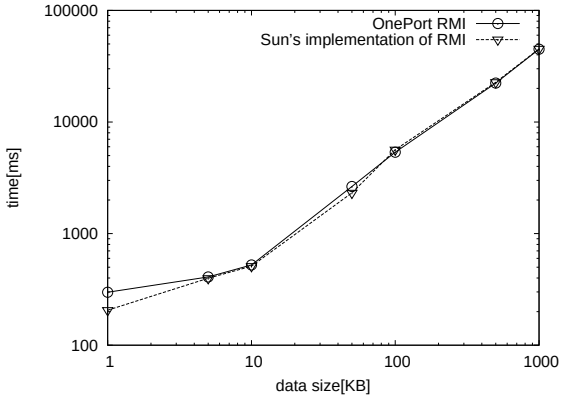

Figure 8. Comparison of the round trip time between OnePort RMI and Sun's implementation of RMI when DES_Channel is used.

OnePort RMI can be utilized by any RMI applications.

In this stage, though OnePort RMI has all necessary functions to handle multiple sockets on one port, it is not fully compatible with RMI specifications. For example, OnePort RMI lacks a distributed garbage collector or configuration properties. They will be implemented in future work.

References

[1]Lange, D.B. and Oshima, M.: *Programming and Deploying Java Mobile Agents with Aglets*, Addison Wesley (1998).

[2]Tarau, P.: Inference and Computation Mobility with Jinni, *The Logic Programming Paradigm: a 25 Year Perspective* (Apt, K., Marek, V. and Truszczynski, M., eds.), Springer, pp.33-48 (1999).

[3]Satoh, I.: MobileSpaces: A Framework for Building Adaptive Distributed Applications using a Hierarchical Mobile Agent System, *Proceedings of IEEE International Conference on Distributed Computing Systems*, IEEE Press, pp.161-168 (2000).

[4]White, J.E.: *Telescript Technology: The Foundation for the Electronic Marketplace*, General Magic (1994). http://www.genmagic.com/WhitePapers.

[5]Karjoth, G., Lange, D.B. and Oshima., M.: A Security Model for Aglets, *IEEE Internet Computing*, Vol.01, No.4, pp.68-77 (1997).

[6]Farmer, W., Guttman, J. and Swarup, V.: Security for Mobile Agents: Issues and Requirements, *Proc. 19th Nat'l Information Systems Security Conf. (NISSC 96)*, pp.591-597 (1996).

[7]Tardo, J. and Valente, L.: Mobile Agent Security and Telescript, *Compcon '96. 'Technologies for the Information Superhighway' Digest of Papers*, pp.58-63 (1996).

[8]Sun Microsystems: Java Remote Method Invocation, Web page (1997). http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmi-title.html

[9]Winer, D.: XML-RPC Specification, http://xmlrcp.com/spec (1998).

[10]Motomura, S., Kawamura, T. and Sugahara, K.: Logic-Based Mobile Agent Framework with Concept of Field, *IPSJ Journal*, Vol.47, No.4 (2006).

[11]Motomura, S., Kawamura, T. and Sugahara, K.: A Logic-Based Mobile Agent Framework for WEB Applications, *Proceedings of the 2nd International Conference on Web Information Systems and Technologies*, pp.121-126 (2006). Setubal, Portugal.