Security Service Design for the RMI Distributed System based on Parameterized RBAC

NEZAR NASSR, ERIC STEEGMANS

***Abstract*—Incremental growth in computing has enabled businesses to distribute their computing environments. In consequence, an increasing number of threats challenge distributed applications. Remote Method Invocation (RMI) is a distributed systems platform used to implement distributed applications, that is vulnerable to an increasing number of security threats. Several paradigms for protecting security of software systems have emerged. Role Based Access Control (RBAC) is a paradigm used for controlling users access on software systems.**

RMI has already now been used in many enterprise business applications. However, no security service has been concretely implemented as of yet. Thus, our focus was on designing and implementing a security infrastructure for RMI to address the security vulnerable issues that may arise in distributed systems developed using RMI. We introduced a new secure architecture for the Java RMI that employs the concepts of authentication and authorization based on Parameterized Role Based Access Control (PRBAC), which enables application developers to employ these concepts in distributed applications implemented using our proposed secure RMI.

***Index Terms*— distributed systems, remote method invocation, role-based access control, security.**

I. INTRODUCTION

Distributed systems security is of paramount importance for software systems constructed by using this technology. As a consequence, much research has been focusing on defining security standards and architectures for distributed systems middleware. In fact, there were many security standards and architectures defined for distributed systems middleware over the past years such as the CORBA security server, DCE security, Web Services Security and etc. However, there were no security standards or infrastructures defined for the Java Remote Method Invocation (RMI). Such an infrastructure could facilitate the design and development of secure distributed applications implemented using the Java RMI. More so, it reduces the security inconsistencies that might arise in developed distributed applications.

Much attention has been devoted recently to security issues and it is apparent that a high level of security is a fundamental prerequisite for Internet-based transactions, especially in the business-to-business area [1].

Distributed business applications require to be protected in terms of security to prevent access to confidential information. Moreover, they demand a way of regulating the user access to the system. A business application must determine who can access the system as well as how a user can access the system. e.g. a user in an organization that has access to an application must access only data he is entitled to see.

Java RMI is a programming technology that is used for developing distributed applications. It provides facilities for invoking methods on remote objects. The secure version of RMI (RMI-SSL) provides point to point security between clients and servers. RMI offers no security at the objects nor methods invocation levels .

The review of related research has shown that no work has been done towards defining a security infrastructure for RMI. However, there has been a vast area amount of work done on defining security infrastructures for other distributed systems middleware such as Web Services Security and the CORBA Security Server. Thus, our goal was designing and implementing a security infrastructure for RMI to address the security vulnerability issues that might arise in distributed systems developed using RMI. Our work presents a new secure architecture for Java RMI that employs the concepts of authentication and authorization based on role based access control (RBAC), which enables application developers to employ these concepts in distributed applications implemented using our proposed secure RMI.

Our architecture covers the authentication and authorization requirements of distributed applications in an easy-to-use way for software developers. We have followed an approach that keeps the details of the implementation transparent for the application developers. Generally speaking, we have taken into account making the architecture simple and making only the minimal changes required to enable the usage of secure RMI by the software developer. Our architecture also caters to future enhancements as well.

The contribution of this work is that it provides the first secure RMI architecture; that enables software developers to construct secure distributed applications with the Java RMI. We have used parameterized RBAC [2] that provides additional control over roles. We also provided a new way

Manuscript received November 22, 2010.
This work was supported by EastNets R & D Belgium, as part of their support of research towards risk management.
Nezar Nassr*(Member, IAENG)* is currently pursuing the Ph.D. degree at the Department of Computer Science and Engineering, Katholieke Universiteit Leuven , Belgium.(phone: 00 32 498901846; e-mail: nezar.nassr@cs.kuleuven.be). Nezar is also working as a consultant for EastNets R & D, Belgium.
Eric Steegmans is a Professor with the Department of Computer Science and Engineering, Katholieke Universiteit Leuven , Belgium (e-mail: Eric.Steegmans@cs.kuleuven.be).

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

of combining the rules together with possibilities of AND and/or OR combination of roles. In our architecture, we also provided access control at the method level. Moreover, with our design, software developers do not need to make dynamic checks, this ensuring that the method can only be invoked by a client holding the roles that are entitled to him to access the method. Given these checks are already implemented by our *Security Server*.

The remainder of this paper is organized as follows: In the next section, we introduce RMI, then in section III we introduce RBAC and parameterized RBAC is presented in section IV. In section V, we briefly review a number of related security models for distributed systems. Following that expose, we present our solution in section VI, followed by an explanation of the design of the *Security Server* in section VII. Finally section VIII concludes our work.

II. The Java Remote Method Invocation (RMI)

The Java Remote Method Invocation (RMI) is a middleware for constructing distributed applications. RMI enables applications to invoke methods on the server side.

Whenever an application invokes a method on the server side, it passes the arguments to the method, following this step, the method is executed on the server side and the client gets the return value of the method.

Figure 1 shows an architecture example of the Java RMI, the server application consists of the remote interface that defines the methods the client can invoke on the server, and the classes that implement the remote interface.

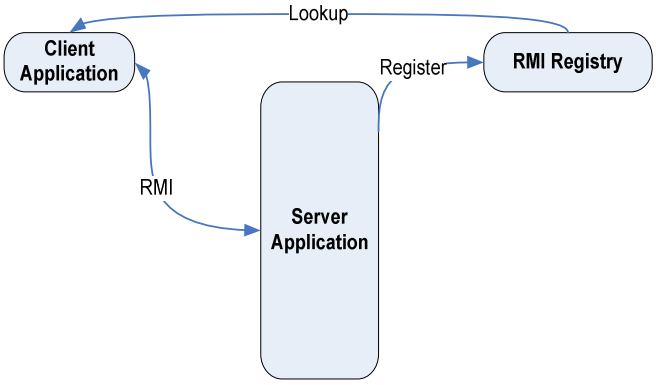

Fig. 1. High-level view of the Java RMI Architecture

The server application must register itself in the RMI registry server which is a naming service and a part of the RMI middleware. Clients must look up the service name before they can call methods. When the client looks up the server from the RMI registry it receives a reference to the server class.

When a client receives a reference to a server, RMI downloads a stub that translates calls on that reference into remote calls to the server. The stub marshals the arguments to the method using object serialization, and sends the marshalled invocation across the wire to the server. On the server side the call is received by the RMI system connected to a skeleton, which is responsible for unmarshalling the arguments and invoking the server's implementation of the method. When the server's implementation is completed, either by returning a value or by throwing an exception, the skeleton marshals the result and sends a reply to the client's stub. The stub unmarshals the reply and either returns the value or throws the exception as appropriate. Stubs and skeletons are generated from the server implementation, usually using the RMI Compiler [3].

RMI has evolved since it was first introduced in Java 1.1. RMI was extended in Java 2 to enable object serialization and eliminate the need for skeletons, which were replaced by reflection to make connections to the remote objects. Java 5 adds support for the dynamic generation of stub classes at runtime, obviating the need to use the Java Remote Method Invocation (Java RMI) stub compiler to pre-generate stub classes for remote objects [4].

The secure version of Java RMI (RMI-SSL) provides means for securing the communication channels between clients and servers; it also provides protection against security-sensitive actions such as accessing the local file system of the server.

RMI does not provide any means for protection by authentication or authorization. Any client which is able to lookup the service name of the RMI server is be able to invoke methods on the server application. Many applications require restricting access on the server methods invocation for only authenticated clients. More so, some applications require the ability to give clients access only on certain methods on the server application rather than full access.

III. Role-Based Access Control

RBAC is a form of access control that explicitly enables or restricts the resources in a software system. It protects against unauthorized use and manipulation of resources. In RBAC permissions are assigned to roles rather than to users, then roles get assigned to users and hence users get their access privileges in function of what roles they already have. Generally, roles represent functions or responsibilities that can be achieved by a software system, but can also represent sub-functions.

At present, application developers and deployers define the roles that make sense for an application and then identify which methods each role should be allowed to call. Therefore, access is defined in terms of operations on components[5]. For example, in a banking core application, we can divide users according to their roles in the bank, e.g. as tellers, account managers, sales, etc. The teller user could have a teller role that enables him to perform transactions and see clients information, whereas a sales person with a sales role could have read only access on client information. In a different design of roles, someone could define a read-only role and a "perform-transaction" role, in this case the teller could be assigned both roles, while the sales user could be assigned only the "read-only" role.

IV. Parameterized RBAC Model

Unfortunately, traditional RBAC does not support different levels of customization for roles. For example, in our banking example, it might be necessary to give different tellers different levels of the "perform-transaction" role. One junior teller could be allowed to do transactions that have small amounts, while another senior teller could be allowed to perform large amount transactions. This would be a nightmare if we are needed to assign an amount limit for each teller. This disadvantage was subject to further research, and it was addressed by some techniques such as Object-sensitive RBAC [6], parameterized RBAC [2], etc.

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

The other option for addressing this issue with traditional RBAC is to define a new role for each teller; which makes managing roles and keeping track of them more complicated. Moreover, It also could be a vast problem in dynamic and large scaled organizations.

Object-Sensitive RBAC [6] addresses the problem by extending the RBAC model to support fine-grained policies. The basic idea is to allow roles and privileged operations to be parameterized by a set of index values, which intuitively are used to distinguish users of the same role from one another. A privileged operation can only be invoked if both the appropriate role is held and the role's index values matches the operation's index value [6].

The parameterized RBAC model is actually similar to the Object-Sensitive RBAC; it improves upon the traditional RBAC model with enhancements that enable the definition of roles which can be customized or adapted according to the function changes between a user and another. This provides a way of implementing hierarchical and levels differences in roles. e.g. a senior teller who is allowed to perform transactions with large amounts and a junior teller that is able only to perform small amount transactions.

In the parameterized version of RBAC, although roles are defined as a single role, their implementation suggests their instantiation into a large number of roles to cater to every client, which presents a huge burden on the intellectual manageability of access rights [2].

The advantages most commonly associated to RBAC models, can be maintained if the roles are modeled as a parameterized RBAC. In this model, core RBAC components, such as roles, would depend on the values of a parameter. To extend RBAC into a parameterized model, data about the values of the parameters should be provided. New permissions that might be created due to the parameterization should also be identified [2]. By Parameterized RBAC, the problem explained above with teller roles can be solved effectively by defining one role called teller and associating an amount limit parameter to the teller role, then we can assign a small amount for the junior teller and a large amount for the senior teller.

In our work we used the Parameterized RBAC model as the access control model since it achieves extremely fine grained access control granularity, as well as provides completeness in terms of investigating all the concepts and semantics of a Parameterized RBAC and supporting all the definitions and features of other well-known RBAC models [2].

V. Related Security Models for Distributed Systems

In this context we can mention the DCE and the CORBA Security Services. DCE is a distributed platform based on RPC (Remote Procedure Call). Security is one of the basic components of DCE, and each CELL (a group of hosts) of DCE associates a security service, which has to be trusted by all member hosts of the CELL[7]. In DCE a different authentication procedure is necessary. When a user logs in, the login program verifies the user's identity using the authentication server, while authorization is handled by associating an ACL (Access Control List) with each resource [8].

The CORBA Security Service includes interfaces that define services for the following well-known areas of computer security: Authentication, Message Protection (including encryption for guaranteeing confidentiality as well as integrity), Access Control, Auditing, and Non-repudiation [1]. CORBA authorization is based on access control lists (ACL); which provide access control over resources or services provided by a CORBA application.

There have been some initiatives to improve upon the CORBA Security. R. Obelheiro [9] proposed an access control model for CORBA based on RBAC that supports automatic role activation by the security components of the middleware.

J2EE provides authentication and authorization mechanisms for Enterprise Java Beans (EJB). The authorization is based on basic RBAC access control. In general, security management should be enforced by the EJB container in a manner that is transparent to the enterprise beans business methods. EJB security provides security on the EJB method level where methods can be annotated with roles [4].

Moreover, there have been initiatives for adding security to Web Services, Damiani [10] has proposed an infrastructure for web services security that enforces access control policies on the request calls carried by SOAP. Wonohoesodo [11] has proposed two access control techniques: one for single services and another for global services. Their approach introduced global roles which are used in the mapping to local roles of other service providers. Moreover, they have proposed a role-mapping mechanism to maintain the autonomy of roles between providers.

VI. Our Proposed Architecture

RMI provides an infrastructure for developing distributed applications. This infrastructure bypasses some security mechanisms that are considered mandatory for many applications. In this work we extended the RMI infrastructure to adapt two goals that are not provided with the RMI infrastructure, which are authentication and authorization. The proposed architecture is shown in figure 2.

We introduced a *Security Server* in the RMI Infrastructure. The purpose of the *Security Server* is implementing security concepts of user authentication and authorization.

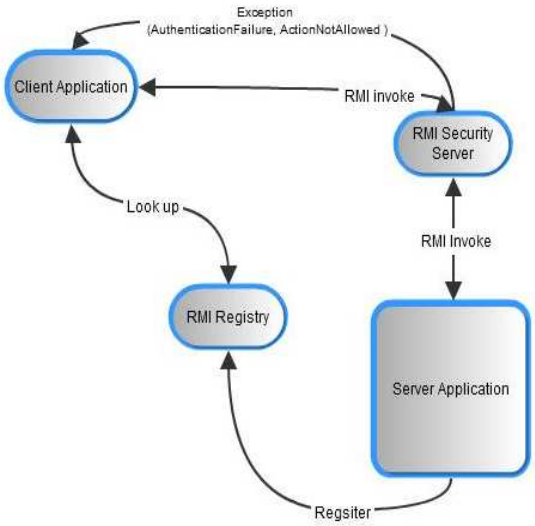

Fig. 2. The proposed secure RMI architecture.

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

A. Authentication

Authentication is the process of determining if a principal (that is a user or process that needs to communicate securely) really is who he/she/it claims to be [8]. Whenever a client attempts to invoke methods on the RMI server application for the first time, it must authenticate itself. This requires the client to provide a username and a password to the server application. The username and password pair are then checked by the *Security Server* and it determines whether or not the client is entitled to access the RMI server.

The security server must have the authentication information pre-defined in its database before clients can make requests to the server. We suggest that the client authenticates itself once a step could be directly after the client looks up the service name from the RMI registry service and gets a reference to the server, the *Security Server* afterwards maintains the session between the client and server applications.

B. Authorization

Once a user has been authenticated, the question arises concerning which resources that user may access and how, this issue is called authorization. We implemented an authorization mechanism on the method level using parameterized RBAC[2].

At the server application side, all the RBAC rules must be defined in the remote interface; each method will be annotated by one or more roles that restrict access of that method to holders of that roles.

Besides the authentication information of the client the security server must have the roles granted for that client. The *Security Server* extracts roles for methods by reading the annotations defined in the remote interface.

Once a client is authenticated and makes a request to invoke a method on the server, the *Security Server* should be able to determine whether the action is allowed or not, based on the roles assigned for that client and the roles needed by the method as defined in the remote interface.

VII. Security Server Design and Implementation

Now that, in the previous section, the proposed architecture has been explained, we will now introduce our design of the RMI security server.

Before getting into the details of the design of the *Security Server*, we would first like to expand upon the in-depth architecture of the Java RMI infrastructure.

RMI is built upon three layers as shown in figure 3; the first layer is the Stub and Skelton layer, which lies beneath the view of the developer. The stub marshals the arguments to the method using object serialization, and sends the marshaled invocation across the wire to the server. The skeleton carries on a conversation with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value, and then writes the return value back to the stub [3, 12].

The following layer is the Remote Reference Layer. This layer understands how to interpret and manage references made from clients to the remote service objects. More so, it connects clients to remote service objects that are running and exported on a server [12].

The third and final layer is the transport layer; which is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies [12].

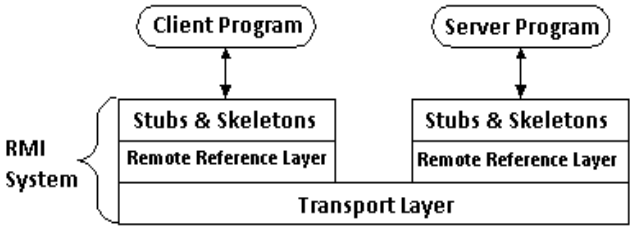

Fig. 3. Layered Architecture of the Java RMI, (figure from [12]).

Our design involves adding a new component to the RMI Architecture, which we call the *Security Server*. The *Security Server* is placed in the Skeleton layer at the server side. It also needs some modifications at the Stub level on the client side. We have also adapted our *Security Server* design to support stub and skeleton free RMI introduced in Java 5. This is accomplished through introducing two intermediate components at the client side and the server side that are automatically generated. These components are responsible for catering to the new changes required by the secure RMI. The new architecture is shown in figure 4.

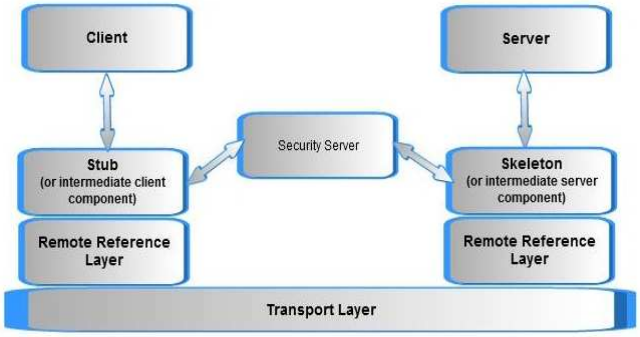

Fig. 4. The Layered Architecture of the proposed Secure Java RMI.

A. Authentication

Once the client looks up the server name from the registry sever, it must authenticate itself to the *Security Server*. We have introduced a new class (`SRMIAuthentication`) with a metthod `authenticateUser`, that is used by the RMI client to authenticate itself. This method sends the username and password of the user or the process to the *Security Server*, then the *Security Server* replies back with a ticket if authentication is successful, otherwise an exception is returned.

```
Ticket auth_ticket =
SRMIAuthentication.authenticateUser("use
r1","password123","localhost");
```

The username and password must be pre-defined in the *Security Server* database. The ticket is an encrypted data structure that is composed from the username, a random number that corresponds to the session ID, and an expiration date/time.

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

The *Security Server* has to maintain a copy of that ticket in its database. The stub (or the client intermediate component in Java 5) then marshals the ticket with each invocation request to the RMI server. The *Security Server* then intercepts each request at the server side and validates the ticket with the value stored in its database. Based on the validation result, the Security Sever either allows further processing or blocks the request.

This process is transparent for the client program, the only change that is needed at the client program is providing its credentials once it starts a new session using the `SRMIAuthentication.`*`authenticateUser`* method.

B. Authorization

The *Security Server* parses the Remote Interface and reads the annotations of the methods, then it determines which roles are needed for invoking each method. The *Security Server* then maintains this information in its database.

When the client sends a request to invoke a method, the stub must then marshal the ticket received from the *Security Server* to the RMI server with the method arguments. When the Skeleton receives the method arguments and the ticket from the Stub, it extracts the ticket from the arguments and sends it together with the method name to the *Security Server* for verification. The *Security Server* then checks if the ticket exists in its database. If it exists then it reads the roles assigned to that client and checks if the request to invoke the method could be achieved. If the client has the roles required for invoking the method, then the *Security Sever* sends a positive signal to the Skelton allowing it to continue the process. If the client does not have enough roles to invoke the method, the *Security Server* then sends a negative signal to the Skeleton which by its turn denies the client's request and throws an ActionNotAllowed Exception which is then marshalled to the Stub.

Nothing has to be changed in the RMI Server application, given all changes are done in the skeleton on the server side which is generated by the RMI Compiler (rmic). All the changes done in the stub and skeleton are systematic, and automatic generation of the changes is very straightforward.

C. RBAC Roles Definition for Server Methods

To control access control on methods at the server side, we have adapted the Parameterized RBAC model explained in section IV. The server methods are annotated with a set of roles that are required by each method to enable invocation. These roles are specified in the methods declarations in the Remote Interface. We use Java annotations to achieve this functionality.

Java EE provides a specification for defining method permissions using annotations for Enterprise Beans [4]. The method permissions for the methods of a bean class can be specified on the class, the business methods of the class, or both. Method permissions can be specified on a method of the bean class to override the method permissions value specified on the entire bean class. The approach provides three types of permissions; the `@RolesAllowed("`*list-of-roles*`")` annotation; is a list of security role names to be mapped to the security roles that are permitted to execute the specified method or methods, while the `@PermitAll` annotation specifies that all security roles are permitted to execute the specified method or methods. Finally the `@DenyAll` annotation states that no security roles are permitted to execute the method or methods [4].

However, this approach does not provide any possibility for defining parameterized RBAC roles. As a consequence, we adapted a similar approach of specifying roles, but with modifications to enable parameterization of roles. A similar approach was introduced in [6] for defining roles with parameters.

We adapted an approach for defining the roles similar to mathematical function definitions, where the roles names substitute the function name and the parameters substitute the function arguments. For example:

Role_Name (param1, param2)

Role parameters could take values in the role definitions which can also utilize wide range of operands e.g. > ,<, =, ∈, etc. If we take an example as a bank account and the role required to enter a payment for an account is *AddPayment* and this role could have parameters such as:

- Account number in group of customers that the clerk can handle (e.g. accGRP).
- The clerk is allowed to do transactions on account with amounts larger than the amount specified (e.g. Amount).

Then our definition of the roles will be as follows:

```
@RolesAllowed {AddPayment (acc_num in accGRP,
Amount > transaction_amount)}
public void Insert_Payment(acc_num,
transaction_amount){

}
```

In the example above, the user must hold the AddPayment role with amount greater than the transaction amount as a parameter for his role to be able to invoke the method. In our approach, it is also possible for a method to have more than one role in the @RolesAllowed annotation. Moreover, it is possible to separate roles with an AND operand allowing the method to require more than a role to be invoked or with an OR operand allowing the user to invoke the method if he has one of the roles specified.

The following code fragment shows an example of a Remote Interface with RBAC roles declarations.

```
import java.rmi.*;
public interface ExpenseServer extends Remote {
      @RolesAllowed{AddPayment(Amount>
transaction_amount)   }
      void InsertPayment(acc_num, transaction_amount,
value_date)
      throws RemoteException;
}
```

Fig. 5. Code fragment shows remote interface with annotated RBAC roles.

The security server reads the annotations ahead of each method then it creates a data structure that have all method names and the required roles and parameters for invocation. The *Security Server* asserts each invocation request with this data structures to ensure that the client possesses enough permission to invoke the method.

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

D. Session Maintenance between Client and Server Applications

In our design we provide a mechanism that avoids redundant authentications between clients and servers when a client tries to invoke a method on the server. Once the client is authenticated, the *Security Server* generates a ticket that is associated to the client authentication information in the *Security Server* Database. Afterwards, the client passes this ticket with each request to invoke a method. The *Security Sever* checks this ticket and determines if the client was authenticated before or not. This ticket has an random generated code that acts like a session identifier.

VIII. Conclusion

In this work we presented both a design and an implementation of a security server for the Java RMI. RMI has been used in many different enterprise business applications, however, no security service has been implemented as of yet. On the other hand, there have been security services implementations for other distributed systems infrastructures such as CORBA, but these designs use access control lists (ACL) and control over resources.

We have designed a simple and extendable architecture that uses parameterized RBAC for access control which is much better than access control lists. This enables software developers who wish to use our secure version of RMI to have more control on accessing methods rather than resources since in many cases resources represents general concepts.

Our current implementation can be enhanced by using more sophisticated authentication techniques and by using a more refined language for expressing RBAC roles. And finally with an RMI compiler that can generate the secure stubs and skeletons.

References

[1] A. Alireza, U. Lang, M. Padelis, R. Schreiner, and M. Schumacher, The Challenges of CORBA Security, *Sicherheit in Netzen und Medienströmen, Informatik aktuell*, 2000.

[2] A. E. Abdallah and E. J. Khayat. A formal model for parameterized role-based access control. *Formal Aspects in Security and Trust.* Springer, 2004.

[3] Java RMI White Paper, http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp

[4] The Java EE 5 Tutorial, http://java.sun.com/javaee/5/docs/tutorial/doc/index.html

[5] R. Sandhu, E. Coyne, H. Feinstaein, C. Youmann,. Role-based access control models, *IEEE Computer* 29 (2). (1996) 38–47.

[6] J. Fischer, D. Marino, R. Majumdar, and T. Millstein, Fine-Grained Access Control with Object-Sensitive Roles, *European Conference on Object-Oriented Programming* (ECOOP 2009).

[7] Xingshe, Z. and Xiaodong, L. Design and Implementation of CORBA Security Service. *In Proceedings of the 36th international Conference on Technology of Object-Oriented Languages and Systems. TOOLS*. IEEE Computer Society, Washington, DC, 2000.

[8] Andrew S. Tanenbaum, Distributed operating systems. *Prentice-Hall, Inc. Englewood Cliffs*, 1995, pp554-563.

[9] Obelheiro, R. R. and Fraga, J. S. 2002. Role-Based Access Control for CORBA Distributed Object Systems. *In Proceedings of the the Seventh IEEE international Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)(January 07 - 09, 2002). WORDS. IEEE Computer Society*, Washington, DC, 53.

[10] Damiani, E., di Vimercati, S. D., Paraboschi, S., and Samarati, P. 2001. Fine grained access control for SOAP E-services. *InProceedings of the 10th international Conference on World Wide Web* (Hong Kong, Hong Kong, May 01 - 05, 2001). WWW '01. ACM, New York, NY, 504-513.

[11] Roosdiana Wonohoesodo, Zahir Tari, "A Role based Access Control for Web Services," scc, pp.49-56, Services Computing, 2004 *IEEE International Conference* on (SCC'04), 2004.

[12] RMI Online Course, http://java.sun.com/developer/onlineTraining/rmi/RMI.html

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)