Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp1045-1050)

A Pluggable Security Framework for Message Oriented Middleware

RUEY-SHYANG WU, SHYAN-MING YUAN
Department of Computer Science
National Chiao-Tung University
1001 Ta Hsueh Road, Hsinchu 300,
TAIWAN, R. O. C.
http://dcsw3.cis.nctu.edu.tw

Abstract: - With the rapid growth of enterprise applications, Message-oriented Middleware (MOM) has become a widely used tool for delivering messages between organizations. Sun Corporation has been aware of this trend and has defined a Java Message Service Application Programming Interface (JMS API) standard. This standard provides a set of uniform interfaces for application development and makes applications more portable. Persistent Fast Java Messaging (PFJM) is a JMS compliant Message-oriented Middleware and it has some outstanding features such as persistent message and high performance.

Although MOM has been widely adopted in enterprise environments, security issues were not noticed in the past. This paper discusses the security issues of MOM. Based on PFJM, a pluggable security framework is developed. "Pluggable" means that the application developers only need to modify configurations and plug in many different security modules to build a secure message delivery system. Developers do not have to modify applications in order to adopt different security strategies, and can be more flexible in development.

Key-Words: - Pluggable Framework, Security, MOM, JMS, JAAS, EAI

1 Introduction

Message-oriented Middleware (MOM) is usefully for application integration because it is efficient, reliable and scalable. Applications can exchange data through MOM quickly; the data will not be lost; adding more applications in the existed environment is also easy. For these reasons, MOM has been widely adopted in enterprise environments. Programmers do not access lower-lever and disagreeable network protocols directly because MOM provides a set of readable and simple application program interfaces (APIs). Programmers can use these APIs to send and receive network messages in heterogeneous environments. By this way, information system can quickly be available. However, when programmers move from one MOM system to another, the codes about communication in an application must be modified because every MOM's API is not the same. For portability, Sun defined a standard of Java Message Service Application Programming Interface (JMS API)[1] adopted by many MOM providers. The advantages of the standard interface are programs written by this API can run on many MOM products and programmers only learn one API.

The latest version of the JMS standard is 1.1. Many MOM providers support the latest JMS standard, such as SonicMQ[2], FioranoMQ[3], OpenJMS[4], and others. Persistent Fast Java Messaging (PFJM)[5] is a Message-oriented Middleware. It also follows JMS 1.1 standard while enhancing the persistent message and high performance features, greatly increasing the program's scalability.

1.1 Motivation

Since the e-commerce was more and more popular, network security became an important issue. The messages delivered via network in plaintext may be snooped by crackers. Therefore, MOM providers began to add security functions to their products, including user authentication, trust authorization and message encryption. However, the functions were usually and tightly bound with the implementation of MOM's core. Furthermore, the security functions were developed by MOM providers, which are preventing third-party providers from plugging in their security functions into MOMs, so this makes applications less flexible.

Therefore, this study suggests a pluggable security framework for MOM. The core concept is loosely-coupled design. Based on this design, the MOM's core functions can be separated from security functions. The security functionalities can be packed in the form of modules. Thus, development, programmers need only be concerned with business models, rather than security issues. During deployment, system administrators can simply construct a secure message system by modifying the

configuration, parameters, and suitable security modules.

Based on PFJM, we will develop the modular concept of security functionalities. In addition, we will implement several useful security modules to demonstrate the advantages of the pluggable framework.

1.2 Objective

Although many MOM products adopt a fixed manner of authentication and authorization, the program architecture variation is large and resource management is difficult. A more flexible way to accomplish authentication and authorization is needed. The Java Authentication and Authorization Service (JAAS) is a framework defined by Sun for flexible authentication, and is widely used in industry, but JAAS has many disadvantages in authorization functions and still has some constraints which prevent its use in distributed environment. For this reason, a new, more flexible security framework is discussed in this paper.

This paper focuses on adding security functions to MOM, but not security functionality itself. The implementation and strength of security functions lie beyond the scope of our discussion. Pluggable security provides a security interface for both system architecture and developers. System architecture can adjust the security functions based on the business requirements; developers can write programs without realizing the underlying security functions of the MOM. Without the framework, developing MOM applications will be more flexible, more scalable and easier.

2 Related Works

2.1 Java Message Services (JMS)

Java Message Service, defined by Sun Corporation and other cooperators, is a set of standard interfaces for message delivery. With JMS, programmers can create, send, receive, and read messages via simple interfaces. The standard only defines the semantics of interfaces, not implementation. Following the interfaces, all MOM providers can have their own implementation. Today, almost all MOM products are compatible with JMS. Common design architectures can be divided into two categories: central architecture and distributed architecture.

2.1.1 Central Architecture

A master server is responsible for message delivery and all applications are clients. When an application wants to send messages, it becomes the sender, calling the JMS API and giving the destination of the messages. Messages are delivered from the application to the central server. The central server then looks for the destination of the messages and sends them to the appropriate client. The advantage of this architecture is that it can simplify system design and reduce developing effort. However, the shortcoming of a central architecture is the server-bottleneck problem. If the central server experiences low performance or even failure, the entire message exchange system will becomes unavailable.

There are many MOM products such as IBM WebSphere MQ[18], TIBCO Redenzevous[17], Sonic Software's SonicMQ**Error! Reference source not found.**], Fiorano Software's FioranoMQ**Error! Reference source not found.**] and OpenJMS**Error! Reference source not found.**] which is released in the form of open source.

2.1.2 Distributed Architecture

There is no master server under this architecture. Every client is responsible for message delivery. Because there is no longer a server, every client must be aware of some information of the other clients. For example, clients must know the IP addresses and port numbers of other clients. The advantage of this architecture is that loading is divided between every client, exactly opposite of a central server. As a result, the single-point-failure problem does not exist in this architecture. However, resource management becomes complicated and difficult.

PFJM (Persistent Fast Java Messaging)[5] is a JMS compliant product designed by our laboratory. PFJM adopts the distributed architecture and implements message delivery protocol using IP multicast technology. Every PFJM instance simultaneously acts as a client as well as server. When one PFJM instance starts, it connects to other PFJM instances via multicast messages in the network and obtains all necessary information. When delivering a message, an application talks to PFJM and indicates what message to send and which Topic the message should go. PFJM is responsible for dividing the message into many equal-sized Memory Buffer Units (MBUs) and putting them into the delivery buffer in memory. Then Carrier picks up these MBUs and actually sends them out via multicast. While PFJM fills the role of receiver, it will register the appropriate multicast channel corresponding to the Topic. Incoming messages (MBUs) will be received by the Carrier and then composed by the Composer. Finally, the origin message is delivered to application.

Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp1045-1050)

2.2 Security
Network security is becoming more and more important today due to hackers and computer viruses. Performance is not only the focus on MOM. Some are more concerned with issues of privacy, non-replacement, and non-deniability. This means that every publisher and subscriber in the message exchange environment needs to be authenticated, and messages transferred in the network need to be encrypted. For this reason, MOM needs more functions such as user authentication, authorization and message encryption. Most MOM providers were not concerned with security issues in the past, but recently, they have added these functionalities enthusiastically

Some industry products use the Cipher interface of JCE (Java Cryptography Extension)[6]. Its advantage is that encryption algorithms can be dynamically added into a Java environment. In Sun's implementation, it supports AES, Blowfish, the DES series, and the RSA series algorithms. However, it does not provide complete authentication and authorization functions. Another standard, JAAS (Java Authentication and Authorization Services)[7] has those functions but is not included in most industry products. To provide complete security functions, a standard interface that has authentication, authorization and encryption is necessary.

3 Pluggable Security Framework
Pluggable Security Framework architecture uses a modular design. After building the core module, other functionalities can be added to the software as modules without modifying the software and existing program code. To complete the architecture, the interface is designed and the configuration files are illustrated.

3.1 Architecture
The pluggable security framework can fit most MOM products, regardless of central architecture and distributed architecture. However, distributed architecture is more flexible than central architecture. For this reason, we used a distributed architecture, PFJM, as our implementation environment. Fig. 1 shows the architecture. The authentication interface, authorization interface and encryption interface all talk to related modules to actually perform authentication, authorization and encryption.

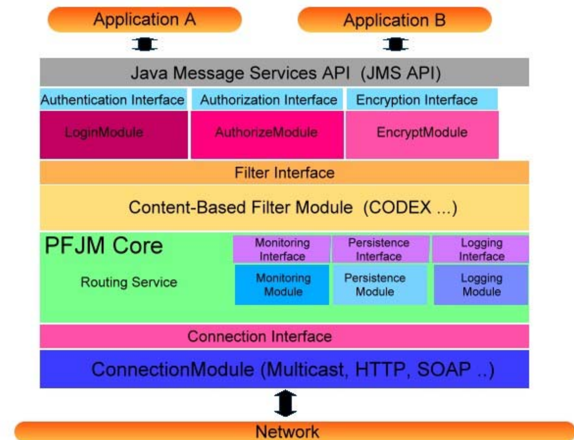

Fig. 1 Architecture of pluggable framework on PFJM

3.2 Interface Design
MOM security functions include two major interfaces: authentication and filter interface.

Authentication Interface. The authentication, authorization, and encryption interfaces are security related interfaces. An abridged MOM system should have authentication functionality. When a client accesses the MOM system, it should be authenticated and be a legal user of the system. After successful authentication, the MOM system may authorize clients. It determines which Topic the client can create or remove, and if they can publish to or subscriber to the topic. Authorization usually executes with authentication. Similar to authentication, authorization can be performed on an authentication server or another authorization server. When a client passes the processes of authentication and authorization, and it has the appropriate access, it can start publishing or subscribing to messages. In order to protect messages against theft in the network, published messages should be encrypted.

Filter Interface: The JMS standard only defines an essential application based filter. The client indicates interesting kinds of messages according to the information in message headers and properties. But this becomes too simple when applied to complicated applications. Some MOM providers may add filters such as content based filters. Another kind of filter is the authentication based filter where each client will only receive their own message. Others without privileges will not get that message.

From the programmer's perspective, MOM should provide the two functions mentioned above so that applications can have all security features. Based on requirements, the following interfaces are defined. The Authentication Interface should have the *LoginModule*, *AuthorizationModule*, and *EncryptionModule*. Those modules are embedded

Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp1045-1050)

into MOM and programmers will never recognize those interfaces.

```
public interface LoginModule
{
   boolean abort ();
   boolean commit ();
   void initialize (Subject subject, CallbackHandler callbackHandler,
            java.util.Map sharedState, java.util.Map options);
   boolean login ();
   boolean logout ();
}
```

Fig. 2 LoginModule Interface

The *LoginModule* covers MOM application startup, joining the enterprise environment, taking actions after login, and leaving the environment. The *abort()* and *commit()* commands will be executed if the login is a failure or success, respectively. The *initialize()* command requires some information about MOM and security requirements. The *callbackHandler* acquires security information, such as username and password, or digital signature. The *sharedState* allows two different login modules to exchange information. The design follows JAAS architecture.

```
public interface AuthorizeModule {
  public void initialize (Subject subject, Authority authority,
            Map sharedState,Map options);
  public boolean retrievePolicy ();
  public boolean cleanup ();
}
```

Fig. 3 AuthorizedModule Interface

The *AuthorizeModule* uses login information from the *LoginModule* to gain application privileges. After authentication, the *LoginContext* will turn control over to the *AuthorizeContext*. Just like the *LoginModule*, the *AuthorizeContext* will call every *AuthorizeModule's* initialized method and pass in the Subject gained from the *LoginContext*. Moreover, the *AuthorizeContext* will pass in an authority data structure for storing permission information. The *sharedState* and options parameter is the same as the *LoginModule*. The *retrievePolicy()* will get privilege information from the MOM or database. It is defined by the *TopicPermission*, like creation, deletion, read and write. Finally, the *cleanup()* command will clear privilege data.

```
public interface EncryptModule {
  public void initialize (Subject subject, Map sharedState,
            Map options);
  public byte[] getEncryptedMessage (byte[] msg)
            throws EncryptException;
  public byte[] getDecryptedMessage (byte[] msg)
            throws EncryptException;
  public boolean cleanup ();
  public boolean negotiate ();
}
```

Fig. 4 EncryptModule Interface

The *EnvryptModule* will encrypt and decrypt messages. The *initialize()* will prepare the encryption information, such as key information or key negotiations with peers. The *getEncryptedMessage()* and *getDecryptedMessage()* can perform encryption and decryption. Finally, it will call the *cleanup()* to remove this information.

3.2 Configuration File

The configuration file of PFJM is in XML form. The security section in this file represents the security related setting. Fig. 5 is an example; PFJM instance uses the *LDAPLoginModule* to login to the message exchange system. The server option informs the *LDAPLoginModule* where the Lightweight Directory Access Protocol (LDAP) server is located. Then it uses the *LDAPAuthorizeModule* to gain its permission and finally, uses the *DESEncryptModule* to encrypt the outgoing message using a Data Encryption Standard algorithm.

```
<Security>
  <!-- This part specifies which LoginModule should be invoked -->
  <!-- "needs" option can be 'required','sufficient','requisite','optional' -->
  <LoginModule
      name="com.cmc.jms.security.modules.LDAPLoginModule"
      needs="required">
   <!-- Optionally to indicate which CallbackHandler should be used -->
   <CallbackHandler
      name="com.cmc.jms.examples.MyCallbackHandler"/>
     <Option debug="true"/>
     <Option server="140.113.88.237"/>
     <SharedState myState1="ok"/>
  </LoginModule>

  <!-- This part specifies which AuthorizeModule should be invoked -->
  <AuthorizeModule
    name="com.cmc.jms.security.modules.LDAPAuthorizeModule">
   <Option debug="true"/>
   <SharedState myState2="ok"/>
  </AuthorizeModule>

  <!-- This part specifies which EncryptModule should be invoked -->
  <EncryptModule
    name="com.cmc.jms.security.modules.DESEncryptModule">
   <Option debug="false"/>
   <SharedState myState3="no"/>
  </EncryptModule>
</Security>
```

Fig. 5 Configration File

4 A Security System on PFJM

After integration with the pluggable framework, many security modules are used to construct a secure message exchange environment.

4.1 Overview

In order to integrate authentication and authorization into PFJM, it is necessary to manage various resources well, including items such as user list and topic directory. Unfortunately, this is more difficult in distributed architecture because the management protocol may be too complicated and less effective. To control all information, a Security server was constructed to handle authentication and privilege information.

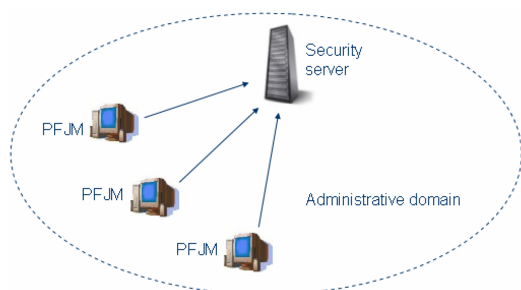

Fig. 6 A secure system architecture for PFJM environment

4.2 Security Server

LDAP (Lightweight Directory Access Protocol) is used as the security server. LDAP is a lightweight protocol for rapidly searching for specific data in a LDAP server. The data structure in a LDAP server has a tree architecture and is denoted as DN (distinguished name). In the server, three resources are classified: user directory, topic directory and keystore directory. User directory has everything about a user such as user name and password. Topic directory keeps some topic information, like topic name and user privilege. Keystore directory stores all private keys.

4.3 Module Design

The *LDAPLoginModule* authenticates PFJM to the OpenLDAP server and authorizes via *LDAPAuthorizeModule*. Finally *DESEncryptModule* encrypts messages that are sent to topic. The following modules are necessary to complete the environment:

LDAPLoginModule: Responsible for user authentication. It uses the LDAP protocol and communicates with the OpenLDAP server. The *LDAPLoginModule* will ask for user name and password using CallbackHandler. Then it passes this information to OpenLDAP using a SSL secure connection.

LDAPAuthorizeModule: Used to retrieve necessary permissions and pass to PFJM. If PFJM has no permission to perform that action, it returns the exception.

DESEncryption: Used to encrypt messages. The module uses a 64-bit key space and Data Encryption Standard algorithm.

4.4 Scenario

An OpenLDAP server was set up to implement an e-Paper publishing system to demonstrate the advantages of the pluggable framework. The system flow was as follows:

1. There were three clients. Client A was the message publisher; client B and C were receivers, but client B received more information than C.
2. Client A published messages.
3. Client B and C required a username and password to identify the user.
4. After authentication, Client B and C received the message.

Fig 7 shows the user login screen. The central screen is Client A that will publish messages. Client B is at the right screen and Client C is at the left screen.

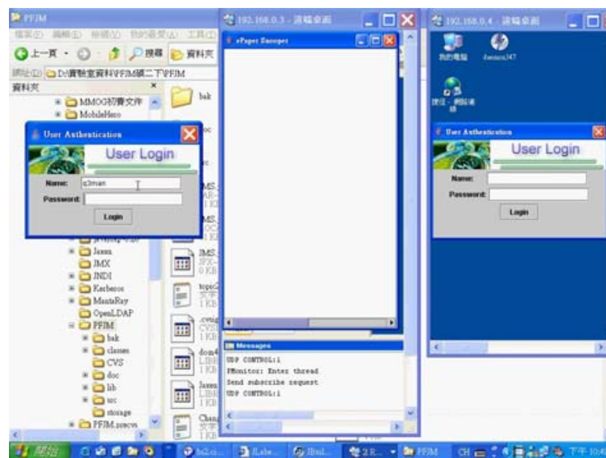

Fig. 7 user login screen

Fig 8 shows the message that users received. Client B will get more information than Client C.

Proceedings of the 5th WSEAS International Conference on Applied Computer Science, Hangzhou, China, April 16-18, 2006 (pp1045-1050)

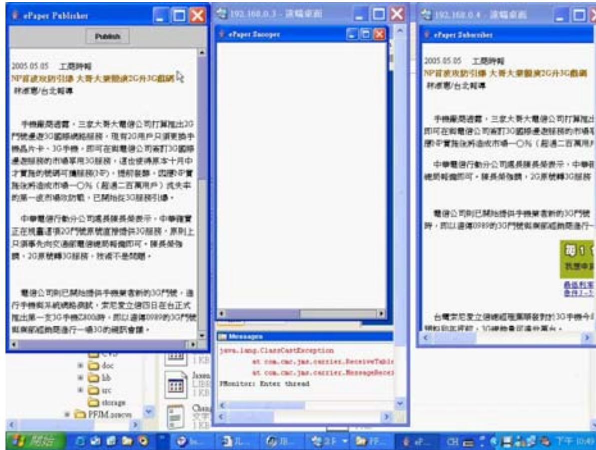

Fig. 8 users receive messages

The pluggable framework advantages of flexibility and ease-of-use are illustrated in the above scenario. The programmers were able to write to a traditional JMS client and discard security functions. The pluggable security framework takes over these functions, such as authentication and message filtering.

5 Conclusion and Future Works

MOMs simplify the sophisticated processes for delivering messages. For portability, Sun defined the JMS API and programs written using JMS API have the benefit of "write once, run anywhere". Because security issues are more and more concerned, every MOM providers usually add their own security functions into their products. In this paper, many MOM security issues, including user authentication, trust authorization, and message encryption, are discussed. We surveyed the methods how MOM products on the market solve these issues. In addition, we conclude that their design is less flexible and cannot be adapted to distributed systems. Therefore, a pluggable security framework with more flexibility was proposed. We designed our security functions according to the modular concept, and defined several interfaces for security modules. We then built the pluggable security framework into PFJM and wrote several security modules for it. Finally, we designed a system that uses a LDAP server for authentication and authorization to demonstrate the flexibility of the pluggable security framework. Using the pluggable framework, the distinction and collaboration of software components becomes clearer. Moreover, the maintaining and adding to the software become more convenient.

In the future, we will also implement more security functions, such as asynchronized key encryption functions. We will also enhance the security level, like per-topic encryption. Providing more security functions can not only evaluate the interface but also make the MOM more secure.

References:

[1] Sun Microsystems, Java Message Service Specification Version 1.1, April 2002

[2] Sonic Software's SonicMQ, http://www.sonicsoftware.com/index.ssp

[3] Fiorano Software's FioranoMQ, http://www.fiorano.com/products/fmq/overview.htm

[4] Project OpenJMS, http://openjms.sourceforge.net/

[5] Yu-Fang Huang, Tsun-Yu Hsiao, Shyan-Ming Yuan. A Java Message Service with Persistent Message, Proceeding of Symposium on Digital Life and Internet Technologies 2003

[6] Sun Microsystem, Java Cryptography Extension (JCE) Version 1.1

[7] Sun Microsystem, Java Authentication and Authorization Services (JAAS) Version 1.0, December 1999

[8] DEC-RFC 86.0 from SunSoft, Unified Login with Pluggable Authentication Modules (PAM), October 1995

[9] MIT, Kerberos: The Network Authentication Protocol

[10] Eric Glass, The NTLM Authentication Protocol, 2003

[11] RFC 3377, Lightweight Directory Access Protocol (v3) Technical Specification, September 2002

[12] National Bureau of Standards, "Data Encryption Standard," U.S. Department of Commerce, FIPS pub. 46, Jan. 1997

[13] National Institute of Standards and Technology (NIST), "Advanced Encryption Standard (AES)", FIPS Publication 197, Nov. 2001, http://csrc.nist.gov/encryption/aes/ index.html

[14] Project dom4j, http://dom4j.org/

[15] Project OpenLDAP, http://www.openldap.org/

[16] Sun Microsystem, Enterprise JavaBeans Technology (EJB) Specification Version 2.1

[17] TIBCO Software Inc., http://www.tibco.com/

[18] IBM Software – WebSphere MQ., http://www-306.ibm.com/software/integration/wmq/