Secure Chained Observer Pattern in Distributed Systems

VIVEK BABU GONDI AND DATTA G, MINDTREE LIMITED, BANGALORE, INDIA

Abstract: The concept of sending messages to several receivers is a common scenario in distributed applications. However, there is a certain degree of complexity involved assuming that the sender and receiver are physically at remote locations. Aspects like accessibility, security, and large volumes are key factors in these systems. Therefore, messaging in distributed systems poses several challenges and requires optimal design. We use this problem to come up with a pattern in sending secure messages automatically from the sender to receiver(s) using a secure observer classes.

General Terms
Secure, Messaging, Observer Pattern, chained, private/public key pairs, distributed systems

Keywords
SSN: Social Security Network, PKI: Public Key Infrastructure

1. NAME OF THE PATTERN

This pattern is named 'Secure Chained Observer Pattern in Distributed Systems' because it uses a combination of observer classes and security mechanisms to relay messages from the sender to the receiver. It is named 'chained' as messages are transferred from sender to receiver via a particular path. [1]

2. CONTEXT

In distributed systems, communication between objects is a key factor for a good system design. It is required that the starting point and end point needs to be connected in a secure and automatic fashion. Often at each step, action needs to be taken without any effort from the sender. For e.g.: look ups need to be performed to route to a particular sender, messages need to be formatted in the intermediate steps etc. After each step, an event might need to be sent to the receiver. Message notification might need to happen in several steps finally reaching the intended recipient. This kind of situation can be addressed using this pattern. [2 & 3]

3. FORCES

Ease of Access: Users can subscribe to different services in the market via Service Providers like Bank, Telecom company etc. It is cumbersome to contact all these providers by an end user when a uniform message needs to be sent to all of them. For e.g.: A user would like to Renew / Cancel any contracts with a service provider, or modify personal details with all of them**.**
Security of messages: With the increasing threat of hacking personal data, it is imperative that messages are transmitted using proper encryption mechanisms. Without this feature, the messages could be read by external parties over the network and can be misused.
Message Handling: In many situations content of messages must be looked at each step and necessary action like routing to another receiver might be needed. Without this happening in an automatic way, lot of time would be lost in delivering the message to the receiver.
Volume of Messages: All major companies who use messaging between their customers have high throughput of messages to handle everyday. With the increasing competition in the market, if the volume is not handled efficiently and automatically, the vendor would not stand up to the competition from others

4. SOLUTION

4.1 USECASE DIAGRAM

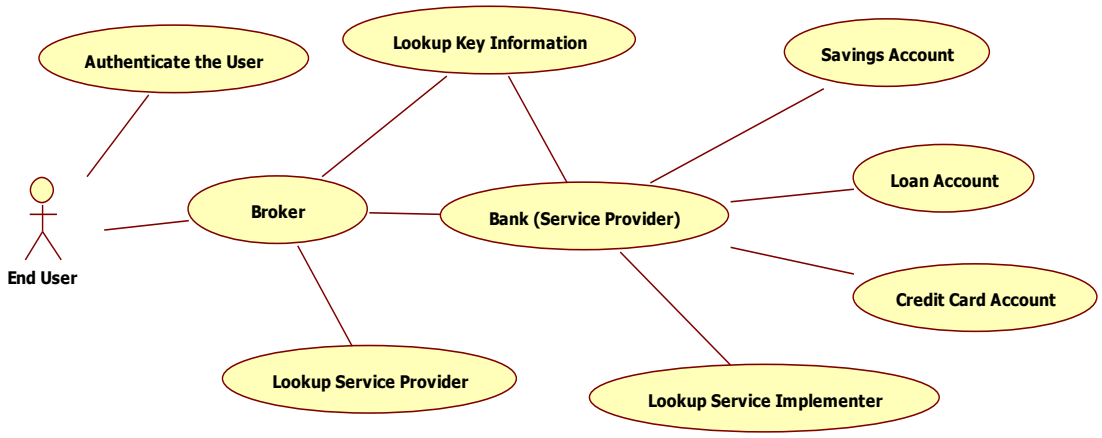

Figure 1: Use Case for a typical scenario of user contacting any service of a bank

4.2 DESCRIPTION

Overview: One possible solution to implement the above problem is described in this section. Assume a typical use case, where the user would like to communicate to a 'Bank' (receiver) for some requirement. For understanding the sequence of events that occur in this pattern, refer to Fig 3. Initially, the credentials of the user are verified by an 'Authentication Service'. The credentials could be valid SSN (Social Security Number)/Date of Birth/password etc or any identity that the user has registered with the system. Once the authentication is successful, the flow passes next to the 'Broker Class'. This class mediates all requests from the user and routes them to the 'Service Provider' (e.g.: Bank) using lookup information. Both the 'Broker' and 'Service Provider' implement 'Observer' and extend 'Observable' classes. The service provider in turn routes the request message to the required 'Service Implementer' (e.g.: Savings, Credit Card Account etc). The class which is responsible for execution of the user's request is called 'Service Implementer'. Encryption and decryption of messages between the above classes is handled by the 'Message Handler' class. All public/private key related information is obtained from the 'KeyStore' class.

The detailed description for each of the above classes is given below.

LoginVO: This class is a client that demonstrates the flow of the user request from the beginning to the end. Initially, the credentials of the user are validated using 'Authentication service' class. The credentials could be valid SSN, Date of Birth, Bank Details etc or any identity that the system is registered with. Upon successful login, it instantiates the 'MessageHandler' class and sets the required input message to the handler object. It is an 'Observable' class with add and notify methods. The addObserver() contains the 'Broker' as the observer and notifyObserver() is responsible for publishing the message to the 'Broker' automatically using Observable class
Finally, it contains the executeService() method. This method sends the encrypted message received from the MessageHandler class and calls the notifyObserver() method to notify the observers subscribed to this object.

MessageHandler Class: This is a utility class for handling all encrypted messages between different objects. The main responsibility of this class is to return an encrypted message (getEncryptedMsg()) to the calling class using the private key of the sender. When a sender gets an encrypted message, it is also responsbile for decrypting the message (getDecryptedMsg()) by obtaining the required public key from the 'KeyStore' class.

ServiceLookup Class: This class provides the list of observers for a particular object. For e.g.: If a 'Broker Class' is the input to the lookup() method, the method would return the list of 'Service Observers'. It contains one overloaded method to accept different objects.

BrokerObserver Class: This class is the 'Observer' for the 'Login' class. This class acts as the first 'Observer' for calls received by the client. This also extends the 'Observable' class as any change to this object is 'observed' by the Service Provider class. The

'addObserver()' contains the list of 'Service Provider(s)'. The relationship between the 'Broker' and 'Service Provider' is provided by the 'Service Lookup' class. Broker class gets the encrypted message from the 'Message Handler' and decrypts the message by obtaining the public key from the 'KeyStore' class.

Service Provider (Observer) Class: This class receives requests from the 'Broker Observer'. Just like Broker it is both 'Observable' and also extends the 'Observer' interface. This class observes changes to the Broker class and notifies the 'Service Implementer' classes. This class routes the message to appropriate 'Service Implementer' using look up. Just like the Client-Broker-Service Provider channel, the 'Broker-Service-Provider-Service Implementer' classes communicate in a similar way. Thus a chained observer structure is realized.

Service Implementer Class: They are the 'Observer(s)' to the Service Providers and hence listen to all updates in the Service Provider class. The service implementers actually execute the service requested by the client and are the destination point for which the client is looking to. It contains the execute() which accepts the service message, decrypts using the public key and performs the required action.

KeyStore Class: This contains public key for all observers to decode the encrypted messages. The key pair generation for each of the 'Observer' classes is handled here. Every 'Observer' object gets its public key from this class to read the encrypted messages when required. For e.g.: The Broker Observer class decrypts the message, deciphers what service the client is looking for using the key obtained from this class. It then forms the required content for message using the Message Handler class getEncryptedMessage(). The message is signed using Broker private key and passed to Service Provider Observer. Upon receiving the message by the 'Service Provider', it calls getDecryptedMessage() and decrypts the message as per key obtained from the 'KeyStore'

The class diagram is illustrated below in Fig 2.

4.3 CLASS DIAGRAM

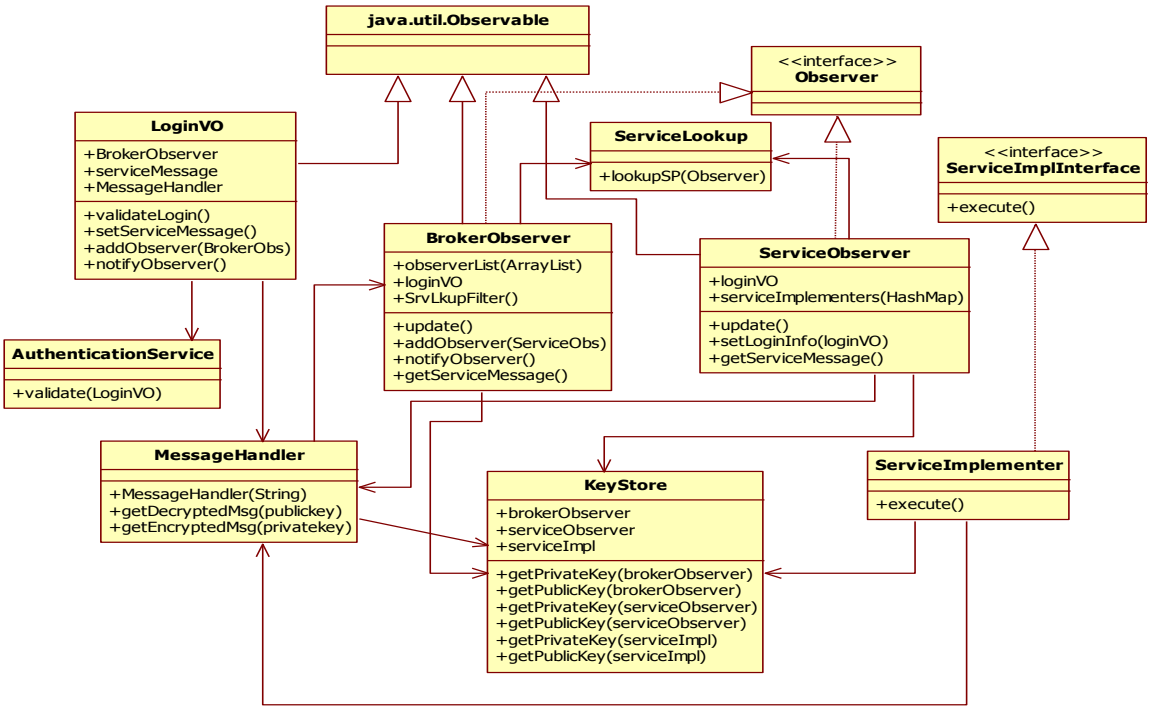

Figure 2: Class diagram for secure chained observer pattern

4.4 SEQUENCE DIAGRAM

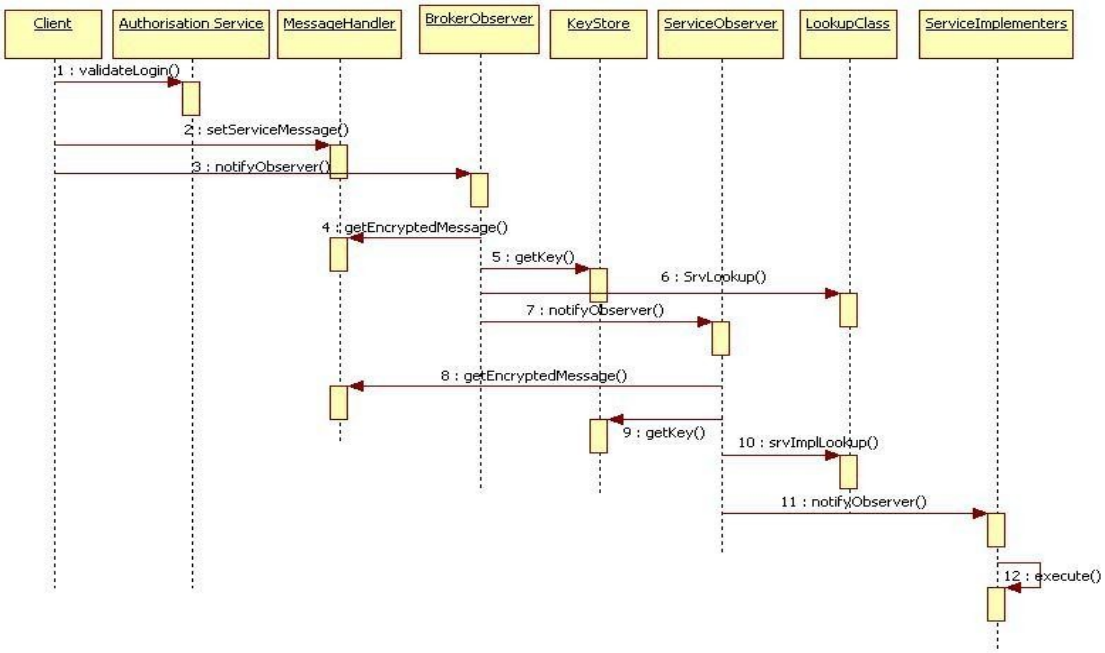

Figure 3: Sequence diagram showing message transmission from sender to receiver

5. CONSEQUENCES

5.1 BENEFITS

Uniformity: The pattern brings ease of transmission of information by the client to several other parties in a single call. It brings about uniform dissemination of information to all parties concerned via the broker interface in an encrypted manner and hence there is no ambiguity of information being sent to several parties.

Confidentiality: The secure transmission aids in privacy of the data being considered and is therefore very useful in manysituations. The customer can be rest assured that his data is not being made public and misused by unwanted parties.

Authenticity: It is also possible to add digital signatures to let the recipients verify the authenticity of messages

5.2 DRAWBACKS

Non-Uniform Message Formats: This pattern assumes proper communication channels between sender and receiver classes. But there might be disagreements in terms of message formats between them. Hence messages might not reach the end receiver as expected.

One-way Communication: This pattern only uses a one way communication from the sender to the receiver. There is no acknowledgement from the receiver to the sender for the messages delivered. Hence delivery of message is not 100% guaranteed.

6. KNOWN USES

1. IBM WebSphere Message Broker handles message transactions in distributed environments. Customers use this product to define message routing and processing for their business purpose. It provides all facilities like authentication, routing of messages, security, and processing the events that occur. This product provides ‘broker nodes’ which are used for defining message routing, mapping and conversion of message formats from one node to another. A node can be configured to receive events, process events and pass on to the next node in the chain. Many health care companies and banks use Message Broker to securely transmit the messages with outside world using this product. Some of the

customers of this product are Fortis Bank, Netherlands and Sharp Corporation, Japan. Other customers can be found on the product's website [4 & 5]

2. The architecture document of 'BEA Tuxedo Application Server' (a middleware platform for distributed computing) describes messaging paradigms that can be achieved between client and server. Out of the different modes of communication, the 'Publish-Subscribe' mode talks of a similar idea discussed in this paper. [6] The 'Application Services' layer talks of message routing, encryption, transaction etc. The 'Messaging' layer is positioned above this. This layer is responsible for communication between client and server. In this layer, there is an 'Event Broker' component which notifies events to clients (receivers). The events are triggered by the server automatically. Events could be of different types: 'Application Events' or 'System events'. For e.g.: In an Application Event, the client can be notified about the status of his bank account when the withdrawal is above a certain limit

3. Some of the companies offer real-time stock market updates. They use observer patterns to automatically update the browser content in a secure manner using https protocol. One such company is http://www.wdsoftware.com. [7] The company specializes in financial chart reporting used for intra day and end of day trade reporting of stocks. The data is displayed in applet and the updates occur automatically to the user. It uses a similar pattern discussed in paper using observer and security classes.

7. RELATED PATTERNS

Front End Interceptor: Used in routing requests in the broker layer
Publish/Subscribe: Used to broadcast events to several receivers that are subscribed
Façade: Used to hide the complexity of message handling across different classes
Strategy: Used in the conversion of message formats across service providers
MVC: Used by the observer classes as they use the model-controller pattern

8. ACKNOWLEDGEMENTS

I offer my thanks to Dr. Eduardo Fernandez for reviewing and giving positive suggestions for the paper. I also thank Mr. Raj Datta from MindTree Limited for encouraging me to submit to the conference. Also, I thank Mr.Witold Dutek from WDSoftware who shared the technical details about his product. And last but not the least, thanks to my colleagues: Nirmal, Senthilnathan for their feedback and timely reviews.

9. REFERENCES

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal., Pattern-oriented software architecture, Wiley 1996

[2] A. Braga, C. Rubira, and R. Dahab, "Tropyc: A pattern language for cryptographic object-oriented software", Chapter 16 in Pattern Languages of Program Design 4 (N. Harrison, B. Foote, and H. Rohnert, Eds.)..

[3] Eric Gamma, Richard Helm, Ralph Johnson: Design Patterns: Elements of Re-Usable Object Oriented Software

[4] IBM WebSphere Message Broker Documentation: http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r1m0/index.jsp?topic=/com.ibm.etools.mft.doc/ab20551_.htm

[5] Success stories of IBM WebSphere http://www-01.ibm.com/software/success/cssdb.nsf/softwareL2VW?OpenView&Count=30&RestrictToCategory=default_WebSphereMessageBroker&cty=en_us

[6] Oracle BEA Documentation on Tuxedo Server: http://edocs.weblogicfans.net/tuxedo/tux91/int/intatm.htm#1144565

[7] WDSOFTWARE STOCK UPDATE APPLICATION. http://www.wdsoftware.com

10. APPENDIX

The source code for the above use case is illustrated below.
LoginVO Class:

```
package util.test;

import java.util.Observable;

//this is the only observable interface in the pattern
public class LoginVO extends Observable {
   BrokerObserver brokerObs = new BrokerObserver();
   //the message to be sent
   private String serviceMessage;
   MessageHandler messageHandler = new MessageHandler();
```

```
 //the brokerobservers are added here, in our pattern only one observer is added
blic void addObserver(BrokerObserver obj){
    this.addObserver(obj);
}
```

//This method validates the user credentials
//Further it assigns the message read by the MessageHandler to the observer(s) added to the system using notifyObserver() and executes the requested service

```
public boolean executeService(){
messageHandler.setServiceMessage(serviceMessage);
    System.out.println("called the message reader "+messageHandler.getMessageHandler());
//notify the Observer, in this case BrokerFilter
    System.out.println("called the validate method");
    notifyObserver();
    return true;
}

//calls the update of next observer i.e. the broker with the required message
public void notifyObserver(){
    brokerObs.update(messageHandler);
}

}
```

MessageHandler Class:

```
public MessageHandler();
    //some custom defined Reader to read messages
    public MessageHandler getMessageHandler() {
        return messageHandler;
    }

    public void setMessageHandler(MessageHandler someHandler) {
        this.messageHandler = someHandler;
    }

//returns encrypted signed message
    public String getServiceMessage() {
                // encrypt the data
                return message;
    }
public void setServiceMessage(String serviceMessage) {
        this.serviceMessage = serviceMessage;
    }
```

BrokerObserver Class:

```
package util.test;

import java.util.*;

public class BrokerObserver extends Observable implements Observer {

//lookup for service providers based on the service description by the loginVO
private HashMap serviceProviders = new HashMap();
//holds list of observers for this subject(broker is now subject for serviceproviders)
private ArrayList observerList = new ArrayList();
 private MessageHandler messageHandler ;


//get list of service providers for the service from lookup
ServiceLookup svfilter = new ServiceLookup();
HashMap serviceProviderList =svfilter.lookupSP(this);
ServiceObserver servObs;

 //get the signed message from the handler
    public void update(Observable obj,Object ob){
        if(obj instanceof MessageHandler)
```

```
        this.setMessageHandler((MessageHandler) obj);
        System.out.println("called the broker update method");
        String brokerMessage = decryptBrokerMessage(messageHandler.getServiceMessage());
        addObserver(brokerMessage);
        notifyObserver();
    }


private void setMessageHandler(MessageHandler messageHandler) {this.messageHandler=messageHandler;   }
    private MessageHandler getMessageHandler() {return messageHandler; }
}


//decrypt messages using the key obtained
    private  String decryptMessage(String encryptedMessage){
    //decrypt message and route to service provider
        return KeyStore.getKey(this);

    }

public void addObserver(String message){
        //adding a test service observer object  from message received
        serviceProviderList.put(new ServiceObserver(),new ServiceObserver());
        Iterator it =   serviceProviderList.values().iterator();
        while(it.hasNext())
            observerList.add(it.next());
    }
    public void notifyObserver(){
        Iterator it=serviceProviderList.values().iterator();

        while( it.hasNext() ) {
                servObs = ( ServiceObserver )it.next();
                System.out.println("service observers added in loop");
                servObs.update(this, messageHandler );
        }
    }
}
```

ServiceLookup Class:

```
package util.test;
import java.util.HashMap;

public class ServiceLookup {
     public HashMap lookupSP(BrokerObserver brokObs){
        HashMap srvProvider = new HashMap();
        //look up code for service providers
        return srvProvider;
    }


        public HashMap lookupSP(ServiceObserver servObs){
        HashMap srvImpl = new HashMap();
        //look up code for service implementers
        return srvImpl;
        }
}
```

ServiceObserver Class:

```
package util.test;

import java.util.Observer;
import java.util.Observable;
import java.util.HashMap;
import java.util.Iterator;
public  class ServiceObserver extends Observable implements Observer {
    MessageHandler messageHandler ;
    HashMap servImpl ;

private HashMap lookUPServImpl(){
```

```
        String message = decryptServMessage();
        //based on message lookup service Impl
        servImpl = new ServiceLookup().lookupSP(this);
        return servImpl;
    }


     private String decryptServMessage(String encryptedMessage){
        //decrypt message and route to service provider
        return KeyStore.getKey(this);
     }

//add test service implementers

    public void update(Observable obs,Object obj){
        lookUPServImpl();
        servImpl.put(new ServImpl_1(),new ServImpl_1());
        //get list of serviceimplementers
        Iterator it = servImpl.keySet().iterator();

    if(obs instanceof BrokerObserver){
        //get broker and client VO
        this.setClientInfo(((BrokerObserver)obs).getClientVO());
        //execute relevant service methods
        while(it.hasNext()){
        ServiceImplInterface sp =(ServiceImplInterface)it.next();
            System.out.println("called the update method in ServiceObserver");
            sp.execute(messageHandler.getMessage());
        }
    }
    }

private void setMessageHandler(MessageHandler messageHandler) {this.messageHandler=messageHandler;   }
    private MessageHandler getMessageHandler() {return messageHandler; }
}
```

ServImplInterface Interface:

```
package util.test;
public interface ServiceImplInterface {
    public void execute(MessageHandler messageHandler) ;
}
```

ServImpl Class:

```
package util.test;

public class ServImpl implements ServiceImplInterface  {
    public void execute(MessageHandler msgHandler){
    System.out.println("called the service implementer method for client");
    //implementation of actual service here
}
}
```

KeyStore Utility Class:

```
package util.test;

public class KeyStore   {
    private static String servObsKey;
    private static String brokerObsKey;
    private static String servImplKey;


    public static String getKey(ServiceObserver sobs){
        //some implementation
        return servObsKey;
    }
     public static String getKey(BrokerObserver bobs){
        //some implementation
        return brokerObsKey;
```

```
    }
}
```

Test Class:
This is a tester client class to test the above set of classes for running a secure chained sequence of events.

```
package util.test;

public class TestChainedObserver  {

    public static void main (String args[]){
        try{
        LoginVO loginVO = new LoginVO();
        MessageHandler messageHandler = new MessageHandler();
       messageHandler.setMessage("Test message from client- update personal Info");
        loginVO.setMessageHandler(messageHandler);
        loginVO.executeService();
        }catch(Exception ex){
            //catch and log exceptions
        }
    }
}
```

Output after running the Test class in Eclipse IDE:

```
called the message Test message from client- update personal Info
called the validate method
called the broker update method
service observers added in loop
called the update method in ServiceObserver
called the service implementor method for client

Process finished with exit code 0
```