Java Based E-Commerce Middleware

Sub Ramakrishnan
Department of Computer Science
Bowling Green State University
Bowling Green, OH 43403-0214

Abstract

Organizations that sell products over the Internet usually have a product database located at or near the web hosting server. Web clients use SSL to connect to the web server and order products, which are then entered into the product database. Since web hosting services may be outsourced, the order information needs to be transmitted securely to the corporate database of the enterprise that sells these products. This paper concerns the development of a secure middleware application in Java that connects the web hosting database with the corporate backend database [9]. We discuss the design details of the middleware and implementation issues of some of the software pieces of this puzzle.

1 Background

Objects are valuable for communicating with students and colleagues about the structure and dynamics of a system being studied. The objects of the system can be divided into groups where each group shares related characteristics. Object Oriented Programming (OOP) provides a clear set of program design guidelines: (i) Identify the object classes that make up the system. (ii) Define the interface (i.e. the attributes and methods) of each object class. (iii) Implement the class. (iv) Build the software that should manipulate the objects according to the interface definition. OOP enables modular design because each object class encapsulates a data structure and the operations that manipulate that data structure or abstract data type. Interfaces help separate the services from implementation details. They also promote software reuse since developers can determine whether a class provides the necessary services (through the interface) without having to verify and inspect all the source code with respect to the implementation. In addition, one can use inheritance to extend or refine an existing base class.

Java [1] is becoming the preferred OOP language of choice for both stand-alone and web enabled applications as compared to earlier OOP languages such as Small Talk [5]. Java code compiled on one machine architecture can be ported to another architecture without modification. With the widespread availability of Java code on the web, applications can be built by reusing components found elsewhere. Though portability and reusability are strong traits of object-oriented languages few languages embrace these two features as well as Java. On the downside, Java startup and runtime costs are higher as compared to other fully compiled languages. Java runs at least an order of magnitude slower than C code. However, Java is catching up and just in time (JIT) compilers should help improve runtime performance. Due to the simplicity of Java language primitives, program design, debug and maintenance cycles are substantially lower than that of other languages. Java Cryptographic Extension (JCE) addresses secure computing needs by providing plug-in cryptographic libraries and seamless addition of a number of security components and services such as message digests, digital signatures, random number generators and algorithms for cryptography.

Commerce over the Internet is growing at an exponential rate. Several vendors are providing secure solutions for B2B transactions. For example, in an Internet supply chain system, [2] provides mechanisms for separation of data from different owners along the entire path that the data takes from its source to a recipient. Products such as Web Server 4D [13] allow manufactureres to sell products or services and include features such as shopping cart and automatic credit card processing. If the website is outsourced, the manufacturereruns a a program such as Timbuktu, locally, to remotely control the website end. IBM and Tivoli's SecureWay [5] family of solutions address a number of issues in providing for a secure, managed environment for web enabled e-business.

Today's Internet enabled market place allows customers to connect to a web server and place order for goods and services over the web. The order sits in a local database located behind the web server. These web services are typically outsourced to third parties, known as *web hosting services* (also known as the storefront). However, the goods and service provider *enterprise* operations (its computing and other infrastructure resources) are usually located elsewhere in the Internet, often behind the enterprise firewall.

The web client (user) talks to the web hosting services' web server using a secure, SSL, connection. The web hosting service manages a web server and a local database. This paper concerns the design and development of a secure e-commerce Middleware in Java that allows for secure transfer of data from this database to the enterprise backend database. In an earlier paper [9], we gave an overview of such a middleware Java application. (see Figure 1). Our contribution is the development of a secure client-server application to link the two databases; the java client at the web hosting end securely transfers orders from the (web hosting services') local database, to the corporate database via the server at the corporate enterprise, as seen in Figure 1. The advantage of this approach is that existing programs that pick up and process the order from the corporate database are unaffected and continue to function normally.

This paper provides additional details of this middleware. Section 2 gives a schematic of the application. The software and cryptographic implementation issues are addressed in Section 3.

2 A Middleware E-Commerce Application

The middleware client and server use public key cryptography to do authentication with integrity protection [6]. To provide added security, the private key is not stored in raw form in the local disk; instead a pass-phrase based encryption mechanism is used to store the private key file, which is encrypted using a key derived from a (rather long) pass-phrase string that the user types in, just once, when the middleware is brought on-line at either end. Following that, they establish a session key and use symmetric encryption to encrypt their communication. The client and server know each other public-key, as well as their respective private key. The client's public key may also be obtained at runtime if the first message from client is modified to include the client's public key certificate. This is especially useful if the same server has to interact with multiple web hosting services' clients at the same time.

The client and server communicate with an email server periodically and send information to the enterprise about activity logs and order statistics. Figure 2 shows the Java code for an email client. As can be noted the main module seeks connection with the *smtp* server (usually available on port number 25) and sends the email to it according to the smtp protocol conventions. Command line arguments specify the four required parts: mail server name, sender email address, receiver email address, and the message string (email) itself.

Details of middleware process flow are illustrated in Figure 3 (from [9]). The client resides at the web hosting services while the server resides, behind a firewall, within the enterprise. The client provides primitives to access the web hosting services' database while the server provides primitives to access the enterprise backend database. The client and server communication is over a socket, which is protected for privacy. The socket connection (between the client and server) stays open as long as there are active orders, and is disconnected during long periods of no activity.

The middleware client employs a threaded model, and consists of the main thread and three other threads: (i) the javaTriggerServer thread gathers new orderIDs and drops them in a line (queue) for processing by javaPullOrderThread. (ii) javaPullOrderThread picks the next orderID from the line and pulls the corresponding order information (header and detail records for this orderID) from the web hosting services' database using a JDBC or JDBC-ODBC bridge connection, and securely transmits them to the server at the enterprise. In addition, the thread processes acknowledgement information from the middleware server and updates a local database table, orderStatus [8]. (iii) javaKeyManagementThread is responsible for public-key based authentication exchange protocol at the middleware client. Then, it cooperates with the middleware server and generates a session key for use by javaPullOrderThread. The main thread creates these three threads when needed, establishes the socket connection with the server and tears it down when it is no longer needed. Our model allows de-coupling of the functions and provides for concurrency within the system.

The middleware server (bottom half of Figure 3) at the enterprise, is located behind the corporate firewall, and has a master role in our architecture. The firewall (a separate machine) is normally configured to allow the socket connection at a specific port, at an Internet visible IP address. (Thanks to the firewall, IP address at the enterprise side of the firewall is invisible to the Internet community.) The server is multi-threaded (not shown for brevity) and listens for connection, at a specific port, from the client. It then does authentication negotiation and works with the client in setting up a session key for communication of payload and acknowledgement. Though the server may be communicating with multiple middleware clients at the same time, all of the server *threads* share the same (single) connection with the backend database at the enterprise mainframe. For improved fault-tolerance, the server middleware modules are located in a machine distinct from the database machine (enterprise mainframe).

The client at the web hosting service can detect that an order is in by using a database trigger (as in Figure 3; the trigger launches *javaTriggerClient* that communicates the orderID to *javaTriggerServer*) or by periodically polling the web hosting services' database for a new order. As noted in [9], we prefer the polling mechanism though it delays order processing by a few seconds. However, if the polling frequency is every second, then the order is extracted within one second of an order arrival.

The client and server maintain log files that track events as perceived by the respective sides. A snapshot of the application log file, at client and server ends, is shown in Figure 4.

3 Implementation Issues

JCE [11, 8] comes with a security provider, known as *SunJCE*. We wrote a Java class, simpleDESEncryptAndDecrypt (Appendix A) to demonstrate the use of a Cipher, from the SunJCE provider, for symmetric key encryption. The lines are numbered in Appendix A so we can discuss them in the body of this main text. The main program covers lines 14–25. Line 17 is a convenient way to add the SunJCE at runtime. Member function createSymmetricKey (lines 26-34) creates a DES key by getting a *DES* instance from the engine class, *KeyGenerator*. Line 19-20 of the main then calls the function, desEncryptThisMessage (lines 36–47) to encrypt the message HelloWorldMessage. The body of desEncryptThisMessage function uses the methods of the engine class, *Cipher*, to do encryption. It creates a cipher instance with the ECB mode and PKCS padding (line 40), specifies whether encryption or decryption is needed and supplies the key (line 41), and finally feeds the message to get the cipher out (line 45). The *init* method (line 41) resets the engine to an initial state discarding any previous data in the engine. This method has other overloaded derivatives that handle requirements such as initialization vector when the Cipher needs one. Function desDecryptThisCipher (lines 48-56) serves a role complementary to desEncryptThisMessage, decrypting incoming cipher data. Note the symmetry in the code for these two functions. Line 23 of main prints the decrypted output.

A variant of Appendix A could be used to construct a passphrase based encryption algorithm that would be required for storing key files on disk (see Section 2, first paragraph).

Java's Vector class allows elements of different type (including objects) to be added to a vector instance. The vector itself is an object; the entire vector can be sent as an object following encryption over the socket. This process can be made simpler by wrapping an encryption filter over the socket; any serializable object that is sent over the socket would be automatically (implicitly) encrypted according to the encryption wrapper algorithm, as illustrated in Figure 5. Method setUpFilterStreams of IpcClientServer sets up the required filters. Method sendObjectEncryptedAndSealed shows how to send a sealed and DES encrypted version of the object, objectToBeEncrypted, First two lines in this function is the (DES) encryption aspects (from Appendix A), while the third line provides a SealedObject filter around the encrypted version of objectToBeEncrypted. The *try block* sends the sealed object out using the writeObject method of ObjectOutputStream, which is a filter over the socket.

One can write a provider, similar to SunJCE, with a set of classes to implement public key cryptographic algorithms. A provider has many classes, each class implements an algorithm that has a type and a name. The type says whether it is Cipher, Signature etc, and the name gives algorithm specifics, such as *DES/ECB/PKCS5Padding*. An example of a provider specification, named as *OrderProcess*, is given in Figure 6. The *put* primitive names the algorithm and the corresponding class that implements it. The name of the provider can also be added to an application at runtime by a call similar to Line 17 of Appendix A. Additional resources on writing a provider are found at Sun website [12].

Implementation of a cipher engine for a public-key cryptographic system (such as the algorithm of ElGamal [3]) or a symmetric key cryptography system is complicated. For brevity, we only identify the methods behind such an implementation [7, 8]. The engine class, such as Cipher, is designed to allow users to employ their own provider architecture, for example implementation of RSA. In order to write such providers, the engine classes support an additional interface called the security provider Interface (SPI). The SPI is a set of abstract methods that every engine provider must implement in order to fulfill its part in providing a particular operation. The advantage of this approach is that each provider has same set of methods in it, though the functionality of these methods will be different depending on what algorithm is used for implementation. One can implement a cipher by extending the corresponding SPI, CipherSPI. Implementing a provider, as we did in Figure 5, and using an already written implementation of a Cipher as we did in building simpleDESEncryptAndDecrypt (see Appendix A), is relatively straightforward. However, implementing the classes (for example, *MyAlgorithmCipher* of Figure 6) that must accompany the provider is much harder.

An implementation of a Cipher class provides the functionality for the corresponding abstract methods defined in class CipherSPI: (i) engineSetMode – the mode, for example, ECB for DES, (ii) engineSetPadding – padding scheme used, (iii) engineGetBlockSize – the block size for the engine. Ciphers, such as DES, that allow padding can process input that is not a multiple of block size. Some algorithms, such as ElGamal have neither a mode nor padding options so implementation of these two methods can be easily accomplished. (iv) engineGetIV – the vector to initialize the cipher, (v) engineInit – the direction of transformation, whether plaintext to cipher (ENCRYPT_MODE) or cipher to plaintext (DECRYPT_MODE). There are other overloaded derivatives, of engineInit, to initialize the cryptographic engine. (vi) engineUpdate – does the bulk of the work, and generates cipher or plaintext as output. engineUpdate processes any full blocks in the input, and the last incomplete block, if any, is left in the buffer for processing by the next engineUpdate or engineDoFinal call.

The astute reader would have noticed that methods in CipherSPI have names similar to that in Cipher except for the prefix, *engine*, in methods of CipherSPI class. Usually, there is very little intelligence in the methods of the Cipher class itself; virtually most of the methods in Cipher class are simply pass through calls to the corresponding methods in the SPI. See [8] (pp. 308) for an example of a simple cipher implementation.

The BigInteger class of java.math provides a number of features such as generation of prime numbers of different width with certain probability, doing exponent and modulo arithmetic. For example, from [3, 8] the key pairs for El Gamal are generated with the public key being $<g, p, T>$ and the private or secret key being S, where $g^S \bmod p = T$, as in Diffie-Hellman. The modulus, p, is random and is prime. The public parts, g and T are two other random numbers, and are less than p

One can generate a prime number, p, with a given width (or strength, say 512 bits) using the elegant probabilistic algorithm due to Solovay and Strassen [10]. To test a large number, p, for primality, this algorithm picks a random number, a, form the uniform distribution on $\{1, \ldots, p\text{-}1\}$ and tests whether

$$\gcd(a, p) = 1 \text{ and } J(a, p) = a^{(p-1)/2} \bmod p.$$

where $J(a, p)$ is the Jacobi symbol and has a value in $\{-1, 1\}$. If the above test holds for numberOfTests (say, 10) randomly chosen values of a, then p is almost certainly prime; there is a negligible chance of one in 2^{10} that p is not prime. One can certainly better the odds by increasing the value of numberOfTests. Fortunately, all of this computation is elegantly captured in one of the Java BigInteger constructors, as shown below.

p = new BigInteger (strengthOfKey, numberOfTests, anInstanceOfARandomNumber)

where strengthOfKey is the width of the desired prime number (say, equal to 512 bits), and anInstanceOfARandomNumber is a random number that can be obtained by instantiating SecureRandom(). Returning to the computation of key pairs, the public part, T, can be computed using one of the BigInteger methods, as shown below.

$$T = \text{g.modPow}(S, p)$$

where g and S are random numbers of type BigInteger.

For brevity, we could not discuss the full implementation of the cryptographic engines.

There are some providers outside of United States [14] who have Java based public key implementations. However, the author is unaware of the scope and applicability of such tools.

References

1 Cornell, G et al., Core Java, SunSoft press/Prentice hall 1996.

2 Data Driven Access Control Technology. White Paper. http://www.cryptek.com

3 ElGamal, T. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms, *IEEE Transactions on Information Theory*, Vol. 31, pp. 469-472, 1985.

4 Goldberg, A. et al., Smalltalk-80: the Language and its Implementation, Addison-Wesley, 1983.

5 IBM and Trivoli, http://www.tivoli.com/products/solutions/security/news.html, Using Tivoli® SecureWay® to Manage e-business Security.

6 Kaufman, C et. al., Network Security: Private Communications in a Public World, Prentice Hall. 1995.

7 J Knudson, J. Java Cryptography. O'Reilly and Associates. May 1998.

8 Oaks, S. Java Security. O'Reilly and Associates, February 1999.

9 Ramakrishnan, S. Architecture of a Secure Order Entry System in Java, International Systems Security Engineering Conference 2001, Orlando, FL, February 2001, *to appear*.

10 Solovay R, and Strassen, V. "A Fast Monte Carlo Test for Primality," *SIAM J. Computing*, vol. 6, no. 1, pp. 84-85, March. 1977.

11 Sun Microsystems, http://java.sun.com/products/jce/

12 Sun Microsystems, http://java.sun.com/products/jce/doc/guide/HowToImplAProvider.html

13 WS4D/eCommerce, MDG Computer Services, Inc, Bartlett, IL.

14 The Cryptix Foundation Limited, http://www.cryptix.org

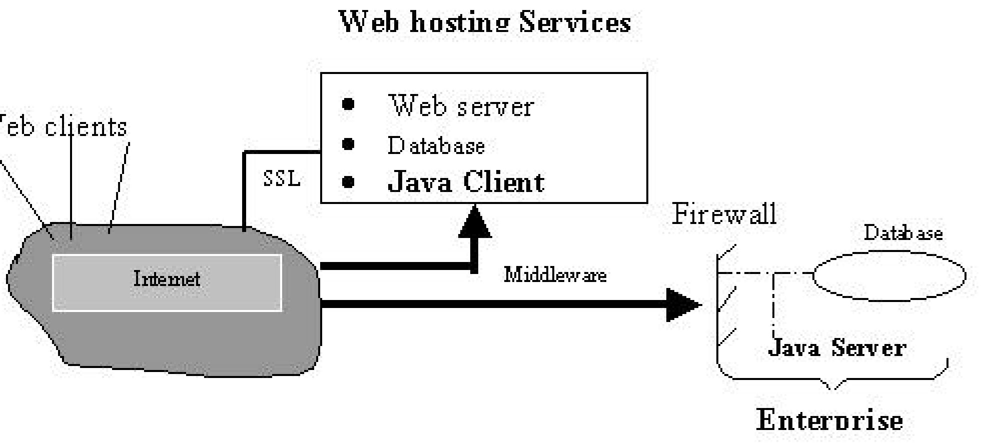
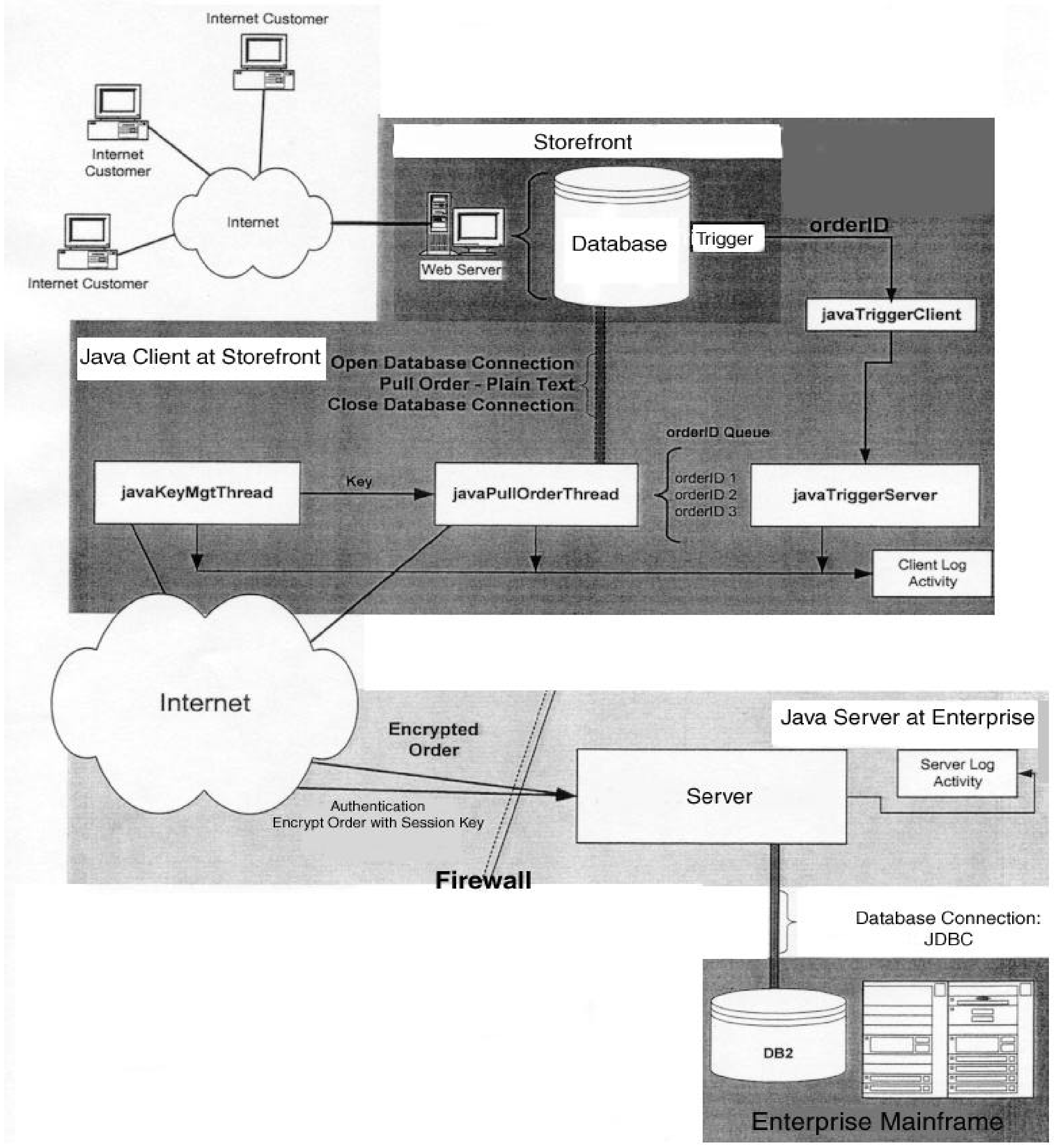


Figure 1: Architectural View of Order Flow [8]

```
//          usage: java smtpoExample  mailServeName senderAddress receiverAddress messageText
public class smtpExample {
        public static void main(String[] args) throws IOException {
                smtpMail  smtpMailHandle = new smtpMail (args [0]);
                smtpMailHandle.login();
                smtpMailHandle.send( args[1], args[2], args[3]  );
                smtpMailHandle.quit();
        }
}
class smtpMail {
        Socket sendMailSocket = null;  PrintWriter sendToSmtp;   BufferedReader rcvFromSmtp ;
        public smtpMail (String smtpServer) throws IOException {
                sendMailSocket= new Socket (smtpServer, 25);
                sendToSmtp = new PrintWriter ( sendMailSocket.getOutputStream(), true);
                rcvFromSmtp = new BufferedReader ( new InputStreamReader ( sendMailSocket.getInputStream()));
                getSmtpResponse();
        }
        public void login () throws IOException {
                String address =  sendMailSocket.getInetAddress().getHostAddress();
                sendToSmtp.println("HELO " + address);
                getSmtpResponse();
        }
        public void send (String sender, String receiver, String message) throws IOException {
                sendToSmtp.println("MAIL FROM: " + sender); getSmtpResponse();
                sendToSmtp.println("RCPT TO: " + receiver); getSmtpResponse();
                sendToSmtp.println("DATA" ); sendToSmtp.println( );
                sendToSmtp.println( message);  sendToSmtp.println( );  sendToSmtp.println( ".");
                sendToSmtp.flush(); getSmtpResponse();
        }
        public void quit () throws IOException {
                sendToSmtp.println("QUIT " );
                try { getSmtpResponse();}
                catch (Exception error) {}
                sendMailSocket.close(); rcvFromSmtp.close(); sendToSmtp.close();
        }
        protected String getSmtpResponse () throws IOException {
                String line;
                do line = rcvFromSmtp.readLine();
                while (rcvFromSmtp.ready() );
                System.out.println (line); return null;
        }
}
```

Figure 2: SMTP Mailer Client

Figure 3: Process Flow and Application Components



```
Middleware Client Log Activity
10/26/99 10:03 PM Program just started...
10/26/99 10:03 PM Connected to server inside firewall
10/26/99 10:03 PM I am (client) authenticated!
10/26/99 10:03 PM New symmetric key generated
10/26/99 10:03 PM Key manager sleeping...OK
10/26/99 10:04 PM Masked IP Address-860487226 Order trigger!
10/26/99 10:04 PM Masked IP Address-860487226 Order being pulled..
10/26/99 10:04 PM Masked IP Address-860487226 Order done...
Middleware Server Log Activity
10/26/99 10:01 PM Program just started...
10/26/99 10:01 PM Invalid LogIn! USERNAME
10/26/99 10:02 PM Program just started...
10/26/99 10:02 PM Connected to ServerDatabase.
10/26/99 10:03 PM Connected to outside firewall  Client...
10/26/99 10:03 PM Authentication complete!
10/26/99 10:04 PM Masked IP Address-860487226 New Order.
10/26/99 10:04 PM To dump line items: 2
10/26/99 10:04 PM Masked IP Address-860487226 Order done...
```

Figure 4: Snapshot of Log Information

```
package ipcAbstractSealed;
import java.net.*;
import java.io.*;
import javax.crypto.*;
/*          The class instance contains (Serializable) object which are encrypted. A Vector
             is a serializable object that can be contained, in encrypted form, in an instance of the SealedObject    */
public class IpcClientServer   {
            public void setUpFilterStreams   () {
                        try {
                                    OutputStream out = theSocket.getOutputStream();
                                    ObjectOutputStream  easyGetOutputObject = new  ObjectOutputStream (out);
                         }
                        catch (IOException fileHandleError) {}
            }
            public  SealedObject sendObjectEncryptedAndSealed (Object   objectToBeEncrypted)
                        throws NoSuchAlgorithmException, Exception {
                        cipherHandle.init (Cipher.ENCRYPT_MODE, ... );
                        SealedObject encryptedAndSealedForIPC =
                                    new SealedObject ((Serializable)objectToBeEncrypted, cipherHandle);
                        try {
                                    easyGetOutputObject.writeObject (encryptedAndSealedForIPC);
                                    easyGetOutputObject.flush();
                        }
                        catch (IOException fileHandleError) {}
              }
}
```

Figure 5: Process for Transmission of Encrypted Serializable Objects

```
package HomeDir.OrderProcess.crypto;
import java.security.*;
public class Provider extends java.security.Provider {
     public Provider () {
            super  ("OrderProcess", 1.2, " OrderProcess 's Cryptography provider ");
            put("KeyPairGenerator.MyAlgorithm","HomeDir.OrderProcess.crypto.MyAlgorithmlKeyPairGenerator")
            put ("Cipher MyAlgorithm ",  "HomeDir.OrderProcess.crypto.MyAlgorithmCipher");
       }
}
```

Figure 6: An Example of a Provider Specification in Java

```
import java.io.*;
import java.net.*;
import java.security.*;
import javax.crypto.*;
import sun.misc.*;
/**

Do DES encryption and decryption of a "HelloWorldMessage"
            Use SunJCE Provider
*/

public class simpleDESEncryptAndDecrypt {

  public static void main (String[] args)
            throws NoSuchAlgorithmException, Exception {

            Security.addProvider(new com.sun.crypto.provider.SunJCE());
            Key desKeyInstance  = createSymmetricKey ();
            byte[]  cipherText =
                        desEncryptThisMessage  (  "HelloWorldMesage" , desKeyInstance   ) ;
            String plainTextAfterDecrption =
                        desDecryptThisCipher  ( cipherText  ,  desKeyInstance   ) ;
            System.out.println("Decrypted Message  ..." + plainTextAfterDecrption );

  }
  public static Key  createSymmetricKey ( )
                        throws NoSuchAlgorithmException, Exception {
                        Key keyName;

                        KeyGenerator desGenerator=  KeyGenerator.getInstance ("DES");
                        desGenerator.init(new SecureRandom());
                        keyName = desGenerator.generateKey();
                        return keyName;
  }

  public static byte[] desEncryptThisMessage  (
            String plainTextMessage , Key keyName)
            throws NoSuchAlgorithmException, Exception {

                        Cipher cipher = Cipher.getInstance ("DES/ECB/PKCS5Padding");
                        cipher.init (Cipher.ENCRYPT_MODE, keyName);
                        String plainText = plainTextMessage;

                        byte[] bytePlainText = plainText.getBytes("UTF8");
                        byte[] cipherText= cipher.doFinal(bytePlainText );
                        return cipherText;
  }
  public static String desDecryptThisCipher  (byte[] cipherText , Key keyName) {
                        System.out.println("will do Decryption");
                        Cipher cipher = Cipher.getInstance ("DES/ECB/PKCS5Padding");
                        cipher.init (Cipher.DECRYPT_MODE, keyName);
                        byte[] almostPlainText = cipher.doFinal(cipherText);
                        String plainTextAfterDecryption = new String (almostPlainText, "UTF8");
                        return plainTextAfterDecryption;
  }
}
```

Appendix A: Use of SunJCE Provider Architecture for DES Encryption and Decryption