Lavigne Tom

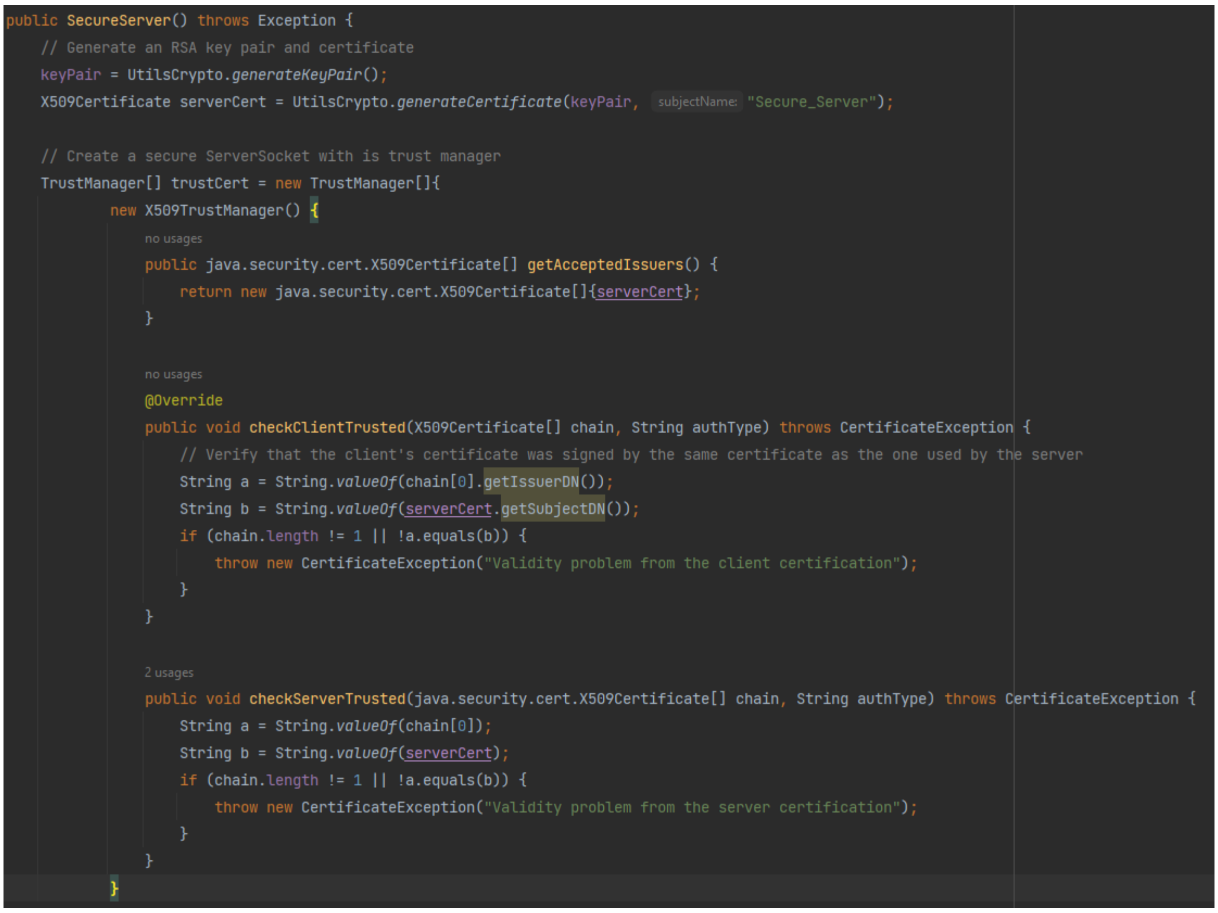
CSU33032 Project 2
Securing Social Media Applications

1. Introduction to SecureServer Functionality:

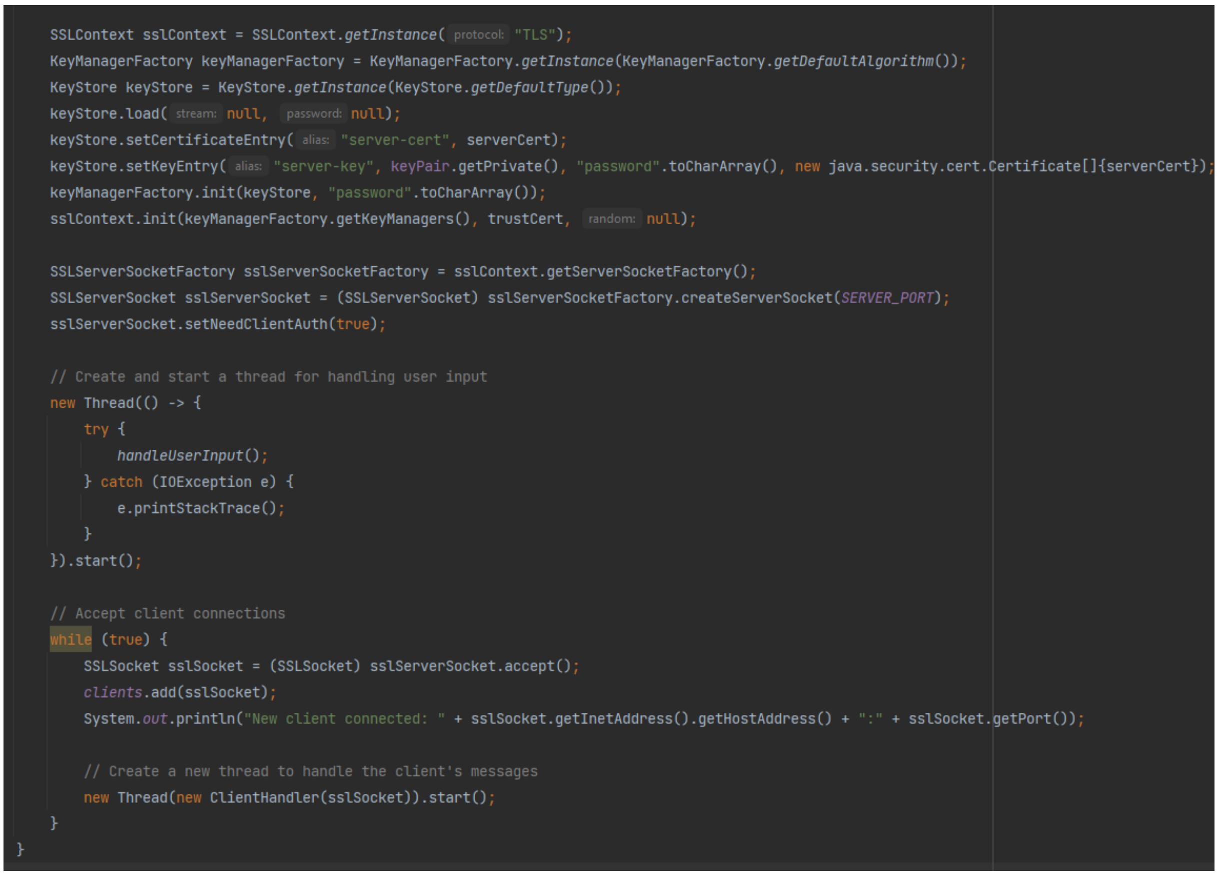
This class represents a secure server that uses SSL/TLS for secure communication. When a client connects, the server verifies the client's identity using a public-key certificate. If the client's certificate is valid, the server adds the client to a list of connected clients.. Then, for each message it receives that has been encrypted with its public key, it decrypts it with its private key and verifies the signature with the client's public key and the message. He will then re-encrypt the message with his private key if the client is authorized to read and sign the message.

It maintains two lists: clients for all connected clients, and securedClients for clients that have been added to a secured group. Here is the detail of the protocol :

SecureServer Constructor: The constructor initializes the server. It generates an RSA key pair and a self-signed certificate for the server. It then sets up a SSLServerSocket to accept client connections. The constructor also starts a new thread to handle user input for adding and removing clients from the secured group. In an infinite loop, it accepts new client connections, adds them to the clients list, and starts a new thread to handle each client's messages.

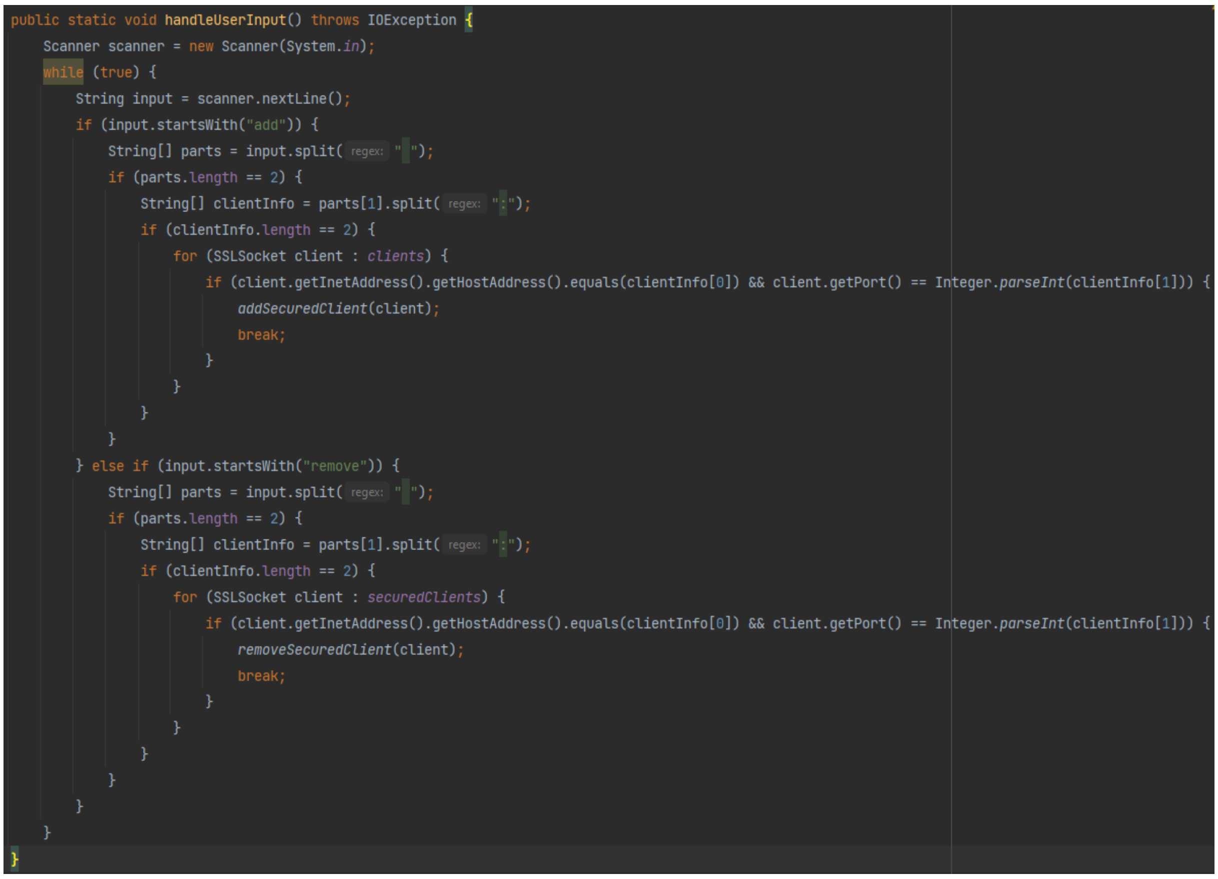


```
public SecureServer() throws Exception {
    // Generate an RSA key pair and certificate
    keyPair = UtilsCrypto.generateKeyPair();
    X509Certificate serverCert = UtilsCrypto.generateCertificate(keyPair,  subjectName: "Secure_Server");

    // Create a secure ServerSocket with is trust manager
    TrustManager[] trustCert = new TrustManager[]{
            new X509TrustManager() {
                no usages
                public java.security.cert.X509Certificate[] getAcceptedIssuers() {
                    return new java.security.cert.X509Certificate[]{serverCert};
                }

                no usages
                @Override
                public void checkClientTrusted(X509Certificate[] chain, String authType) throws CertificateException {
                    // Verify that the client's certificate was signed by the same certificate as the one used by the server
                    String a = String.valueOf(chain[0].getIssuerDN());
                    String b = String.valueOf(serverCert.getSubjectDN());
                    if (chain.length != 1 || !a.equals(b)) {
                        throw new CertificateException("Validity problem from the client certification");
                    }
                }

                2 usages
                public void checkServerTrusted(java.security.cert.X509Certificate[] chain, String authType) throws CertificateException {
                    String a = String.valueOf(chain[0]);
                    String b = String.valueOf(serverCert);
                    if (chain.length != 1 || !a.equals(b)) {
                        throw new CertificateException("Validity problem from the server certification");
                    }
                }
            }
```



```java
    SSLContext sslContext = SSLContext.getInstance( protocol: "TLS");
    KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
    KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
    keyStore.load( stream: null,  password: null);
    keyStore.setCertificateEntry( alias: "server-cert", serverCert);
    keyStore.setKeyEntry( alias: "server-key", keyPair.getPrivate(), "password".toCharArray(), new java.security.cert.Certificate[]{serverCert});
    keyManagerFactory.init(keyStore, "password".toCharArray());
    sslContext.init(keyManagerFactory.getKeyManagers(), trustCert,  random: null);

    SSLServerSocketFactory sslServerSocketFactory = sslContext.getServerSocketFactory();
    SSLServerSocket sslServerSocket = (SSLServerSocket) sslServerSocketFactory.createServerSocket(SERVER_PORT);
    sslServerSocket.setNeedClientAuth(true);

    // Create and start a thread for handling user input
    new Thread(() -> {
        try {
            handleUserInput();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }).start();

    // Accept client connections
    while (true) {
        SSLSocket sslSocket = (SSLSocket) sslServerSocket.accept();
        clients.add(sslSocket);
        System.out.println("New client connected: " + sslSocket.getInetAddress().getHostAddress() + ":" + sslSocket.getPort());

        // Create a new thread to handle the client's messages
        new Thread(new ClientHandler(sslSocket)).start();
    }
}
```

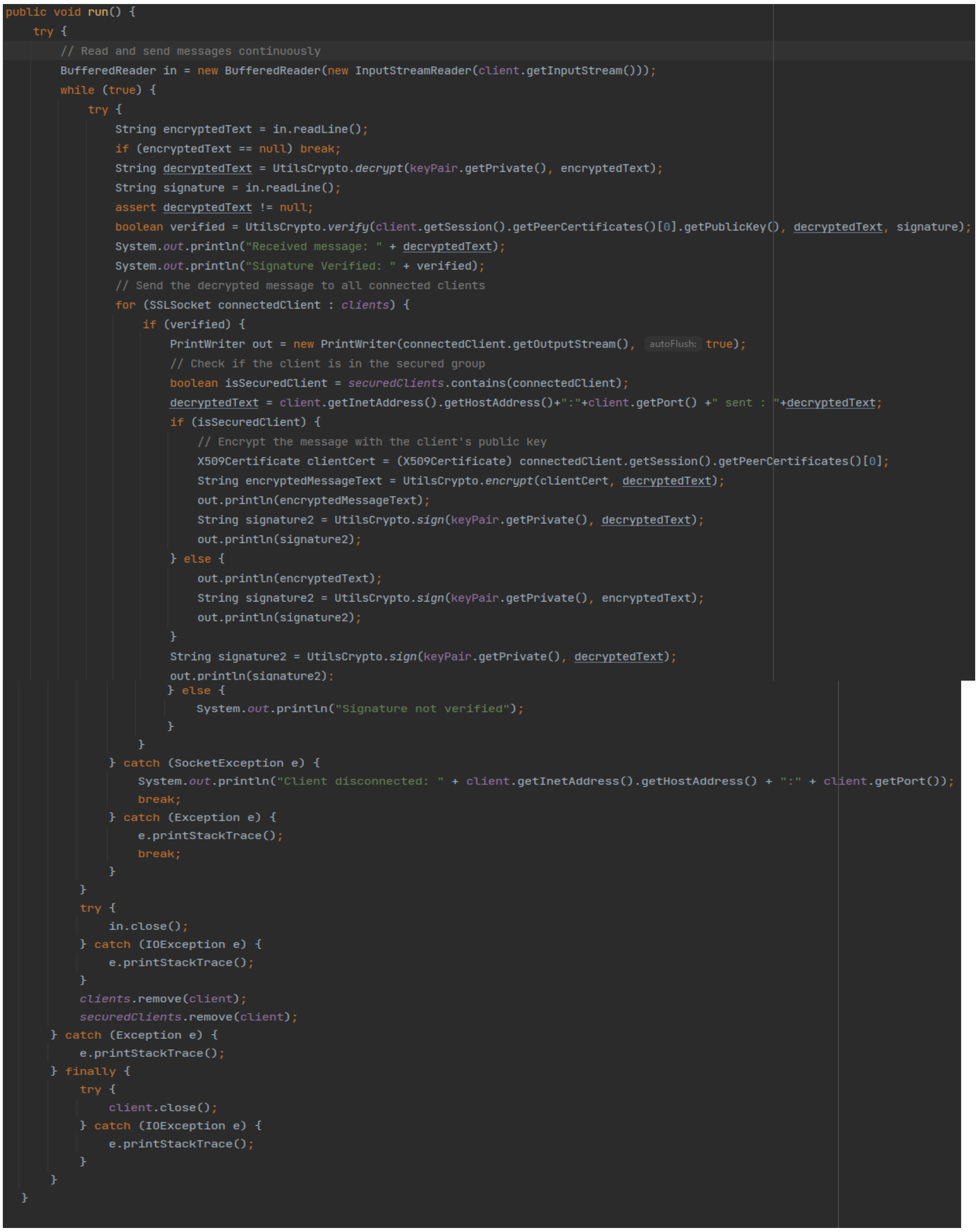
handleUserInput(): This static method handles user input for adding and removing clients from the secured group. It reads input from the console and parses commands to add or remove clients based on their IP address and port.



```java
public static void handleUserInput() throws IOException {
    Scanner scanner = new Scanner(System.in);
    while (true) {
        String input = scanner.nextLine();
        if (input.startsWith("add")) {
            String[] parts = input.split( regex: " ");
            if (parts.length == 2) {
                String[] clientInfo = parts[1].split( regex: ":");
                if (clientInfo.length == 2) {
                    for (SSLSocket client : clients) {
                        if (client.getInetAddress().getHostAddress().equals(clientInfo[0]) && client.getPort() == Integer.parseInt(clientInfo[1])) {
                            addSecuredClient(client);
                            break;
                        }
                    }
                }
            }
        } else if (input.startsWith("remove")) {
            String[] parts = input.split( regex: " ");
            if (parts.length == 2) {
                String[] clientInfo = parts[1].split( regex: ":");
                if (clientInfo.length == 2) {
                    for (SSLSocket client : securedClients) {
                        if (client.getInetAddress().getHostAddress().equals(clientInfo[0]) && client.getPort() == Integer.parseInt(clientInfo[1])) {
                            removeSecuredClient(client);
                            break;
                        }
                    }
                }
            }
        }
    }
}
```

The **ClientHandler Class** inner class represents a handler for each client connection. It implements the Runnable interface so that each client can be handled in a separate thread. The constructor initializes the ClientHandler with the SSLSocket for the client it will handle.

ClientHandler.run(): This method is called when the ClientHandler thread is started. It reads messages from the client, decrypts them with the server's private key, verifies their signatures, and then sends them to all connected clients with his signature. If a client is in the securedClients list, the message is encrypted with the client's public key before being sent and if not, the server let the message encrypted by his public key. It also handle if a client disconnects.

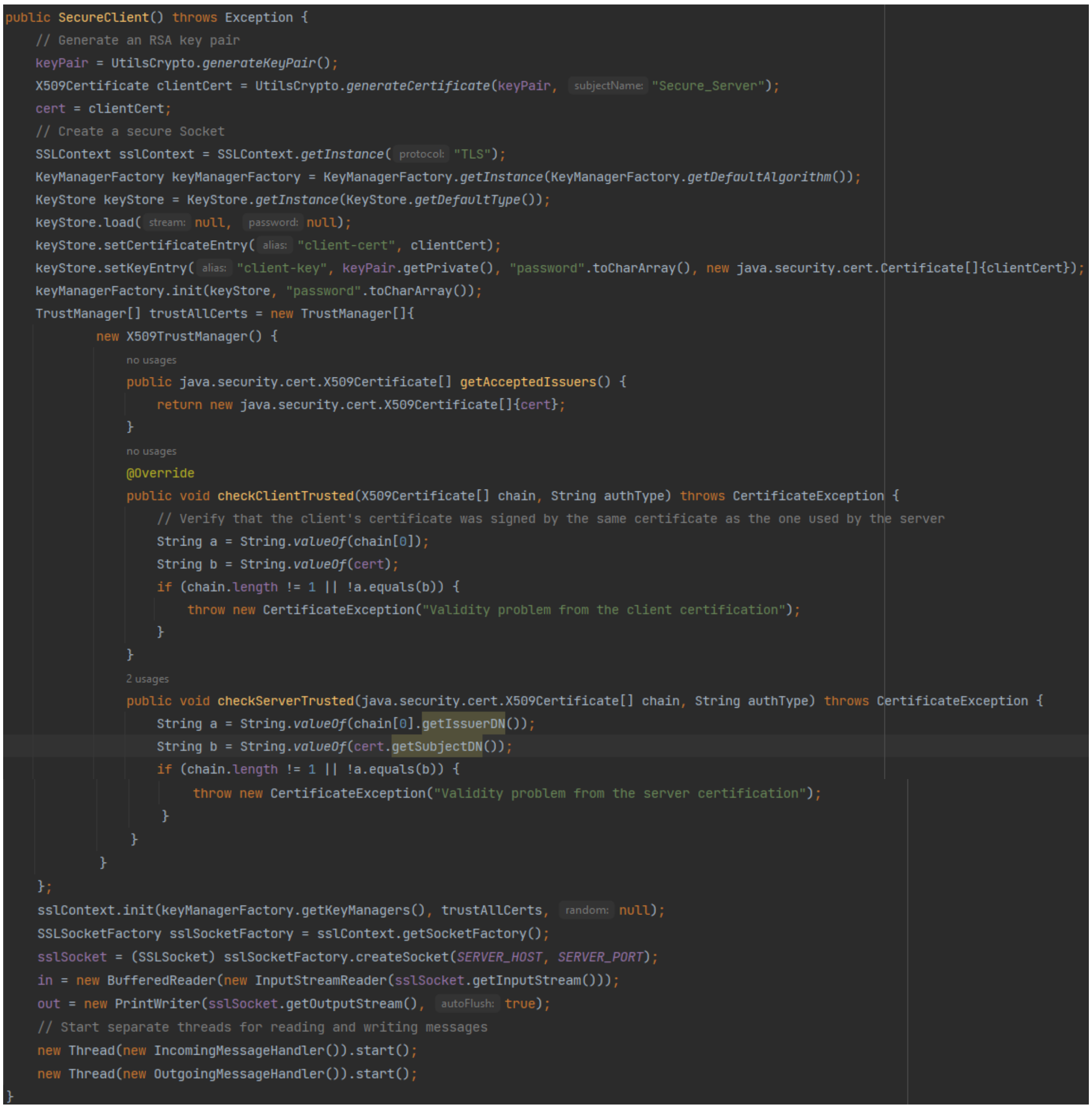


```java
public void run() {
    try {
        // Read and send messages continuously
        BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
        while (true) {
            try {
                String encryptedText = in.readLine();
                if (encryptedText == null) break;
                String decryptedText = UtilsCrypto.decrypt(keyPair.getPrivate(), encryptedText);
                String signature = in.readLine();
                assert decryptedText != null;
                boolean verified = UtilsCrypto.verify(client.getSession().getPeerCertificates()[0].getPublicKey(), decryptedText, signature);
                System.out.println("Received message: " + decryptedText);
                System.out.println("Signature Verified: " + verified);
                // Send the decrypted message to all connected clients
                for (SSLSocket connectedClient : clients) {
                    if (verified) {
                        PrintWriter out = new PrintWriter(connectedClient.getOutputStream(), autoFlush: true);
                        // Check if the client is in the secured group
                        boolean isSecuredClient = securedClients.contains(connectedClient);
                        decryptedText = client.getInetAddress().getHostAddress()+":"+client.getPort() +" sent : "+decryptedText;
                        if (isSecuredClient) {
                            // Encrypt the message with the client's public key
                            X509Certificate clientCert = (X509Certificate) connectedClient.getSession().getPeerCertificates()[0];
                            String encryptedMessageText = UtilsCrypto.encrypt(clientCert, decryptedText);
                            out.println(encryptedMessageText);
                            String signature2 = UtilsCrypto.sign(keyPair.getPrivate(), decryptedText);
                            out.println(signature2);
                        } else {
                            out.println(encryptedText);
                            String signature2 = UtilsCrypto.sign(keyPair.getPrivate(), encryptedText);
                            out.println(signature2);
                        }
                        String signature2 = UtilsCrypto.sign(keyPair.getPrivate(), decryptedText);
                        out.println(signature2);
                    } else {
                        System.out.println("Signature not verified");
                    }
                }
            } catch (SocketException e) {
                System.out.println("Client disconnected: " + client.getInetAddress().getHostAddress() + ":" + client.getPort());
                break;
            } catch (Exception e) {
                e.printStackTrace();
                break;
            }
        }
        try {
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        clients.remove(client);
        securedClients.remove(client);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            client.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. Explanation of the SecureClient Class:

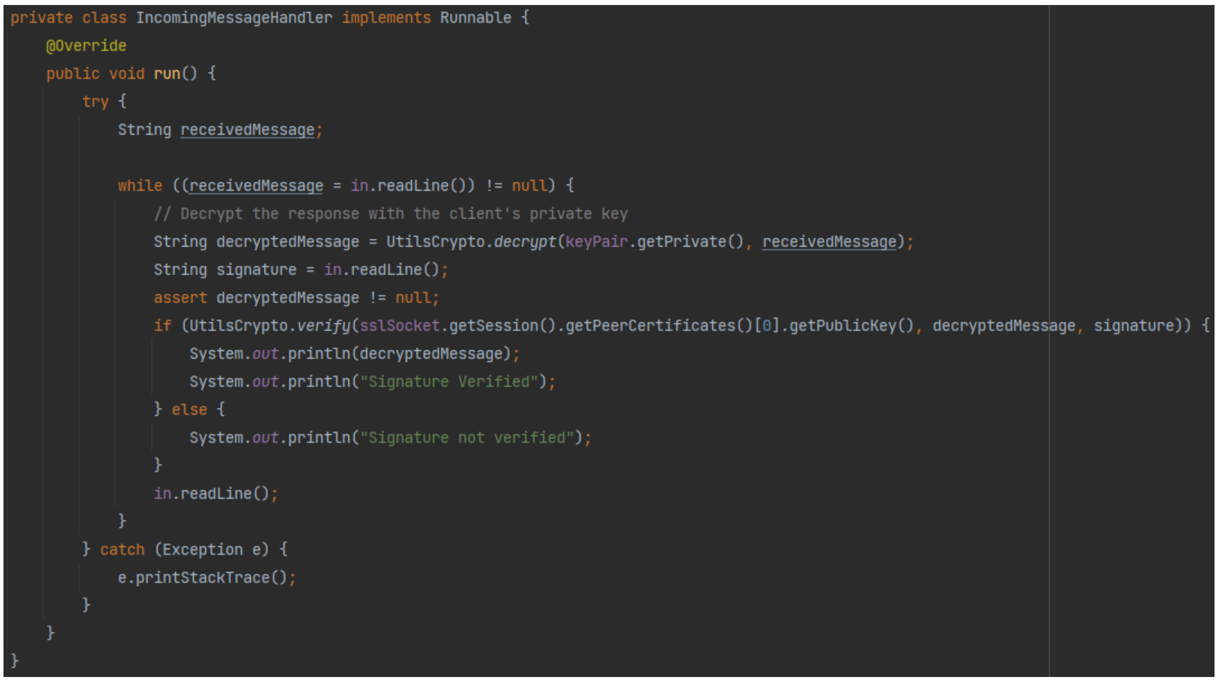
SecureClient Class: This class represents a secure client that uses SSL/TLS for secure communication. When a user sends a message, the client encrypts the message with the recipient's public key and sends the encrypted message to the server. The client also manages user keys and certificates. When a user creates an account, the client generates a key pair and a self-signed certificate. The client sends the certificate to the server, which stores it in a certificate database. When a user wants to send a message to another user, the client retrieves the recipient's certificate from the server and uses it to encrypt the message. It maintains a SSLSocket for communication with the server, and BufferedReader and PrintWriter for reading and writing messages.

SecureClient Constructor: The constructor initializes the client. It generates an RSA key pair and a self-signed certificate for the client. It then sets up a SSLSocket to connect to the server. The constructor also starts two new threads: one for reading messages from the server (IncomingMessageHandler) and one for sending messages to the server (OutgoingMessageHandler)



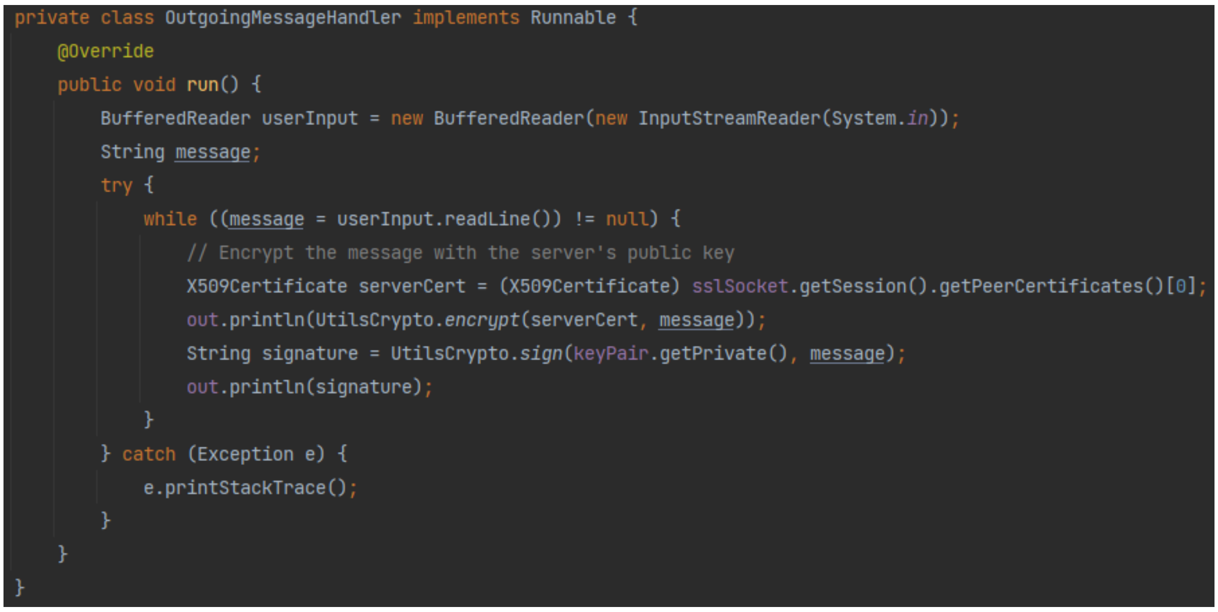
```java
public SecureClient() throws Exception {
    // Generate an RSA key pair
    keyPair = UtilsCrypto.generateKeyPair();
    X509Certificate clientCert = UtilsCrypto.generateCertificate(keyPair,  subjectName: "Secure_Server");
    cert = clientCert;
    // Create a secure Socket
    SSLContext sslContext = SSLContext.getInstance( protocol: "TLS");
    KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
    KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
    keyStore.load( stream: null,  password: null);
    keyStore.setCertificateEntry( alias: "client-cert", clientCert);
    keyStore.setKeyEntry( alias: "client-key", keyPair.getPrivate(), "password".toCharArray(), new java.security.cert.Certificate[]{clientCert});
    keyManagerFactory.init(keyStore, "password".toCharArray());
    TrustManager[] trustAllCerts = new TrustManager[]{
            new X509TrustManager() {
                no usages
                public java.security.cert.X509Certificate[] getAcceptedIssuers() {
                    return new java.security.cert.X509Certificate[]{cert};
                }
                no usages
                @Override
                public void checkClientTrusted(X509Certificate[] chain, String authType) throws CertificateException {
                    // Verify that the client's certificate was signed by the same certificate as the one used by the server
                    String a = String.valueOf(chain[0]);
                    String b = String.valueOf(cert);
                    if (chain.length != 1 || !a.equals(b)) {
                        throw new CertificateException("Validity problem from the client certification");
                    }
                }
                2 usages
                public void checkServerTrusted(java.security.cert.X509Certificate[] chain, String authType) throws CertificateException {
                    String a = String.valueOf(chain[0].getIssuerDN());
                    String b = String.valueOf(cert.getSubjectDN());
                    if (chain.length != 1 || !a.equals(b)) {
                        throw new CertificateException("Validity problem from the server certification");
                    }
                }
            }
    };
    sslContext.init(keyManagerFactory.getKeyManagers(), trustAllCerts,  random: null);
    SSLSocketFactory sslSocketFactory = sslContext.getSocketFactory();
    sslSocket = (SSLSocket) sslSocketFactory.createSocket(SERVER_HOST, SERVER_PORT);
    in = new BufferedReader(new InputStreamReader(sslSocket.getInputStream()));
    out = new PrintWriter(sslSocket.getOutputStream(),  autoFlush: true);
    // Start separate threads for reading and writing messages
    new Thread(new IncomingMessageHandler()).start();
    new Thread(new OutgoingMessageHandler()).start();
}
```

IncomingMessageHandler.run(): The IncomingMessageHandler Class represents a handler for incoming messages from the server. This method is called when the IncomingMessageHandler thread is started. It reads messages from the server, decrypts them with the client's private key, verifies their signatures, and then prints them to the console.



```java
private class IncomingMessageHandler implements Runnable {
    @Override
    public void run() {
        try {
            String receivedMessage;

            while ((receivedMessage = in.readLine()) != null) {
                // Decrypt the response with the client's private key
                String decryptedMessage = UtilsCrypto.decrypt(keyPair.getPrivate(), receivedMessage);
                String signature = in.readLine();
                assert decryptedMessage != null;
                if (UtilsCrypto.verify(sslSocket.getSession().getPeerCertificates()[0].getPublicKey(), decryptedMessage, signature)) {
                    System.out.println(decryptedMessage);
                    System.out.println("Signature Verified");
                } else {
                    System.out.println("Signature not verified");
                }
                in.readLine();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

OutgoingMessageHandler.run(): This method is called when the OutgoingMessageHandler thread is started. It reads messages from the console, encrypts them with the server's public key, signs them with the client's private key, and then sends them to the server.



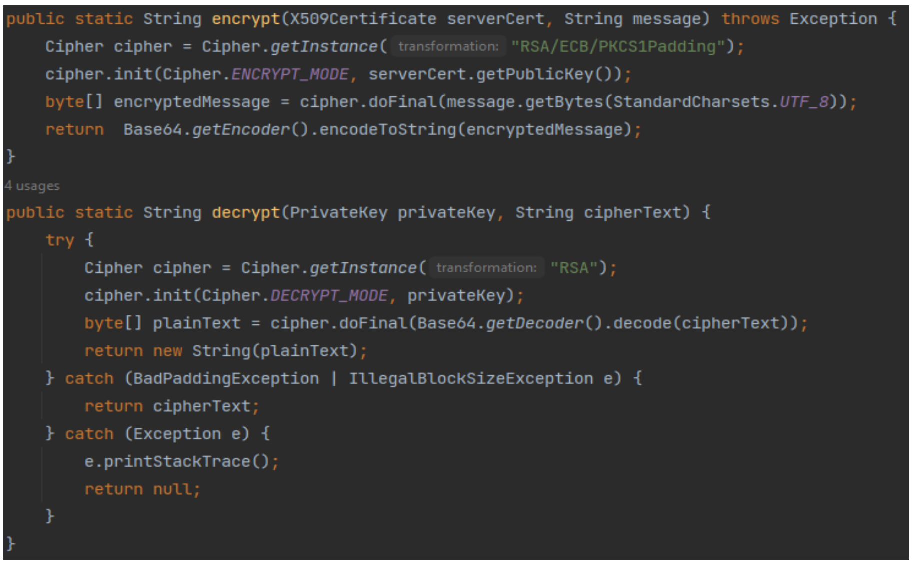
```java
private class OutgoingMessageHandler implements Runnable {
    @Override
    public void run() {
        BufferedReader userInput = new BufferedReader(new InputStreamReader(System.in));
        String message;
        try {
            while ((message = userInput.readLine()) != null) {
                // Encrypt the message with the server's public key
                X509Certificate serverCert = (X509Certificate) sslSocket.getSession().getPeerCertificates()[0];
                out.println(UtilsCrypto.encrypt(serverCert, message));
                String signature = UtilsCrypto.sign(keyPair.getPrivate(), message);
                out.println(signature);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3. Description of the Cryptographic Utilities:

UtilsCrypto class provides various cryptographic utility methods, such as encryption, decryption, signing, and verification. It also provides methods for generating RSA key pairs and self-signed X.509 certificates.

A. Encryption

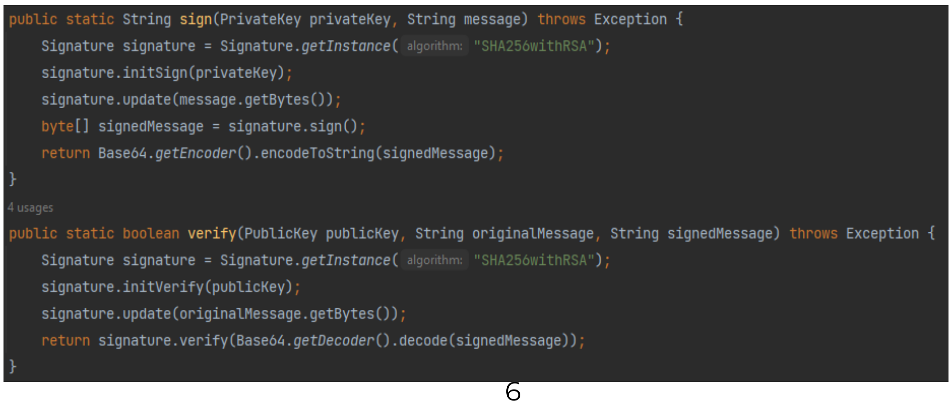
encrypt(X509Certificate serverCert, String message) & decrypt(PrivateKey privateKey, String cipherText) : The first method encrypts a message using the public key of a given X.509 certificate. It uses the "RSA/ECB/PKCS1Padding" cipher and returns the encrypted message as a Base64-encoded string. The other decrypts this Base64-encoded cipher text using a given private key. It also uses the "RSA" cipher. If the cipher text is not intended for the given private key, it returns the cipher text as is. If any other exception occurs, it prints the stack trace and returns null.



```
public static String encrypt(X509Certificate serverCert, String message) throws Exception {
    Cipher cipher = Cipher.getInstance( transformation: "RSA/ECB/PKCS1Padding");
    cipher.init(Cipher.ENCRYPT_MODE, serverCert.getPublicKey());
    byte[] encryptedMessage = cipher.doFinal(message.getBytes(StandardCharsets.UTF_8));
    return  Base64.getEncoder().encodeToString(encryptedMessage);
}
4 usages
public static String decrypt(PrivateKey privateKey, String cipherText) {
    try {
        Cipher cipher = Cipher.getInstance( transformation: "RSA");
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        byte[] plainText = cipher.doFinal(Base64.getDecoder().decode(cipherText));
        return new String(plainText);
    } catch (BadPaddingException | IllegalBlockSizeException e) {
        return cipherText;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

B. Signature

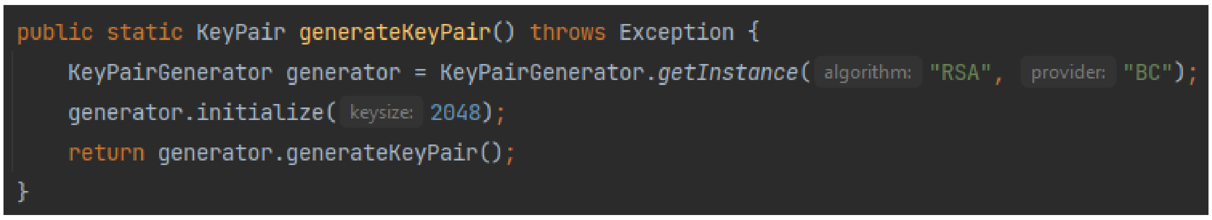
sign(PrivateKey privateKey, String message) & verify(PublicKey publicKey, String originalMessage, String signedMessage: The first method signs a message using a given private key. It uses the "SHA256withRSA" signature algorithm and returns the signed message as a Base64-encoded string. The second one verifies a signed message using a given public key. It uses the "SHA256withRSA" signature algorithm and returns true if the signature is valid, and false otherwise. SHA256 is used to ensure data integrity. It works by taking an input (a message or a file) and producing a fixed-size output (a hash value or a message digest). The output is unique to the input and even a small change in the input will result in a drastic change in the output.



```
public static String sign(PrivateKey privateKey, String message) throws Exception {
    Signature signature = Signature.getInstance( algorithm: "SHA256withRSA");
    signature.initSign(privateKey);
    signature.update(message.getBytes());
    byte[] signedMessage = signature.sign();
    return Base64.getEncoder().encodeToString(signedMessage);
}
4 usages
public static boolean verify(PublicKey publicKey, String originalMessage, String signedMessage) throws Exception {
    Signature signature = Signature.getInstance( algorithm: "SHA256withRSA");
    signature.initVerify(publicKey);
    signature.update(originalMessage.getBytes());
    return signature.verify(Base64.getDecoder().decode(signedMessage));
}
```

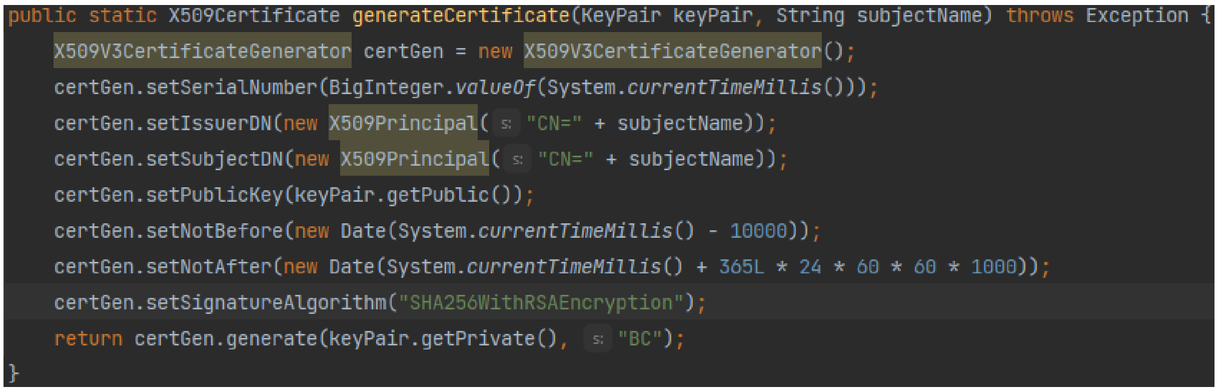
C. Keys generation

generateKeyPair(): This method generates a new RSA key pair. It uses the "RSA" key pair generator with a key size of 2048 bits and the BouncyCastle provider.



```java
public static KeyPair generateKeyPair() throws Exception {
    KeyPairGenerator generator = KeyPairGenerator.getInstance( algorithm: "RSA",  provider: "BC");
    generator.initialize( keysize: 2048);
    return generator.generateKeyPair();
}
```

generateCertificate(KeyPair keyPair, String subjectName): This method generates a new self-signed X.509 certificate using a given key pair and subject name. It uses the "SHA256WithRSAEncryption" signature algorithm and sets the certificate's serial number, issuer, subject, public key, and validity period. The certificate is generated using the BouncyCastle provider.



```java
public static X509Certificate generateCertificate(KeyPair keyPair, String subjectName) throws Exception {
    X509V3CertificateGenerator certGen = new X509V3CertificateGenerator();
    certGen.setSerialNumber(BigInteger.valueOf(System.currentTimeMillis()));
    certGen.setIssuerDN(new X509Principal( s: "CN=" + subjectName));
    certGen.setSubjectDN(new X509Principal( s: "CN=" + subjectName));
    certGen.setPublicKey(keyPair.getPublic());
    certGen.setNotBefore(new Date(System.currentTimeMillis() - 10000));
    certGen.setNotAfter(new Date(System.currentTimeMillis() + 365L * 24 * 60 * 60 * 1000));
    certGen.setSignatureAlgorithm("SHA256WithRSAEncryption");
    return certGen.generate(keyPair.getPrivate(),  s: "BC");
}
```

4. Recommendations for Further Improvements or Enhancements:

In conclusion, the key management system employed in the application uses public-key certificates to enable secure communication between clients and the server. However, there are several potential improvements that could be made to the application. The first one would be to decentralize the server because our solution work well but the server have all the information. One improvement would be to use a more robust certificate management system, such as a certificate authority (CA), to issue and manage client and server certificates. This would provide an additional layer of security and trust in the system. Another improvement would be to implement forward secrecy, which would provide additional security by generating a new session key for each communication session or when a user leaves or connects. This would ensure that even if a session key is compromised, it cannot be used to decrypt previous or future communication. The application could also be adapted to support other types of secure communications, such as secure email and secure file transfer. Additionally, keeping the certificates in cache and the keys in hard files could further improve the security of the application. These improvements would enhance the overall security and functionality of the application.