IMPLEMENTATION

Security implementation

Phase 1:Creating of the Message Hash

The Message was initially hashed using the SHA-256 cryptographic hash function that was provided by the **Java.Security.MessageDigest** class.A **getHash(String message)** method was created that takes in the message to be sent and hashes it by indicating to the **MessageDigest** object which cryptographic function to use to which in this case SHA-256 and produces a byte array.

Phase 2: Signing of the Hashed bytes

In this stage the hashed message was signed with the private key of the client.In our implementation **RSAUtils** class was created that contains the method **encrypt(byte[] data,Key key)** that takes in the data to be signed/encrypted in this case and the private key of the client since we are signing.The encrypt method uses a **Javax.Crypto.Cipher** class that is provided which encrypts by passing the algorithm used for encryption in this case its RSA algorithm with Cipher block chaining with PKCS1Padding

Phase 3 : Concatenating

In this phase we concatenate the signed message digest with the message to be sent(both in bytes).This was done using the **CompressionUtils.concat(byte[] A,byte[] B)** routine that concatenates in the following manner

- Create a byte[] C with a length of the sum of lengths of both A and B plus 4 extra bytes.
- Insert the length of the array A into C first, then contents of both byte[] A and byte[] B.The length is inserted first so as to know how long the first array extends
  So as to easily separate the new array back to its original byte arrays(A&B)

Phase 4:Compressing

In this phase we compress the concatenated byte[] obtained above using **compress(byte[] data)** routine created in the CompressionUtils Class. This method makes use of the **Java.Util.Zip.deflater** class to compress the byte[] and returns a compressed byte[].

Phase 5:Shared Key Encryption

In this phase we encrypt the compress byte[] using shared key created using the **encrypt(byte[] data)** routine that is in the **AESUtils class** which encrypt data using the shared key using the AES algorithm.In this phase also we encrypt the shared key using the **encrypt(byte[] data,Key key)** method in **RSAUtils** as explained in phase 2 but in this case it took a shared key as the data to be encrypted and the public key of the server as the key.

Phase 6:Concatenating Encrypted Shared Key with Encrypted compressed Bytes

In this phase we use the same concatenation routine used in phase 3 which creates a new concatenated byte[] that is to be sent across to the server.

Decryption implemntation

Phase 1 : Deconcatenate received byte[]

In this phase we deconcatenate the received byte[] in the following manner

- Get the first 4 bytes which hold the value of the length of the first array
- Separate the two arrays, since from the length obtained above we know how long the first array extends and thus the two arrays can be separated easily

After that procedure we obtain the Encrypted shared Key and Encrypted compressed byte[]

Phase 2 : Decryption of Shared Key & Compressed byte[]

The shared key is firstly decrypted using **decrypt(byte[] data.Key key)** in RSAUtils that uses the Java.Crypto.Cipher class that operated in decrypt mode with the private key of the server.

The Encrypted compressed byte[] can now be decrypted using the decrypt method in AES utils that would decrypt the encrypted compressed byte[].

Phase 3 : Decompressing & Deconcantenating

In this phase the compressed byte[] is decompressed by the use **decompress(byte[] data)** in **CompressionUtils** class.This method makes use of the **Java.Zip.Inflater** class and in the end it returns a decompressed byte[]. Then byte[] was deconcatenated using the procedure explained in phase 1(of decryption) and the resulting byte[] are the message byte[] and encrypted message digest byte[].

Phase 4 : Integrity checking

In this phase we check if the message was not tampered with and this is done in the following manner

- Decrypting the encrypted message digest byte[] using the **RSAUtils.decrpyt** method using the client public key
- Comparing the hash of the message byte[] with the message digest byte[]
- If equal then the message byte[] is converted into a string to be displayed to the user
- Otherwise an warning message will be printed indicating to the user that the message was modified in transit

The rest of the application

The application also consists of the server a client are can be run independently from each other. Both the client and server were written in Java and have a graphical user interface to ensure that the application is user-friendly. Below we discuss the implementation of the client and server in more detail.

Client implementation

The client implementation consists of 2 classes, the graphic user interface class called the(ClientGUI) class and a Client class with client utilities. The client GUI class consists of the methods for building the graphics interface and the variables for doing so. It also consists of listeners to buttons on the GUI to respond correctly.

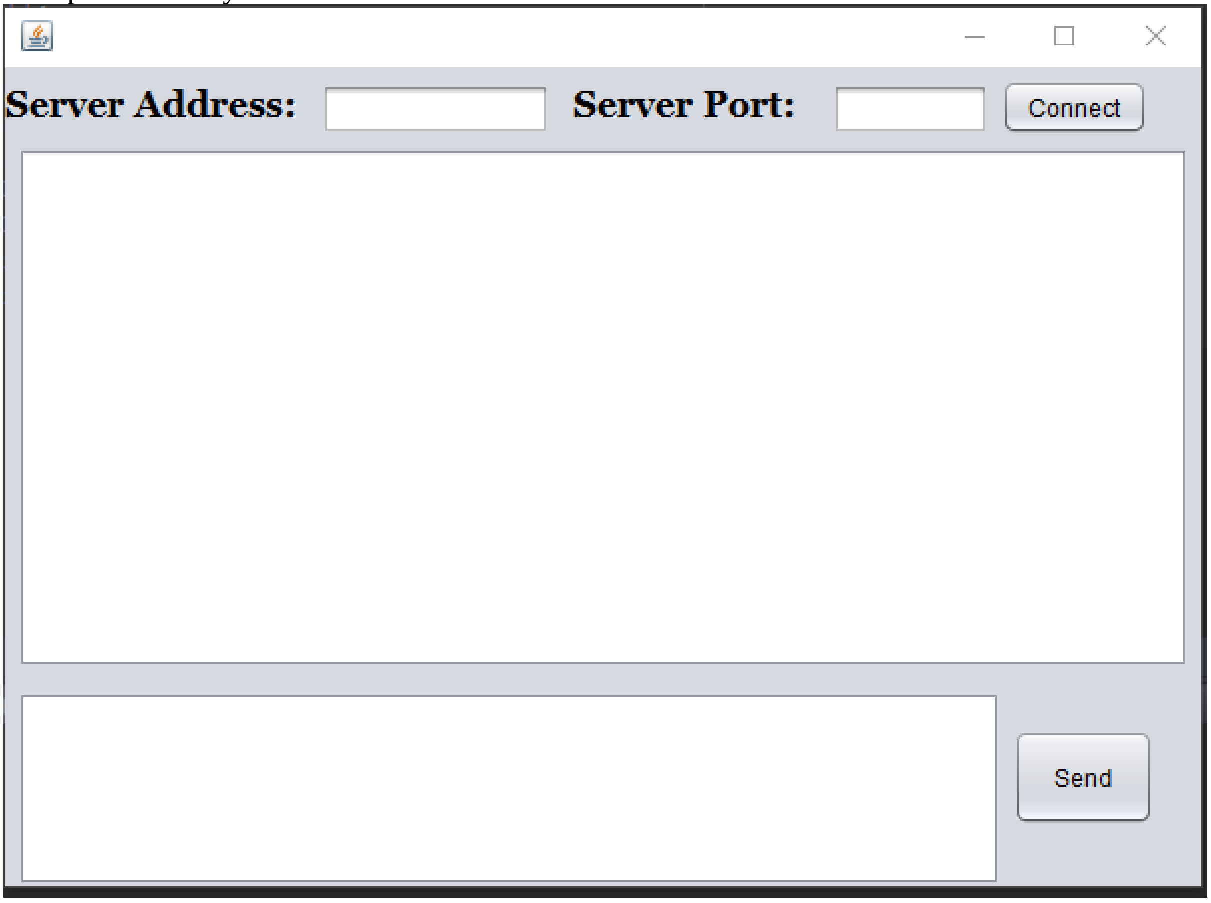
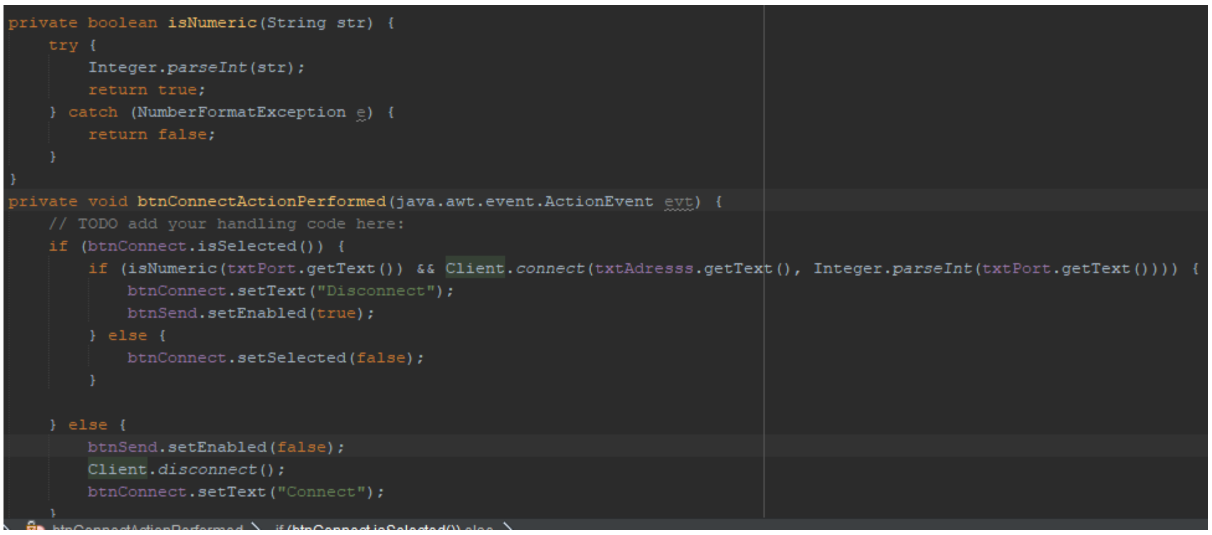


Figure 1 showing how the client interface looks when launched

Therefore, the client has two important methods which are both listeners. First we have the send button listener which listens to see if we have the send message button is pressed. If pressed, we then take the text in the text area next to the send button and we send that to the server and then we append this message to the text area above so the user can have a history of the send images. The sending of message button is disabled when we are not connected to the server and is then only enabled. If there is no data in the text area nothing is not and this avoid sending empty messages to the server. The method for sending messages is included in the Client class. The message will however first be encrypted using the methods in the PGP class.

We also have a listener to the connect which when pressed will try connecting to a server. It uses the code from the provided in the client class to connect to the server. The connect method returns a Boolean value which signals if the connection was successful or not. However before trying to connect we first check if the port number is a digit that can be evaluated to a Integer as below:



```java
private boolean isNumeric(String str) {
    try {
        Integer.parseInt(str);
        return true;
    } catch (NumberFormatException e) {
        return false;
    }
}
private void btnConnectActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    if (btnConnect.isSelected()) {
        if (isNumeric(txtPort.getText()) && Client.connect(txtAdresss.getText(), Integer.parseInt(txtPort.getText()))) {
            btnConnect.setText("Disconnect");
            btnSend.setEnabled(true);
        } else {
            btnConnect.setSelected(false);
        }

    } else {
        btnSend.setEnabled(false);
        Client.disconnect();
        btnConnect.setText("Connect");
    }
```

Figure 2 showing the code that responds to pressing of the connect button

Server Implementation

The server is implemented to be dull, in the sense that it only is capable of accepting connection, getting messages and decrypting them before displaying them in the text area. The implementation of the server is also separated into the Server clast that accept connection from users and another ServerGUI class that contains the view of the ser GUI.

Choice of Language

Java was used in this case as it has a rich library of security routines that can be used in the implementation of PGP.It was easy as well to implement the Graphical user interface using the Java Swing library that could be used in displaying of the sending and receiving of messages of the client and server respectively

CHOICE OF CRYPTO ALGORITHMS

Two forms of encryption were used in this project which are public and private encryption. There are a number of algorithms that one could use for public encryption which include RSA, Elliptic Curve Cryptography, El Gamal, Digital Signature Algorithm(DSA) .

In this project the algorithm chosen for public key encryption was RSA. The are many reasons why we chose RSA and I will expand on these reasons below. A user of RSA creates and then publishes a public key based on two large prime numbers, together with an additional value. RSA can be used to sign and to encrypt data at the same time. In this project we use RSA for signing data only. This means that we do not encrypt data directly but we encrypt the hash of the message using RSA. The reason we use RSA for signing and not for encrypting data is because RSA is a very slow algorithm.

When signing data with RSA, the private key of the sender(the client in this case) is used to encrypt the hash of the data to be transmitted. In RSA a user can specify the size of the key to use for encryption. For basic applications it is recommended that 1024 bits be used. The fact that one can increase the number of bits to use for their keys makes RSA to be very secure compared to its counterpart algorithms. Some of the key lengths that one could use are 2048, 4096 and 8192. However larger keys tend to compromise the decryption speed.

Private key encryption also has a number of encryption algorithms we could have chosen from. Some of them are Twofish, Serpent, AES, Blowfish, DES, 3DES to mention a few. In this project we decided to use AES(Advanced Encryption Standard) which is used vastly on the internet. AES is a block cipher with a block length of 128 bits.

The reason AES is used for symmetric encryption is because it is faster compared to algorithms used in asymmetric. This enables us to use AES to encrypt messages directly before sending to the message.

When compared to its counterparts like triple DES, AES is stronger and faster. The number of bit keys that one can specify on AES are 128/192/256. AES has different rounds for processing different key lengths e.g 10 rounds for processing 128 bits and 12 rounds for 129 bits.

All this makes AES secure and faster, hence suitable for symmetric key encryption.

KEY MANAGEMENT

In real world projects, the Diffie Hellman algorithm is used for key distribution. In this project we assumed that the client and the keys already have respective keys used in public key encryption.

All public keys are stored on the disk in a file as base64 strings. The program reads and parses the keys from the file to use for the respective encryption algorithms.

To avoid using stale keys, a new secret key is generated every time a message is sent to the server. As a result secret keys are not stored on the disc as public keys are.

COMMUNICATION CONNECTIVITY MODEL

For connection in this project we used the TCP connection service to ensure that the client and the server are connected before sending the data from the client to the server. This ensures that messages sent from the client will be one way or the other received by the server as TCP guarantees us that.