
University of Cape Town

CSC4026Z

Networks and Internetwork Security

# Pretty Good Privacy Chat System

*Authors:*
Mukundi Chitamba
Mujaahid Mullagee
Jonathan Swanepoel
Suvanth Ramruthen

*Student Numbers:*
CHTTIN007
MLLMUJ001
SWNJON004
RMRSUV002

June 5, 2022

## 1 INTRODUCTION

We have been tasked to utilize cryptographic techniques to exchange encrypted messages in a group chat. We should follow the Pretty Good Privacy (PGP) cryptosystem that combines public-key encryption, shared key encryption, and certificates. The key roles of such a system should include message integrity, authenticity, and confidentiality. Upon successful completion, the system should achieve end-to-end encryption between clients in a group chat. A high-level overview of our implementation can be noted in the following classes:

**TCP Group Chat:** *Server, Client, ClientInfo* and *ClientHandler* classes.
**Public and Shared Key Encryption:** *RSA, Certificates* and *Encryption* classes.
**Pretty Good Privacy:** *PGPmessages* and *Compression* classes.

## 2 COMMUNICATION PROCESS

### 2.1 Server Class

The backbone of our program is the *server* class, as it should be in any client-server application. The command to run the *server* class, and the respective *client* classes, can be found within the *readme.txt* file. The *server* class can be created by entering the following command into the terminal:

```
Linux: java -cp .:1.jar:2.jar:3.jar:4.jar:5.jar:6.jar Server
Windows: java -cp .;1.jar;2.jar;3.jar;4.jar;5.jar;6.jar Server
```

This creates a new server at port 100, which also represents the certificate authority (CA) for where our clients will validate keys. The server will wait for at least 3 clients to request to join at the *socket*, where it will spawn the following thread to handle these clients.

```
ClientHandler clientHandler = new ClientHandler(socket);
Thread thread = new Thread(clientHandler);
```

### 2.2 Client Class

The *client* class can be spawned by entering the following command into the terminal:

```
Linux: java -cp .:1.jar:2.jar:3.jar:4.jar:5.jar:6.jar Client
Windows: java -cp .;1.jar;2.jar;3.jar;4.jar;5.jar;6.jar Client
```

The user will be spurred with entering their unique username - this, along with their public key and certificate, are stored within the *client* object. The user is thereafter able to communicate with their friends in the group chat in the terminal interface. The *client* will make use of the *sendMessage()*, *listenForMessage()*, and *handleMessage()* functions to deal with the encryption and decryption of sending and recieving messages within the group chat. *ObjectOutputStream* and *ObjectInputStream* are use for these sending and receiving of messages

***sendMessage():*** To achieve end-to-end encryption, a client-fan-out approach was used to ensure the server never sees the plaintext from a client. This function loops through the *ArrayList<ClientInfo>* clients, which is initiliazed in the *clientHandler* class, and follows the PGP message sending protocol if the *client* has a public key that is trusted by the CA. After initializing a private *ObjectOutputStream* objOutput, the following functions are invoked on the PGP formatted *message* variable.

```
objOutput.writeObject(message);
objOutput.flush();
```

***listenForMessage():*** This function starts a *Thread* which allows it to continue to listen for a message on a seperate thread. After initializing a private *ObjectInputStream* objInput, the following functions are invoked:

```
Object in = objInput.readObject();
handleMessage(in);
```

***handleMessage():*** As eluded to in the *listenForMessage()* function, a *client* invokes this function when it receives a message. There is four switch cases of receivable objects: *String* type, *Public Key* type, *ArrayList* type, or a *PGPmessages* type. If a *String* is recieved, we can be assured it is a message from the group chat in plaintext that doesn't need encryption and will be displayed to the *user*. If a *Public Key* is recieved, we can be assured it is the CA's public key. If it is an *ArrayList*, we can start verifying and trusting the other *clients*. If it is a *PGPmessages* type, we can begin the process of decrypting messages to ensure authenticity and integrity.

## 3 KEY EXCHANGE PROCESS

On server launch, the server generates a public and private key which are used to sign certificates.

```
keyPair = new RSA();
CACertificate = Certificates.generateCACertificate(keyPair);
```

On client join, a public and private key is created using *RSA*. The public key is sent to the server, along with the client's request. This information is stored within the server's *ArrayList*. The *ArrayList* has a list of *ClientInfo* objects which stores the client_username, public_key and certificate variables. After a client has requested to join the server, and sent through their keys, the server returns the list of all the connected clients along with the server's (Certificate Authority's) public_key.

When the client receives the list of connected clients, it will check if the certificates of those clients have been signed with the Certificate Authority's private key. If the certificate is verified, the client will accept the public key and be able to send and receive messages to the other client associated with the key. It will otherwise discard the unverified public key.

```
public static ArrayList<ClientInfo> clients = new ArrayList<ClientInfo>();
if (Certificates.validateCertificate(client.getCertificate(), CAPublicKey))
{
    client.trustPublicKey();
    System.out.println(client.getUsername()+"\'s Public Key Verified");
}
else
{
    System.out.println(client.getUsername()+"\'s Public Key is not valid");
}
```

## 4 MESSAGE INTEGRITY

The client sending a message calculates a hash of it using the SHA-256 algorithm. The hash of the message is sent along with the actual message to the receiver. On the receiver side, the message is decrypted and a new hash of the message is calculated. The sender's hash is then compared with the receiver's hash. The comparison is used to determine if the message was tampered with along the way. Thus, message integrity is ensured by comparing the hash of the message from sender and receiver clients.

```
Signature authCheck = Signature.getInstance("SHA256withRSA");
if (authCheck.verify(signedHash))
{
    // Return message if matched signatures...
}
```

## 5 MESSAGE AUTHENTICATION

Authenticity is achieved in our program by generating a signed SHA256withRSA hash of the *byte[] messageData* with the sender's Private Key. This can then be decrypted by the receiver, with the sender's respective Public Key. If the compared hashes and thereafter content is identical, the receiver can be assured the message is authenticate.

```
byte[] shaSignature = Encryption.signedData(messageData, senderPrivate);
```

## 6 MESSAGE CONFIDENTIALITY

All messages will be signed using the receiver(‘s) public key, which is only able to be decrypted by their respective private key pair. This ensures complete confidentiality within the sending and receiving of messages, and the receiver knows this message was intended for them and only they can read the contents. The use of a shared session key encrypts the message contents within a communication channel between the users within the group chat.

```
SecretKey secretKey = Encryption.secretKeyGeneration();
byte [] encryptedMessage =
Encryption.symmetricEncrypt(secretKey, compressedData, ivParameterSpec);
```

## 7 COMPRESSION

The Java *ZIP* library was utilised to facilitate the compression and subsequent decompression of byte arrays which represented the concatenation of the plaintext and signed hash components being transmitted in the body of messages. The java zip library was preferred as it uses Huffman encoding as a basis which is used widely in well known compression algorithms. The algorithms used for the compression aspects of messages yielded an optimal compression ratio whilst maintaining lossless compression. The compressData method takes in a byte array and makes use of a *deflateroutputstream* to return a compressed byte array.

```
DeflaterOutputStream compresser = new DeflaterOutputStream(output);
compresser.write(data);
```

The decompress method makes use of an inflater outputstream to return the original decompressed byte array.

```
InflaterOutputStream inflater = new InflaterOutputStream(out);
inflater.write(compressedByteArr);
```

## 8 SHARED KEY USAGE

The shared key used in the application uses AES256 with Cipher Block Chaining (CBC) and ensures an encrypted fast session between the respective clients. The establishment of this is XORing an Initiation Vector (IV) with Message 1 and thereafter XORing this result with Message 2. Decrypting this will reverse the process, where the receiver uses the same IV. The shared secret key is used to facilitate secure communication between the members partaking in the groupchat. The shared key is distributed by making use of the Encryptions class *asymmetricEncryption()* method that uses the RSA encryption algorithm, using the recipient’s public key as input. This ensures message confidentiality as the principal with the linked private key may only gain access to the session key. The recipient will run the decryption algorithm with their private key as input. This will return the shared key encoded in byte form which can be reconstructed into the *SecretKey* format by making use of the *SecretKeySpec* constructor. The secret key is used to encrypt the message content being transmitted between the users present in the group chat. Shared key encryption of messages is preferred because of its performance efficiency and the inherent limitations of the RSA encryption algorithm which only allows for encryption of messages that are less than or equal to the key size.

## 9 ORDER OF PGP IMPLEMENTATION

The PGP cryptosystem functionality is implemented through the *pgpMessages.java* class which implements the Java *Serializable* interface. This interface enabled us to utilise the *objectoutput* stream which allowed reliable and efficient transmission of messages and keys. The methods of the *pgpMessages* class encapsulates the compression, encryption and hashing methods present in the *Encryption* and *Compression* java classes. The methods of the *pgpMessages* are called in the send and receive message methods of *Client.java*. This ensures end-to-end encryption such that the server would never be able to view nor intercept the plaintext.

### 9.1 Overview of class

#### Constructor

The constructor contains four instance variables namely the messageComponent, SessionKeyComponent, recipientUserName and senderUsername. The constructor is called to encapsulate the ciphertext byte array and relevant components into a *pgpMessage* object that is serialised and sent over the socket to the members of the group chat.

**Sender side of the PGP messages**

The *sendMessage()* function is a public static method that returns a *pgpMessage* object which will be transmitted over a TCP socket to the intended client. There are five parameters of this function namely the plaintext message, senderUsername, senderPrivateKey, recipientUserNames and recipientPublicKey. The *sendMessage()* function is called when a client is sending messages to the group chat. The steps in the method is as follows:

(1) The byte array encoding of the plaintext message is generated.
(2) The SHA256 hash of the plaintext byte array is computed and digitally signed with the sender's private key.
(3) The message and digitally signed hash byte encoding are written to a shared byte stream which is then compressed using the deflator method of *Compression.java*.
(4) A short-lived secret key of an AES256 instance is generated by calling the *secretKeyGeneration()* method from the *Encryption class*.
(5) The secret key utilises symmetric encryption methods to encrypt the compressed concatenated byte array of the plaintext message and digitally signed hash.
(6) The byte encoding of the secret key is generated by calling the *Secretkey.getEncoded()* method. This is then encrypted using *RSA* asymmetric encryption algorithms that use the recipient's public key as input.
(7) The secret shared key is concatenated with the compressed payload and wrapped in a *pgpMessage* object which is then transmitted to the recipient client securely over the TCP socket.

**Receiver side of the PGP messages**

The *receiveMessage()* function is a public static method that returns a String which represents the data which is derived from the decryption and decompression processes of PGP. If the transmission has been received without it being tampered with the user will be able to view the plaintext, otherwise a message indicating invalid data will be presented to the client. There are three parameters of this function, namely the pgpMessage object, recipientPrivateKey and the sendersPublicKey. The *receiveMessage()* function is called when a client receives a *pgpMessage* object from a fellow client. The steps in the method is as follows:

(1) The cipher text and encrypted session key component are separated from the *pgpMessage* object into a separate byte array.
(2) The session key is decrypted with asymmetric RSA decryption which uses the recipient's private key as input. The bytes are retrieved and the session key is reconstructed using the SecretKeySpec constructor.
(3) The ciphertext byte array is then decrypted using the AES symmetric decryption method present in the *Encryption.java* class which uses the reconstructed secret key as input.
(4) The resulting byte array of step 3 is then decompressed using the inflator algorithm of *Compression.java*.
(5) The plaintext and digitally signed hashes are then separated into their own respective byte arrays.
(6) The signature of the signed hash is compared with a computed hash of the plaintext received.
(7) If the hashes match the string representation of the plaintext byte array is generated and presented to the clients. When the data has been tampered with, clients are given a "Invalid data received" error response.

## 10 TESTING PROCEDURE

We conducted the following unit tests, for the following methods, to ensure our code base was well tested, robust and production environment ready.

### 10.1 Certificate Testing

The *Certificates* class was testing using the *test()* method and the *TestCertificates* class. A Certificate Authority (CA) root certificate is generated and signed with its own private key. Three client certificates were then generated, two clients having their certificates signed with the CA private key and one client having their certificate signed with a fake CA private key. The certificates are validated using the *validateCertificate()* method which checks if the certificate was signed by the CA. The two clients certificates were validated but the client with the fake signature is not. Thus, the certificates of the clients are correctly authenticated and the public keys can be trusted.

### 10.2 PGP Encryption and Decryption

We tested the method that encapsulates our PGP encryption process by have testing the individual elements of the encryption process. We have test methods that have an input, and compare the output with an expected value, for lossless compression, signing, concatenation, secret key generation and reconstruction from a byte array, and the overall PGP encryption and decryption. We made sure to test both wrong and right cases, to ensure that the application is indeed end to end encrypted.

### 10.3 PGP message exchange process

The testing of the PGP cryptosystem message exchange was done through the use of functions in the *pgpTests.java* class and System log messages present throughout the code. The following components of the secure message exchange was tested extensively through running the application with varied input and unit test cases:

- The interception of message contents was tested with the *receiveMessageTest()* function which returns true if the sending and received functions respond appropriately to known authentic and fake generated key pairs. This function shows that the use of unknown keys will not break the application but rather will respond with appropriate feedback to the trusted clients.
- The concatenation of the pgpMessage payload was extensively tested with the *testConcatenation()* method. Message payloads of known separate constituents were formulated and encapsulated into a shared byte array. The data was then split into separate byte arrays and compared to the original byte array components to ensure that the writing to shared byte arrays were functioning as intended.
- The secret shared key was a major concern as it is used to encrypt the plaintext and signed hash payload. The confidentiality and integrity of this payload was of utmost importance. The *secretKeyConstruction()* method tests whether the secretKey can be encoded into a byte array representation, asymmetrically encrypted, transferred over a socket without loss of data, decrypted and reconstructed into an identical secret key. This method ensures that the secret shared key maintains its secrecy and is only made known amongst trusted principals.
- The testing of the utility compression methods were done to ensure that the PGP TCP chat application was performing at maximum efficiency and the compressed transmission was lossless. The compression ratio i.e. compressed byte array length/original data byte array length was calculated to ensure that the function was performing optimally. Tests were carried out using the *compressionLosslessTest()* function as well as the System logs.

We then tested the overall app by running it with multiple terminals open, having a server and 3 clients connected. We exchanged messages to test the message transmission. We also added print statements to see the encryption process at every step, displaying the relevant keys and changes made to the message digest.

## REFERENCES

Wittmann, T. W. (2021, August 17). WittCode. WittCode.Com. Retrieved May 10, 2021, from https://wittcode.com/java/java-socket-programming-multiple-clients-chat

## Appendix A COMMUNICATION PROCESS SEQUENCE DIAGRAM

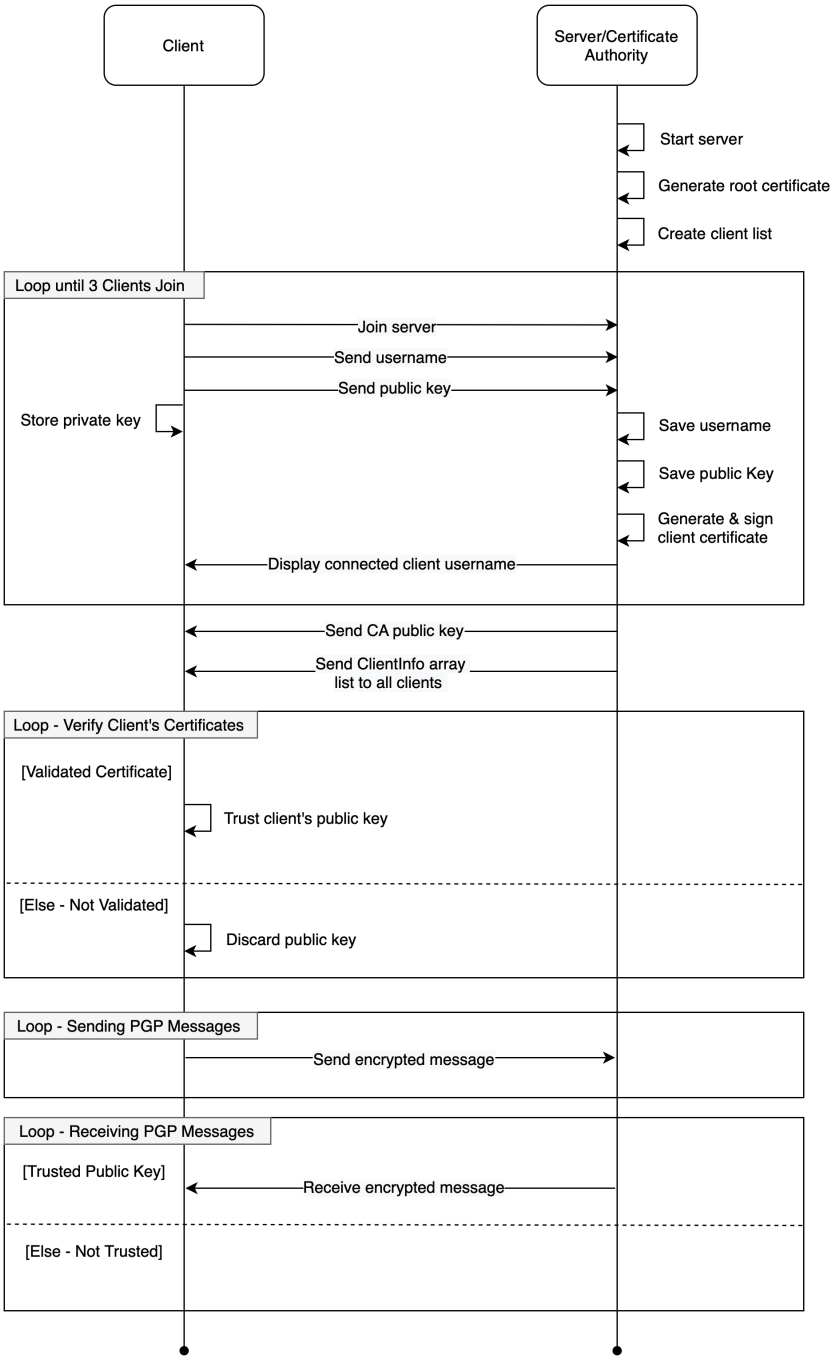


Fig. 1. Sequence Diagram of TCP Protocol

## Appendix B PGP MESSAGE EXCHANGE PROTOCOL DESIGN

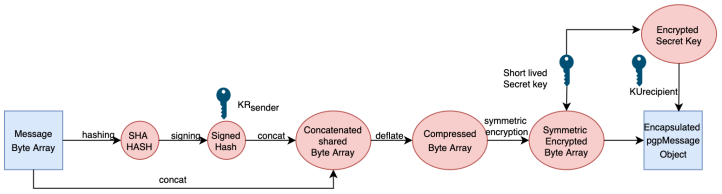


Fig. 2. PGP Message Sender Operations

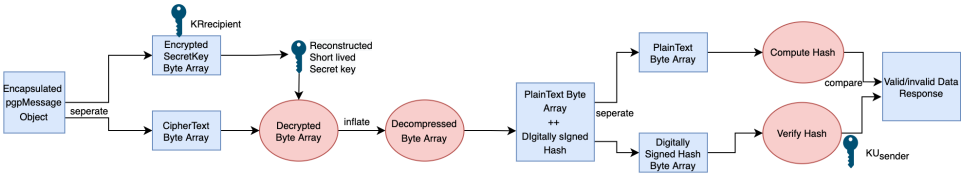


Fig. 3. PGP Message Receiver Operations