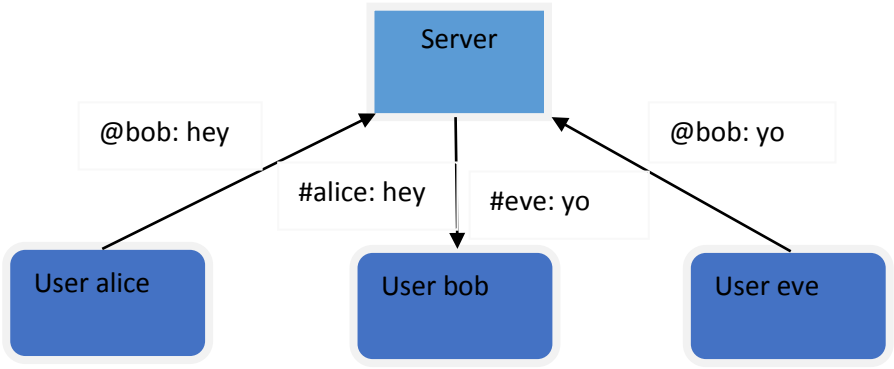
COL374/672 Computer Networks: 2019-20 semester I

Assignment 2

In this assignment, we will build a chat application that allows users to do an encrypted chat with one another, which cannot be decrypted by the chat server. The Figure below shows the architecture we will build. Users can direct messages to other users through an @ prefix, and the server needs to forward these messages to the intended recipient. The message itself would be encrypted between any pair of users so that the server cannot read the messages, the server can only infer that a communication is happening between the given pair of users. This is pretty much what Whatsapp's architecture is all about, so you are indeed designing your own Whatsapp!



Step 1

Let us start with building a protocol for non-encrypted communication. You will have to write a client application and a multi-threaded server application using TCP Sockets. The protocol format can be as follows, much like HTTP, with a \n following each command/header line and a \n\n to indicate end of the message. Rather than HTTP's stateless method though, we will preserve the state for each TCP connection.

- **User registration**

Each client application will open two TCP sockets to the server, one to send messages to other users, and one to receive messages from other users. We can also do it over a single TCP connection but it will require some concurrency management, hence we will keep it simple for now. The client application will need to start with registering the user on both the TCP sockets using the following method.

Client to server registration, for sending messages	REGISTER TOSEND [username]\n \n
Server to client, if username is well formed	REGISTERED TOSEND [username]\n

	\n
Client to server registration, for receiving messages	REGISTER TORECV [username]\n \n
Server to client, if username is well formed	REGISTERED TORECV [username]\n \n
Server to client, if username is not well formed	ERROR 100 Malformed username\n \n

Upon receiving the registration messages, the server will check whether the username is well formed, ie. only alphabets and numerals, with no spaces or special characters. It will acknowledge the registration requests if the username seems good, else it will return an error message. As will also become obvious soon, the server will need to maintain a list of users who have registered and the corresponding TCP Socket objects for the sending and receiving connections over which they registered.

The user registration should be the very first activity that is done when a client opens the sockets to the server. The server should return an error message to all other commands it receives on the socket until the user registration has been done on both the sockets.

Server to client message in response to any communication until registration is complete	ERROR 101 No user registered\n \n

- **Message sending**

On the TCP socket dedicated to sending messages to other users, the following method will be used.

Client to server message	SEND [recipient username]\n Content-length: [length]\n \n [message of length given in the header above]
Server to client message, if message is delivered	SENT [recipient username]\n \n
Server to client message, if message undelivered	ERROR 102 Unable to send\n

	\n
Server to client message, if header is incomplete	ERROR 103 Header incomplete\n \n

The format of the SEND method is similar to that of various HTTP methods, with a header section followed by the data section. The header section has a field for content length which tells the server how many bytes to read after the blank line. If the Content-length field is missing then the server should send an error message back to the client and close the socket. This is because in this case the server will not know how to parse any further data it receives on the TCP connection. The client can then reattempt to open a new connection and start do the registration again.

If the message is received correctly by the server then the server should check if the recipient has registered, and attempt to forward the message to the recipient over the socket on which the recipient had registered to receive messages from other users. This is why we mentioned earlier that the server will need to maintain a list of users who have registered, and the corresponding TCP Socket objects for the sending and receiving connections over which they registered.

The method to forward a message to its intended recipient is explained next. If the recipient is registered at the server, and it acknowledges having received the message, then the server should send a SENT success notification to the sender. Else the server should send an error message to the sender. This will of course be done on the TCP socket of the sender which is dedicated to sending messages to other users.

- **Message forwarding**

On the TCP socket dedicated to receiving messages from other users, the following method will be used.

Server to client message	FORWARD [sender username]\n Content-length: [length]\n \n [message of length given in the header above]
Client to server message, if message is received	RECEIVED [sender username]\n \n
Client to server message, if header is incomplete	ERROR 103 Header incomplete\n \n

The format of the forwarded message is similar to the originally sent message, and the parsing procedure to be followed by the server will also be similar.

- **Client application**

The client application should take a command-line input for the username and the server's IP address (localhost or 127.0.0.1 for when you are running the server locally), and then start with opening two TCP sockets to the server, one for sending messages to other users and one for receiving messages. When the sockets are opened, the client should first send REGISTER messages and read the acknowledgements.

The client will then need to start two threads. On one thread, it should read one line at a time from stdin, ie. what the user types on the keyboard. Each time this thread reads a line, it should parse it to get the intended recipient and the message. Each line typed by the user should be of the form:

@[recipient username] [message]

If the line is not in this format then the thread should reject the line and ask the user to type again.

The thread should then send a SEND message to the server and wait for a response. Upon getting a successful response, it should inform the user that the message was delivered successfully, else convey an error to the user. The application can then loop back to waiting to read the next line typed by the user.

On the other thread, it should wait to read FORWARD messages from the server. Upon receiving a message successfully, it should send the appropriate response to the server, display the message to the user, and go back to waiting to receive more messages from the server.

- **Server application**

The server application would begin with listening for connections on some port number, you can choose any port number. Upon receiving new socket acceptances from a client application, the server should spawn threads for each socket to communicate with client. The threads will begin with expecting to receive REGISTER messages.

For the thread which receives a TORECV message, the thread should add the user and corresponding Socket object to a global Hashtable (or any other suitable data structure), and close the thread, but of course not the Socket.

For the thread which receives a TOSEND message, the thread should begin to wait to receive SEND messages from the client application. It will parse these messages, and upon receiving well-formed messages the thread should send a FORWARD message to the recipient on the recipient's Socket with which it registered through the TORECV message. This Socket will need to be looked up from the Hashtable being maintained by the server. The thread will then wait for a response, which it will

convey back to the sender on the TOSEND Socket of the sender, and then begin to wait for more messages from the sender.

This should give a nice and simple chat application through which users can talk one-on-one with each other.

- **Extensions**

Now extend this with defining your own UNREGISTER message, for when users go offline.

Users may also disconnect arbitrarily by pressing Ctrl-C and not send an UNREGISTER message. How would you deal with such a scenario? You need not implement this, just describe it in detail.

Also think and describe how would you extend the client and server applications to deal with offline users? Similar to the single and double checks in Whatsapp, a sender should be able to send messages to offline recipients and get notified later whenever the messages are delivered to them. You do not have to implement this, just describe it in detail.

Step 2

Let us now think how we can make this communication encrypted so that only the intended recipient can read the message. We will use something called a public-private key based encryption for this purpose. Very simply put, this is how it works:

- Say there are two users A and B, and A wants to send a message to B
- The message is M
- Both A and B will have their own respective public and private key pairs, let's call for (K_{pub}^{A}, K_{pvt}^{A}) for A and (K_{pub}^{B}, K_{pvt}^{B}) for B
- As you would have figured, a public key is public, ie. it can be published publicly, but the private key should not be disclosed
- For A to send an encrypted message to B, it will encrypt the message on B's public key, ie. it will send M' = K_{pub}^{B}(M). B will then decrypt M' through its private key, ie. it will obtain M = K_{pvt}^{B}(M')
- As you would have figured, the public and private keys come in pairs, and there is a cool method called RSA to generate these key pairs. We will study this later in the course.

You should extend the methods you have implemented in your client and server applications to encrypt the messages being sent. Note that you only have to encrypt the message, not the headers. The server will thus still be able to know who is communicating with whom, but will not be able to see the contents of the communication. You will need to do the following:

- When a client starts, it should generate a (public, private) key pair for itself
- The public key should be published openly, and you can do this by extending the REGISTER method to have each client send its public key to the server, or you can also create your own method for a client to send its public key to the server. The server can store this public key in a global registry, just like how it would be storing the TCP Socket objects for each client
- To send messages, a client will first need to fetch the public key of the recipient. You can do this by creating a new message type for FETCHKEY
- For each message to be sent then, the client will encrypt the message on the public key of the recipient and dispatch it through the SEND method
- When this message is received by the target recipient user, it will decrypt the message using its private key

To help you with the encryption and decryption, we are providing you with a CryptographyExample.java file that uses various java.security methods to generate key pairs and encrypt/decrypt messages. The relevant methods you will need to use are:

- generateKeyPair() which generates a java.security.KeyPair object that can give the public and private keys in a byte[] array
- encrypt() which accepts a public key and data that needs to be encrypted, and returns a byte[] array of the encrypted data
- decrypt() which accepts a private key and encrypted data, and returns a byte[] array to the decrypted data
- Important: You should not send the public keys or encrypted data in a binary format in various REGISTER and SEND methods. Think why this is not desirable. You should instead send a base64 encoded message of the binary data. Read about base64 online, it is an encoding method to convert binary data into a text format, and Java provides an easy API to encode and decode:
    - String java.util.Base64.getEncoder().encodeToString(byte[] array)
    - byte[] java.util.Base64.getDecoder().decode(String str)
- Important: Please use the same public key algorithm (RSA, 512 bit key size) and key specifications (X509 and PKCS8) as given in CryptographyExample.java, so that your submissions can be evaluated against unit tests.

Step 3

Although the communication is encrypted, how can message integrity be checked to ensure that the message has not been tampered with? This is where signatures come into the picture, and they work as follows, in very simplified terms:

- A hash method is used to obtain a digest for the encrypted message, such as through MD5 or SHA1 or SHA2: H = hash(M')
- The hash is encrypted on the private key of the sender: H' = K_{pvt}^{A}(H) and sent along with the encrypted message (H', M') to the recipient
- The recipient uses the public key of the sender to check whether it has received the encrypted message correctly, ie. whether hash(M') is the same as K_{pub}^{A}(H')
- Remember to only send base64 encoded signatures

You should extend the client and server applications to use message signatures. You can include the message signature in the SEND message, and at the recipient you will need to build a method to fetch the public key of the sender to verify the signature.

Use the java.security.MessageDigest method with SHA-256 to obtain the digest:

```
MessageDigest md = MessageDigest.getInstance("SHA-256");

byte[] shaBytes = md.digest(inputData);
```

Although signatures can ensure message integrity, they can still not guarantee that indeed the message was sent by the user claiming to have sent it. This is where certificates come into the picture, and although we will not incorporate them in this assignment, you should read more about them. Very simply, a certificate is a signed public key, which has been signed by a trusted third party.

Step 4 (no action needed)

You don't have to implement this, but here is an idea of how you can go even beyond Whatsapp. Whatsapp still requires a central server and which can know who is communicating with whom. A first step can be for a sender client to fetch the IP address of the receiver client from the server, and directly open a TCP socket to send data to the receiver client. Thus, even encrypted data will not pass through the server, clients can directly communicate with each other. A server would still be required though, to maintain a registry of client usernames and IP addresses, much like registries in peer to peer networks. Even this requirement can be relaxed though, if you can set up a Gnutella like network. Whenever clients join a Gnutella network, they can share their username and public key into the network. Later in the course we will also discuss DHTs or Distributed Hash Tables, which are a beautiful way to maintain such information in a completely decentralized manner in a network.

What to submit

A single .zip or .tar.gz file contain an src directory with the source code of your assignment and a README on how to execute the server and client applications, and a doc directory containing a report. The report should specifically contain answers to all the questions asked throughout the assignment document above.

Note that we have given some code samples in Java, you can also programme in Python if you want but you will need to find the right libraries yourself.

Useful security packages you will need

https://docs.oracle.com/javase/7/docs/api/index.html?java/security/package-summary.html

https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html

https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#KeyPairGenerator