Lab 4 - Chat App

Christopher Davisson

02/01/26

Abstract

P2P message app with directory/relay server. DHE and RSA key exchange working.

Introduction

This is a P2P chat app. It uses a relay server to act as a target for two clients behind NAT. The server keeps track of the clients who connect to it and provides a list for users to choose who to chat with. This app only facilitates 2 person conversations at once. Groups are not possible yet. However more than one group of two can message concurrently.

I wanted to make this P2P but I quickly found that without port forwarding there are very few options to get out of local network. I had the chat working in local network quickly but moving to something that could work between networks took some doing. As is the server still needs to be findable. It needs port forwarding and the IP needs to be known by each client.

System Architecture

Direct P2P:

$$\text{[Client A]} \longleftrightarrow \text{TCP} \longleftrightarrow \text{[Client B]}$$

Via Relay:

$$\text{[Client A]} \longleftrightarrow \text{TCP} \longleftrightarrow \text{[Server]} \longleftrightarrow \text{TCP} \longleftrightarrow \text{[Client B]}$$

The program is comprised of two parts:

Server

This is currently a small program. It keeps track of clients who have registered, maintains connections to each of them, and forwards messages from sender to receiver. Upon registration the client can also send a LIST command to the server and get back a list of users.

The server has some bugs that I had trouble fixing, like if you tried to talk to someone in a chat already. I fixed it by just ending the first chat and making the new one. I can think of better solutions but they require a more server-like approach.

Chat Client

Component	Technology
Language	Python
GUI	tkinter
TUI	In progress

Architecture:

- **chat.py** - main body of the chat app
- **gui.py** - tkinter and helper functions
- **crypto.py** - holds the main logic for DHE and RSA key exchange

Chat.py:

I made this first to deal with local P2P connections. It works perfectly, you first set one chat to listen and another to connect to the correct port. Pure P2P.

I then expanded it to also include relay servers. It needs IP and port then it can request a list of available users to chat with. If you click on their names (all randomly generated), then it sends a CHAT_REQUEST message to the server, who then starts to act as a relay.

You can change the name, though it does break the server right now. You can also change your port, though I think it only affects local connections.

I have two crypto options: DHE and RSA. Each with the bit size options: 32, 64, 2048. 32 and 64 are both generated primes. 2048 is hardcoded from RFC 3526. I check the primes using Miller-Rabin algorithm. The cryptography all works and the shared keys and secret keys are displayed. Double clicking on the key will print it to the chat log.

Protocol Specification

Transport Layer: TCP

Application Protocol:

- Regular messages: raw UTF-8 text
- Crypto/control messages: `CRYPTO:` prefix + JSON payload

Message Types:

Type	Fields	Description
DHE_INIT	type, p, g, A	Initiator sends DH parameters and public key
DHE_REPLY	type, B	Responder sends their public key
RSA_PUBKEY	type, n, e	Initiator sends RSA public key
RSA_SECRET	type, c	Responder sends encrypted shared secret
REGISTER	type, name, port	Client registers with server
LIST	type	Request list of online users
CHAT_REQUEST	type, target	Request to chat with another user

Cryptographic Methods

Diffie-Hellman Key Exchange (DHE)

Parameters:

- For 2048-bit: p and g are hardcoded from RFC 3526 Group 14
- For 32/64-bit: p is generated using Miller-Rabin primality test, $g = 2$

Process:

1. Alice generates private key: $a = \text{random}(2, (p-1)/2)$
2. Alice computes public key: $A = g^a \mod p$
3. Alice sends p, g, A to Bob
4. Bob generates private key: $b = \text{random}(2, (p-1)/2)$
5. Bob computes public key: $B = g^b \mod p$
6. Bob sends B to Alice
7. Both compute shared secret: $s = B^a \mod p = A^b \mod p$
8. Key derivation: $\text{key} = \text{SHA-256}(s)$

RSA Key Transport

Key Generation:

1. Generate two primes p and q using Miller-Rabin (512-bit each for 1024-bit RSA)
2. Compute $n = p \times q$
3. Compute $\phi(n) = (p-1) \times (q-1)$
4. Public exponent: $e = 65537$
5. Private exponent: $d = e^{-1} \mod \phi(n)$ (using Extended Euclidean Algorithm)

Process:

1. Alice generates RSA keypair, sends public key (n, e) to Bob
2. Bob generates random secret s
3. Bob encrypts: $c = s^e \mod n$
4. Bob sends c to Alice
5. Alice decrypts: $s = c^d \mod n$

6. Key derivation: key = SHA-256(s)

Prime Generation

Primes are generated using the Miller-Rabin primality test with 40 rounds. This gives a false positive probability of less than 2^{-80}, which is negligible.

Security Considerations

This is a very insecure app. None of the messages are sent encrypted. They are all sent plaintext (per assignment requirements).

Other notes:

- The 32 and 64 bit key options are just for demo, they are not secure
- DHE provides forward secrecy (compromising long-term keys doesn't compromise past sessions)
- RSA key transport does not provide forward secrecy
- The relay server sees all traffic in plaintext
- No authentication - anyone can claim any username

References

1. RFC 3526: More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)
2. GeeksforGeeks: Miller-Rabin Primality Test / Extended Euclidean Algorithm / Modular Inverse