CS 6349 Network Security Project
Design Document

I. Team Members:

Akshitha Srinivasan (axs230012)
Karan Vora (kxv230021)

II. Required Supported Functionalities:

- The server maintains an account for every client, storing their security credentials (Public Key), which are used to authenticate them when initiating IM sessions with other clients.
- A client can request a list of available clients from the server. However, they can only initiate a communication session with a maximum of one client at any given time.
- At any given time, a client can be in one of the following two states: Busy – involved in an IM with another client. Idle – waiting for other clients to do IM with it.
- The server will assist the clients in establishing a secure communication channel by managing the distribution of session keys, which are used for securing their conversations.
- Both clients must be in Idle state before they can establish a communication session with each other.
- Once a session is established, clients communicate directly with each other without involving the server, until the session is terminated.
- A single session can only involve two clients. However, multiple sessions can occur simultaneously between different pairs of clients.

III. Assumptions

a Technical Assumptions

i JDK Version: The project assumes a specific version of the Java Development Kit (Version 12 and above) for compatibility and feature usage.

ii Application Architecture: it is assumed that there is a particular architecture (client-server).

iii It is assumed that the user has access to command line interface for execution

b It is assumed that there is no need for GUI.

c It is assumed that there are no compliance or framework requirements.

d Assume that the server and all clients are connected within the same computer network and share a common subnet

IV. Directory Structure

```
|–Final_Submission
|       |-src
|       |   |-----server
|       |   |-----common
|       |   |---client
|       |   |-----resources
|       |-Makefile
|       |-README
|       |-Final_design_template
```

The directories within src also have files of their own. However, the make is meant to be called from src, and all interactions with the command line happen from here.

1. Overview

This document describes the design and implementation of a network application that facilitates secure communication between a client and a server. The application supports three main functionalities:

1. **Authentication:** Clients authenticate themselves to the server and receive credentials to talk to a client of their choice.
2. **Chat Messaging:** Clients can exchange chat messages with each other securely.
3. **Client Control:** Clients establish connections with the server and peer clients, and gracefully exit the application, closing all open connections.

2. Architecture

2.1 Server Architecture

The server application is implemented using Java and utilizes multiple server sockets to handle different functionalities:

1. **Authentication Server Socket:** Listens for incoming connections from clients sending authentication requests.
2. **Control Server Socket:** Listens for incoming connections from clients for control purposes (client chat session creation and client exit).

2.2 Client Architecture

The client application is also implemented using Java and establishes connections with the server on different ports for various functionalities:

1. **Authentication Socket:** Connects to the authentication server socket for sending authentication requests and receiving a key for a secure session.
2. **Control Socket:** Connects to the control server socket for handling client exit.
3. **Chat Socket:** Connects to the chat socket for exchanging chat messages.

3. Components

3.1 Server Components

1. **Server:** Main class responsible for initializing and managing server sockets for authentication and session key distribution, control, and chat functionalities.
2. **AuthHandler:** Thread class handling incoming client connections for authentication.
3. **ClientHandler:** Thread class handling incoming client connections for session key distribution.
4. **ChallengeManager**: Class handling the authentication of clients using challenge-response mechanism.
5. **PeerChat**: Class facilitating the creation of peer-to-peer connection between clients for chat purposes after clients have agreed to establish a connection with each other. Peer chat helps maintain the log of connections.
6. **RSAUtils**: Class handling RSA-based functionalities used for client authentication and generation of a session key for client chat.
7. **Status**: Class defining the state of client in order to facilitate one-to-one chat connections
8. **User**: Class defining a user object which is used to facilitate peer-connections between clients by storing user details like its name and associated socket.
9. **ConnectionManager**: Class handling the intermediate flow of messages from client to client through server till clients agree to begin a chat session with each other. Sets up a PeerChat instance after forwarding the session key to both clients.

10. **UserPublicKeyManager**: Class handling management of RSA public keys, which are used by the server to verify client signed messages.

3.2 Client Components

1. **Client:** Main class responsible for connecting to the server and providing an interface for secure chat messages and controlling the client application.
2. **ClientInputHandler**: Class responsible for handling all input generated through the command line on client end.
3. **NotificationListener**: Class handling initiation and acceptance of chat requests between clients through server mediation in order to facilitate peer-connection.
4. **UserPrivateKeyManager**: Class handling management of RSA private keys, which are used by the client to sign messages for authentication.
5. **ClientMultipeerwithdynamicport**: Class that contains the implementation that actually executes peer-communication using unique sockets to facilitate multiple simultaneous one-to-one connections.

4. Communication Protocols

4.1 Server Connection Handshake

The server connection handshake ensures a structured initiation of communication between the client and server. It involves the following steps:

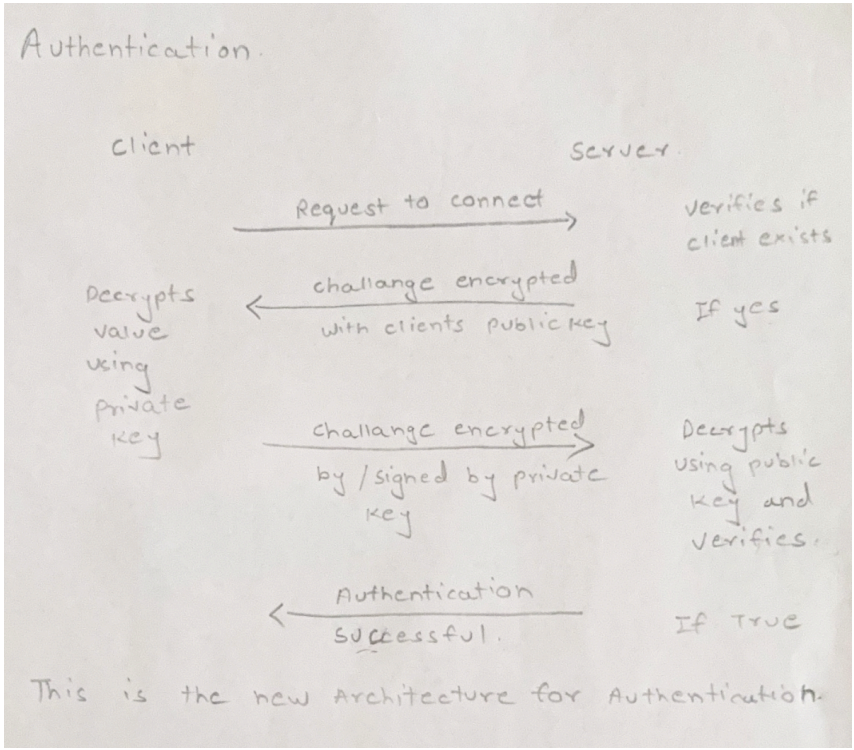
1. **Client Initialization:** The client application initializes and attempts to establish connections with the server on designated ports for various functionalities (authentication and control).
2. **Client Identification:** Upon successful connection, the client sends its name to the server through the control socket and authentication socket connections.
3. **Client Authentication**: Server authenticates the requesting client through an exchange of messages as part of a public key crypto-based authentication (RSA) protocol.
4. **Server Acknowledgement:** Upon receiving the client's name and authenticating it, the server sends an acknowledgment signal back to the client through the respective socket connection. The client waits for the acknowledgment to ensure successful connection establishments. Once authentication is complete, the server closes the authentication socket.

5. **Handler Thread Creation:** After the acknowledgment, the server creates separate handler threads for each client connection to manage communication and requests, which is ultimately used to help clients establish peer connections.

4.2 Client Authentication

Security Modules: Java. Security and Javax.Crypto

a. When a client sends a connect request to the server for the first time the server registers the client and creates the keys for them.
b. Every client will have their own private key, while the server has a copy of their public key.
c. When they wanna initiate a communication, they will start with the authentication. Challenge is an encrypted nonce with a timestamp.



d. This is how the secure Authentication will take place using RSA keys and Challenge-Response Mechanism

4.3 Session Credential Generation

1. Clients send a message to the server asking for a list of available clients.

2. The server responds with the list by checking for the availability status of its clients. A client is only included in this list if it is idle.
3. Client sends the client's name that it wants to connect to, to the server.
4. The server returns session credentials to both clients. This session key is computed by concatenating respective public keys, a nonce for freshness, timestamp, hashing this value, and extracting the first 128 bits.
5. The clients each respond with an acknowledgement. After receiving the acknowledgement from both parties, the server sets both their status to busy until it receives a session closure request.
6. If a client does not see its destination in the list, it sends a default name which leads to the server assuming the communication is ended, and it handles a client closure.

4.4 Chat Messaging

1. Clients send encrypted chat messages, along with a keyed hash of the message as strings to the intended client.
2. Receiving client decrypts the message using the session key, computes the hash, and compares the calculated and received hashes for message integrity.

4.5 Client Control

1. Clients send a target client chat initiation request, Target client accepts the request.
2. Clients send a "quit" command to the receiving interrupt from the client when they want to gracefully terminate their connections.
3. Upon receiving the "quit" command, the client will close the corresponding client connection and send a "quit" message to the server, who updates the client statuses accordingly.
4. Clients send a quit command if they want to terminate all sessions, in which case a graceful exit is facilitated by closing all connections from respective client to server and other client(s).

5. Error Handling

1. The application includes error handling mechanisms to deal with exceptions such as IOExceptions and SocketExceptions.
2. Graceful handling of client disconnections ensures smooth operation of the server.

6. Logging

1. The server logs authentication-related activities to a file named "auth.log". Public keys of all clients, current and registered, are stored in the "Resources" folder.
2. The server also logs the creation and end of chat connections in a file named "activity.log".
3. Logs include timestamps and relevant information about client connections and disconnections, or connection start and end.

7. Security

7.1 Design Requirements

a. **Confidentiality:** The messages exchanged between the server and a client and between any two clients in a session must be secure against passive (unauthorized viewing) and active (unauthorized modification or deletion) attacks. The only security primitive that is available to build a mechanism for confidential/encrypted communication is a custom keyed-hash that uses a chosen hash-function (e.g., SHA-256).
b. **Freshness:** The program must be able to protect an IM session against replay attacks.
c. **Integrity:** The possible message alteration in transit must not go undetected by the communicating parties. The solution for integrity verification should also be implemented using keyed hash

7.2 Security Implementations

A. Session Freshness Guarantee

The session credential is generated as {P1||P2||N||timestamp} where P1 and P2 are derived from public keys of the communicating clients. The nonce, along with the timestamp, provide freshness guarantee.

B. Secure Authentication of Client to Server Using RSA

Clients are first authenticated using their public-key cryptosystem credentials, the public keys for which are reliably stored in the server which is assumed to be the legitimate host.

C. Secure Transmission of Messages using Symmetric Key Encryption:

a. **Confidentiality:** Clients send messages encrypted using a symmetric session key. The encryption method is based on the use of keyed hashes like mixing the plaintext, which so far has been proven to provide security.

b. **Integrity:** Attached to the encrypted message is a keyed hash of the message, both for non-repudiation, as well as to prove integrity of exchanged messages.

c. **Freshness:** The nonce ensures the freshness

D. Attacks considered:

1. **Replay Attack:** Attackers may attempt to capture and resend valid messages to trick recipients. Implementing unique message identifiers and timestamps to generate the session key, which is used to both encrypt and hash the messages will prevent this.
2. **Man in the middle attack:** Attackers may attempt to intercept communications between users, allowing them to eavesdrop or alter messages. Use of end-to-end encryption and keyed hashing for integrity verification will help prevent this.
3. **Data Leak:** Sensitive information may be exposed through insecure transmission. Employing encryption in transit can help protect user data.

8. Future Improvements

1. Implementation of an associated GUI for better client experience
2. Better security features for the Peer-to-Peer communication, like using AES with keyed hash.
3. Allowing 1-to-many client mappings, i.e, allow for multiple simultaneous sessions, as well as group chat features.

9. Contribution

1. Authentication protocol and infrastructure - Akshitha
2. Authentication functional implementation- Karan
3. Chat / Messenger (user-facing end) - Karan
4. Chat / Messenger (functional implementation) - Akshitha
5. Features – Both
6. Security Features - Both