UDP-based Chat Application in Python

Gokul Swaminathan
COMPE 560
San Diego State University
Graduate Student

ABSTRACT

In this paper, I describe the implementation and design decisions of my UDP-based chat application developed as part of COMPE 560.

1. INTRODUCTION

The primary goal of this assignment was to create a client-server chat application incorporating secure and reliable communication through a hybrid key exchange system, message encryption, message integrity, a terminal-based UI, and a simulation of reliable UDP communication.

2. SYSTEM ARCHITECTURE

The application follows a client-server architecture.

crypto_utils.py encapsulates all cryptographic algorithms and functions. This includes RSA key generation, RSA encryption/decryption, AES key generation, AES encryption/decryption, HMAC generation/verification, and helper functions for message encoding/decoding.

server.py holds the central server application. It is responsible for listening to UDP packets from clients. It manages secure key exchange by distributing AES keys encrypted with clients' RSA public keys. It receives encrypted and authenticated chat messages from clients, and verifies message sequence numbers and sends acknowledgements for client-to-server chat messages. Decryption and broadcasting to other connected clients is also done in this file. The server also handles multiple client packets with threading.

client.py holds the client application and provides a user interface for chatting. It is responsible for generating an RSA key pair for secure key exchange, requesting and securely receiving an AES session key, providing a UI for sending and displaying messages, encrypting messages, HMAC-ing messages, reliably sending chat messages, receiving messages, verifying and decrypting messages, and using threading for reliable message sending and UI responsiveness.

Communication is done over UDP sockets, with custom JSON-based packets to describe message types (*KEY_REQ, KEY_REP, DATA, ACK*), sequence numbers, and payloads.

3. CORE FUNCTIONALITY

3.1 UDP Socket Programming

The server binds to a specific port and listens for datagrams. Clients send datagrams to the server's address and port. Python's *socket* module with *SOCK_DGRAM* was used for all network communication.

3.2 Hybrid Encryption - Secure Key Exchange

On the client-side, upon startup, each client generates a 2048-but RSA public/private key pair using *Crypto.PublicKey.RSA*. The public key (PEM formatted, Base64 encoded) is then sent to the server in a *KEY_REQ* packet.

On the server-side, when the server receives a *KEY_REQ* packet, it generates a unique 128-bit AES symmetric key with *Crypto.Random.get_random_bytes* for that client. This AES key is then encrypted using the client's provided RSA public key with PKCS#1 OAEP padding (*Crypto.Cipher.PKCS1_OAEP*).

The RSA-encrypted AES key is sent back to the client Base64 encoded in a *KEY_REP* packet. The client uses its RSA private key to decrypt the AES key and establishes a shared secret AES key for later

message encryption.

3.3 Hybrid Encryption - Message Encryption and Decryption

Once the AES key is shared, all chat messages are encrypted using AES-128 in Cipher Block Chaining (CBC) mode (*Crypto.Cipher.AES*).

A random 16-byte Initialization Vector (IV) is generated for each message using *Crypto.Random.get_random_bytes* and prepended to the ciphertext. PKCS#7 padding (*Crypto.Util.Padding.pad* and *unpad*) is used to handle messages not aligning with the AES block size.

3.4 Message Integrity and Authenticity (HMAC)

HMAC-SHA256 (*Crypto.Hash.HMAC* and *Crypto.Hash.SHA256*) was implemented to ensure message integrity and authenticity. The HMAC tag is calculated over the IV + Ciphertext bytes using the shared AES key as the HMAC secret key. The HMAC tag is then prepended to the IV + Ciphertext before Base64 encoding. Receivers verify the HMAC tag before decryption. If verification fails, the packet is considered invalid.

3.5 Data Encoding and Packet Structure

All binary cryptographic outputs (encrypted keys, public keys, HMAC tags, IV + Ciphertext) are Base64 encoded to ensure sage transmission as strings within the JSON payloads. These JSON structures were defined for UDP packets and include fields like *type* (e.g., *KEY_REQ, KEY_REP, DATA, ACK*), *seq* (for sequence numbers), *ack_seq* (for acknowledgements), and *payload* (for the Base64 encoded data).

4. ADVANCED FEATURES

Advanced features like UI and logging and error handling to represent packet loss and retransmissions were added as per the requirements for Graduate students.

4.1 Terminal-based UI (Curses)

The client has a terminal user interface (TUI) built using Python's *curses* library.

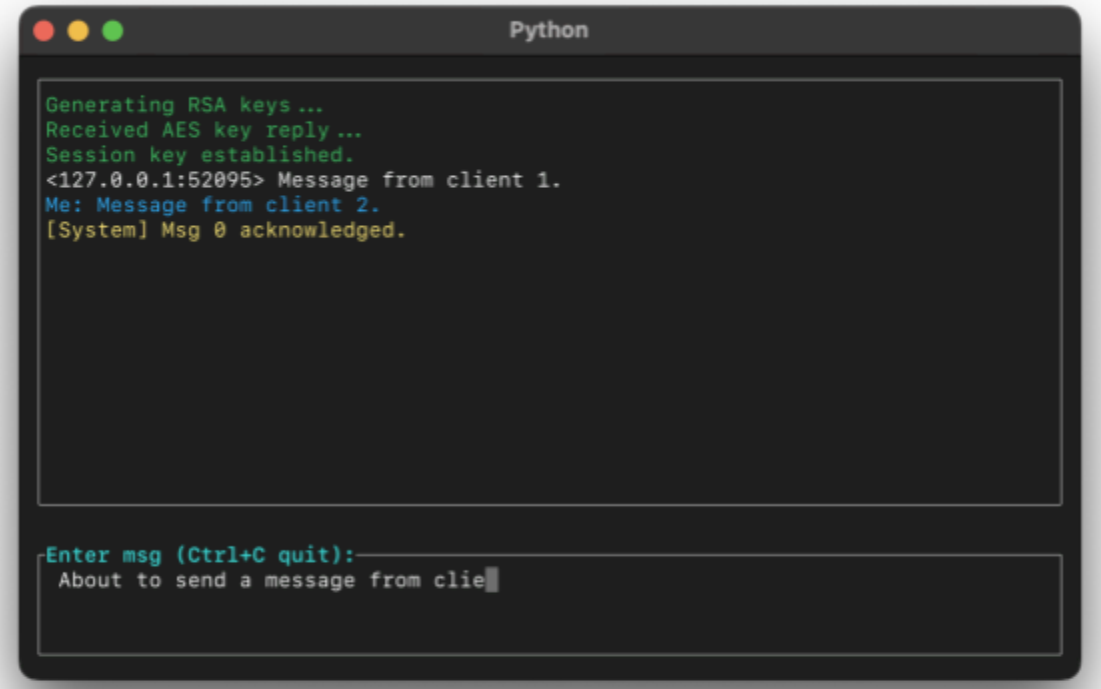

Figure 1: Screenshot of client application

The UI is divided into a main message display window and an input box at the bottom where users type their messages, as shown in Figure 1. It supports real-time display of incoming messages, system notifications, error messages, and messages sent by the user (prefixed with "Me:" and colored differently). It handles basic input (printable characters, backspace, enter) and attempts basic terminal resize handling.

4.2 Logging and Error Handling for Packet Loss/Retransmissions

To simulate reliable UDP for client-to-server chat messages, logging, client-side reliability, and server-side reliability were implemented.

Python's *logging* module was used for both client (to *client.log*) and server (to the console) to record events, errors, packet transmissions, ACKs, timeouts, and retransmissions.

Each outgoing chat *DATA* packet is assigned a unique and incrementing sequence number (*send_seq*) for client-side reliability. The client also expects an *ACK* packet from the server containing the sequence number of the *DATA* packet it's acknowledging. Send *DATA* packets are stored with their sequence number, data, timestamp, and retry count until an ACK is received. If an ACK is not received within the defined timeout (2 seconds), the packet is considered lost. Timed-out packets are retransmitted up to a maximum number of retries (5). If they are still unacknowledged, the message is considered undeliverable and an error is logged. Also on the client-side, user messages are placed in a queue and processed by a dedicated *reliable_sender* thread, which implements a stop-and-wait like protocol.

For server-side reliability, the server maintains the next expected sequence number (*client_expected_seq*) for each known client. Upon receiving an in-sequence *DATA* packet from a client, the server processes it and sends back an *ACK* packet containing the received sequence number. If a *DATA* packet with a previously acknowledged sequence number is received (client retransmission due to a lost ACK), the server resends the ACK for that sequence number. Out-of-order *DATA* packets (future sequence numbers) are currently ignored by the server as no logic is implemented for them.

To summarize, the reliability mechanisms were primarily implemented for chat messages sent from the client to the server. The initial key exchange and the server-to-client broadcasts remain best-effort UDP.

5. DESIGN DECISIONS

5.1 Cryptographic Library

pycryptodome was chosen to align with the example code given in the assignment prompt.

5.2 Hybrid Encryption (RSA + AES)

This is a standard and robust approach. RSA is suitable for the secure exchange of the smaller AES session key, while the faster AES cipher is used for encrypting the bulk chat message data.

5.3 AES Mode (CBC with Random IV & PKCS7 Padding)

CBC mode provides strong confidentiality when used with a unique, random IV for each encryption. PKCS7 padding is a standard way to handle variable-length input for block ciphers.

5.4 Encrypt-then-MAC (HMAC-SHA256)

Applying the HMAC to the ciphertext (including the IV) is generally considered more secure than MAC-then-Encrypt, as it ensures the integrity of what was actually encrypted. SHA256 provides a strong hash function.

5.5 Threading

Threading is implemented on the server side to handle each incoming client packet in a separate thread. This improves server responsiveness by preventing a single slow client operation from blocking the server's ability to handle other clients. A *threading.lock* is used to protect shared data structures.

On the client side, separate threads are used for network reception (*network_receiver*) and reliable message sending (*reliable_sender*). This keeps the UI responsive to user input while background network operations occur.

5.6 JSON for Packet Structure

JSON was chosen for its human-readability, ease of use in Python, and flexibility.

6. CODE AND REPORT REFERENCES

[1] Python Software Foundation. 2025. socket — Low-level networking interface — Python 3.1x.x documentation. Retrieved May 8, 2025 from https://docs.python.org/3/library/socket.html.

[2] Python Software Foundation. 2025. threading — Thread-based parallelism — Python 3.1x.x documentation. Retrieved May 8, 2025 from https://docs.python.org/3/library/threading.html.

[3] Python Software Foundation. 2025. base64 — Base16, Base32, Base64, Base85 Data Encodings — Python 3.1x.x documentation. Retrieved May 8, 2025 from https://docs.python.org/3/library/base64.html.

[4] Python Software Foundation. 2025. json — JSON encoder and decoder — Python 3.1x.x documentation. Retrieved May 8, 2025 from https://docs.python.org/3/library/json.html.

[5] Python Software Foundation. 2025. logging — Logging facility for Python — Python 3.1x.x documentation. Retrieved May 8, 2025 from https://docs.python.org/3/library/logging.html.

[6] Python Software Foundation. 2025. Curses Programming with Python — Python 3.1x.x documentation. Retrieved May 8, 2025 from https://docs.python.org/3/howto/curses.html.

[7] The PyCryptodome Project. 2025. PyCryptodome Documentation (Version 3.x.x). Retrieved May 8, 2025 from https://www.pycryptodome.org/en/latest/.

[8] Tanenbaum, A. S., Feamster, N., and Wetherall, D. J. 2021. *Computer Networks*. 6th ed. Pearson Education.

[9] Stallings, W. 2003. *Data and Computer Communications*. 7th ed. Prentice Hall.