Encrypted Chat Application Documentation

Overview:

This project is an encrypted chat application using RSA encryption for secure communication between a client and a server. The goal is to enable a client-server communication system that ensures messages are securely transmitted over the network. The frontend GUI is built using Tkinter, allowing a simple and intuitive interface for both the server and the client. This application is a demo and does not use multi-threading per client but instead relies on a single thread for handling the network logic and another thread for the Tkinter GUI.

Project Structure:

- Backend (Python Server):
  - Manages the communication between the client and server using TCP sockets.
  - Implements RSA encryption to securely exchange messages between the client and the server.

- Key files include:
  - Server.py: The main server file that handles the core logic for connections, message exchanges, and encryption.
  - RSA.py: Contains logic for RSA encryption and decryption.

- Frontend (Tkinter GUI):
  - Provides a graphical user interface for users to interact with the server and client.
  - Implements message sending and receiving functionality in the background to avoid blocking the interface.

- Key files include:
  - Tkinter GUI code: Both the client and server GUIs are built using Tkinter to handle message input, display, and connections.

- RSA Encryption:
  - Both the client and server generate their own RSA key pair (public and private keys).
  - During the connection, they exchange public keys, allowing the encryption of messages on one end and decryption on the other end using their respective private keys.
  - This ensures that any intercepted messages remain unreadable to third parties.

- Connection Handling:
  - The client connects to the server using a TCP socket, and the server listens on a specified IP address and port.
  - Once connected, the server can send and receive messages to and from the client, all of which are encrypted using RSA.

- Message Sending and Receiving:
  - The messages are encrypted before being sent and decrypted once received.
  - Both the client and server run a continuous loop to listen for incoming messages, ensuring that the chat remains responsive.

Changes in Architecture:

Initially, the system was designed to use one thread per client for handling multiple connections. However, the current implementation uses one thread for the Tkinter GUI and one additional thread for handling network communication. This means that the server can only handle one client at a time in this setup.

The threading model consists of:

1. Main Thread: Responsible for the Tkinter GUI.
2. Background Thread: Responsible for handling the network communication (both sending and receiving messages).

Challenges Encountered:

- No Multithreading for Each Client:
  Unlike the original plan, we do not spawn individual threads per client in this system. This choice simplifies the logic but limits the system to handling a single client connection. Future scalability might require revisiting this decision and introducing either a thread-per-client model or an event-driven system like asyncio for better concurrency handling.

- Handling Long-Running Tasks (GUI Blocking):
  Since Tkinter is not asynchronous, it was crucial to ensure that network operations (which could potentially block) run in a separate thread. This keeps the GUI responsive, allowing users to send and receive messages without freezing the interface.

RSA Key Management:

Transmitting and managing RSA keys securely between the client and server required careful design. The public keys were serialized using Python's pickle library, which allowed easy transmission over sockets. However, care was taken to avoid serialization security issues by validating the received data before deserializing it.

Strengths and Weaknesses:

- Strengths:
  - Security: The use of RSA encryption ensures that all message exchanges between the client and the server are secure.
  - Simple GUI: The Tkinter interface is straightforward and user-friendly.
  - Separation of Concerns: The background thread handles all networking logic, ensuring that the GUI remains responsive.

- Weaknesses:
  - Single Client Limitation: The current implementation supports only one client at a time due to the lack of multithreading for each client. A real-world chat system would need to handle multiple clients concurrently.
  - No Asynchronous Communication: Since the communication is handled in a single background thread, it does not leverage an event-driven or asynchronous model, which could be more efficient for handling multiple clients.

Key Code Components:

- RSA Encryption:
    - Python
      quoted code:
      public_key, private_key = rsa.generate_keypair(1024)
      encrypted_message = rsa.encrypt(plain_text, public_key_remote)
      decrypted_message = rsa.decrypt(encrypted_message, private_key)
      Message Sending and Receiving:

    - Python
      quoted code:
      def send_message():
        message = str.encode(edit_text.get())
        encrypted_message = rsa.encrypt(int(binascii.hexlify(message), 16),
      public_key_remote) conn.send(str(encrypted_message).encode())

    - def recv():
        response_message = int(conn.recv(1024).decode())
        decrypted_message = rsa.decrypt(response_message, private_key)
        listbox.insert(END, f"{name1}: {str(decrypted_message)}")

- Threading Model:
    - Python
      quoted code:
      threading.Thread(target=recv).start()

    - def handle_client():
        threading.Thread(target=receive_messages, daemon=True).start()

- Improvements and Future Work
    - Multithreaded Client Support: The server could be extended to handle multiple clients by spawning a thread for each client that connects or by using an event-driven framework like asyncio. This would allow the server to handle multiple clients simultaneously, which is essential for a scalable chat system.
    - Asynchronous Communication: To further improve the efficiency of the application, an event-driven model could replace the current thread-based model. Python's asyncio library is well-suited for handling multiple I/O-bound tasks (like socket communication) concurrently without blocking the main thread.
    - Error Handling: The current system has basic error handling. More robust error handling and logging could be added to ensure that unexpected errors (e.g., network failures) are captured and managed gracefully without crashing the application.

Conclusion:

This encrypted chat application is a proof-of-concept that demonstrates the use of RSA encryption to secure client-server communication. While the application works well for a single client, future iterations would need to focus on scaling the server to handle multiple clients concurrently and improving the system's resilience and performance.

Author: Nissim Brami