CSCD58 Project Report

SpideyBot

December 2023

Contents

1 Introduction

SpideyBot is a chatbot that builds on principles of networking such as UDP, TCP packets and encryption. Available in three interfaces: CLI, UI, or as a discord bot. Invokes OpenAI's GPT-3.5-turbo LLM model on input supplied to the bot and relays GPT's output back to the user.

1.1 Team Members and Contributions

- Leila Cheraghi Seifabad, 1007465495
  Implemented all the different clients: CLI, Discord, and UI. Also responsible for integrating each client with the main server. Also helped with encryption in the TCP server.

- Maaz Hashmi, 1006804718
  Setup the TCP and UDP servers and clients and also implemented main server handling communication between both parties. Also worked on encryption and decryption using RSA for the TCP handler.

- Mohannad Shehata, 1006774787
  Worked on the main server and helped with client integration. Also responsible for setting up the GPT model and integrating it with the main server.

1.2 Rationale

After learning about networks we wanted to investigate how much we take for granted in our daily use of applications from the networking point of view so we started with an existing application and used our newfound networks knowledge to connect a user to these services (along with additional features we explore below).

1.3 Methodology

We create a server that we can process TCP/UDP requests sent by a client and then extract the messages inside them and uses them as queries to GPT. After receiving a response it sends the response back to the client. Interactions between the client and the server are encrypted,and both exchange public keys as soon as the connection is formed.

We allow the client the option to choose between sending either UDP or TCP packages, and it is possible to switch between both without terminating the program (see below). In addition to the CLI option, we also give the users the possibility of using a UI or having it communicate via a discord bot added in the desired server.

The main obstacle we faced was that the server/client behaved differently based on the device they were running on. We recommend running it on linux or at least running it in a docker container.

2 Implementation

2.1 Overview

We create a server that can communicate with both GPT and clients. For it to communicate with GPT it uses an OpenAI API key which we do not supply here for privacy reasons, but one can easily create their own. To decrease response times and make the interaction resemble that of a usual chatbot we first prompt GPT to send shorter messages before sending messages, using the following:

```
prompt = "You're a chatbot for a quick one to one chat application with a human.  Limit your
responses to the following questions to a sentence or 2 max."
```

We also use multiprocessing to enable the server to run 2 distinct processes to handle TCP and UDP communications respectively.

As for the client, we have multiple versions with similar implementation principles, or more precisely the mechanism by which they send messages to the server is similar. The CLI client allows users to send messages from the terminal, the UI client allows users to input their queries in a text box and receive the answer in a format similar to usual desktop applications, while the Discord bot interfaces with a particular Discord server. The commands are entered through a Discord chat and read by a client program using the Discord.py API, which sends it to a server program, receives a response, and then posts the response back to Discord.

We implement end-to-end encryption using RSA PKCS#1 v1.5 algorithm for TCP connections. The server generates an RSA key pair on startup and the client generates an RSA key pair on successful connection to a server. On connection, both parties exchange their public keys establishing a secure communication channel. Then the client encrypts each subsequent (TCP) message with the server's public key and the server (and only) the server can decrypt this message using its own private key. A similar mechanism is used by the server to encrypt messages it sends back to the client.

2.2 Code Structure / Documentation

The application code lives under the the `src` directory. `tcp.py` and `udp.py` setup the respective handlers for managing TCP and UDP sockets. TCPClient and TCPServer also contain the logic for encryption and generating keys etc. `gpt.py` sets up the GPT model class for retrieving responses based on prompts from the OpenAI API. `server.py` is main server handling core logic for receiving responses from client and forwarding to server and vice versa. The `Server` class also uses multiprocessing to listen for both TCP and UDP messages simultaneously. Then each of the `client_{cli,ui,discord}.py` files contain the logic for for setting up client side interfaces and setting up a connection to the server using the `Handler`. Here's some specific documentation for some of the more important server classes and functions:

- `server.py`
  - `Server`
    The main server class that handles both TCP and UDP connections. It initializes TCP and UDP servers and manages the communication with clients. It processes messages received over both protocols, interacts with a GPT-based chatbot, and maintains conversation contexts for each client.
  - `process_tcp(self, tcp_conn: "tcp.TCPServer.ClientConnection")`
    Handles the processing of messages received over TCP connections. It communicates with the GPT-based chatbot and manages the conversation context for each client.
  - `process_udp(self)`
    Handles the processing of messages received over UDP connections. Similar to `process_tcp`, it interacts with the GPT-based chatbot and maintains the conversation context for each client.
  - `new_tcp_client(self, client: "socket.socket", address: str) -> None`
    Initiates a new TCP client connection by creating a new process for each client using the `TCPServer.ClientConnection` class.
  - `udp_client(self) -> None`
    Initiates a new process for handling UDP messages.
  - `start(self) -> None`
    Starts the server by accepting TCP connections and initiating processes for handling TCP and UDP messages.
- `tcp.py`

- `TCPHandler`
  Handles the basic setup for TCP connections, including the generation of RSA key pairs for encryption. It can be instantiated as either a server or client, with the option to provide existing private and public keys.
- `TCPServer`
  Represents a TCP server, utilizing the TCPHandler for basic connection setup. It includes a method to accept incoming connections and a nested class ClientConnection to manage individual client connections.
- `TCPClient`
  Represents a TCP client, utilizing the TCPHandler for basic connection setup. It includes a method to exchange public keys with the server (`exchange_keys`), and methods to send and receive encrypted messages.
- `generate_rsa_key_pair (inside TCPHandler)`
  Generates a new RSA key pair of 4096 bits using the `rsa` library for secure communication.
- `connect_socket (inside TCPHandler)`
  Establishes a TCP connection based on the type (SERVER or CLIENT) specified during the instantiation of the `TCPHandler` class.
- `close (inside TCPHandler)`
  Closes the socket connection.
- `accept_connection` (inside `TCPServer`)
  Accepts an incoming connection on the server and returns the client socket and address.
- `exchange_keys` (inside `ClientConnection` and `TCPClient`)
  Exchanges RSA public keys between the server and client for secure communication.
- `send_msg` (inside `ClientConnection` and `TCPClient`)
  Sends an encrypted message using the recipient's public key.
- `recv_msg` (inside `ClientConnection` and `TCPClient`)
  Receives and decrypts an encrypted message using the receiver's private key.
- `close` (inside `ClientConnection`)
  Closes the client socket.

- `udp.py`
  - `UDPServer`
    Represents a UDP server, utilizing the `socket` module for basic connection setup. It includes methods to send and receive messages, as well as to close the server instance.
  - `UDPClient`
    Represents a UDP client, utilizing the `socket` module for basic connection setup. It includes methods to send and receive messages, as well as to close the client instance.
  - `send_msg (inside UDPServer and UDPClient)`
    Sends a message to the server/client and returns the number of bytes sent.
  - `recv_msg (inside UDPServer and UDPClient)`
    Receives a message from the server/client and returns the decoded message.
  - `close (inside UDPServer and UDPClient)`
    Closes the UDP client instance and returns `True` if successful, `False` otherwise.

2.3 Usage

Detailed instructions for running the application can be found in the README. We recommend using a Linux machine or running the application in a docker container (Dockerfile provided).

3 Analysis

The implementation of the SpideyBot involves the use of TCP and UDP communication, encryption using RSA, and integration with the GPT-3.5-turbo LLM model. The use of multiple interfaces (CLI, UI, and Discord) adds versatility to the application.

3.1 Networking Principles

The project effectively utilizes networking principles such as TCP and UDP for communication between the server and clients. The TCPHandler, TCPServer, and TCPClient classes encapsulate the setup for TCP connections, including the generation and exchange of RSA keys for secure communication. The UDPServer and UDPClient classes similarly handle UDP communication.

3.2 Security Measures

The application employs end-to-end encryption using the RSA PKCS#1 v1.5 algorithm for TCP connections. The exchange of public keys during the connection setup establishes a secure communication channel. This ensures that the messages exchanged between the server and clients remain confidential.

3.3 GPT Integration

The integration of the GPT-3.5-turbo LLM model enhances the chatbot's responses, providing more natural and context-aware interactions. The use of multiprocessing to handle TCP and UDP communication simultaneously contributes to efficient communication and response handling.

3.4 Client Interfaces

The implementation of multiple client interfaces (CLI, UI, and Discord) allows users to interact with the ChatBot in diverse ways. Each client version follows a similar mechanism for sending messages to the server, providing a consistent user experience.

3.5 Challenges and Recommendations

The project faced challenges related to platform-specific (or processor-related ?) behavior, and the README provides guidance for optimal execution on Linux or within a Docker container. Future enhancements could focus on improving cross-platform compatibility and addressing potential issues that may arise on different operating systems.

We also faced challenges regarding encryption using RSA. We found that the RSA PKCS #1 v1.5 encryption algorithm requires the message to of no longer than a certain length and this length is dependent on the key size used to encrypt. To get around this issue, we increased our key size to a maximal 4096 bytes and also decided to pre-prompt GPT to keep its responses short. We thought other possible solutions to this problem such as using a hybrid AES/RSA encryption, or splitting the message into chunks and encrypt/decrypt these chunks as received. But with thorough research and time constraints we stuck with our original solution.

4 Conclusion

The SpideyBot project successfully combines networking principles, encryption, and advanced language modeling to create a versatile and secure communication system. The use of different client interfaces enhances user accessibility, and the integration of GPT-3.5-turbo contributes to the natural and context-aware conversational experience. The implementation demonstrates a practical application of network concepts and encryption techniques in a real-world scenario, highlighting the significance of secure and efficient communication in modern applications.