Programming Assignment 1 for CSE232

UDP Pinger

Palak Bhardwaj (2022344)
Yashovardhan Singhal (2022591)

16 September 2024

Abstract :

This report provides an overview of **TCP** based web application done as a part of the CN programming assignment 2.

In part (a) -

- The client sends 10 pings to the server. If the client doesn't receive a response in under 1 second, it is assumed that the packet is lost during transmission.
- The packets are sent to the local host, both the client and the server are hosted on a single device.
- The client calculates the round-trip time for each packet.
- The Minimum, Maximum, and Average RTTs are calculated at the end of all pings from the client.
- The packet loss rate is observed (as a percentage).

In part (b) -

- A UDP heartbeat application is made.
- IT is observed as to how many times the UDP Heartbeat packets are sent before you see 3 consecutive responses missing were found to be missing.

.

The initial server code was given in the program. A server socket is created by the server, which represents an end of a communication link (TCP in this case) -

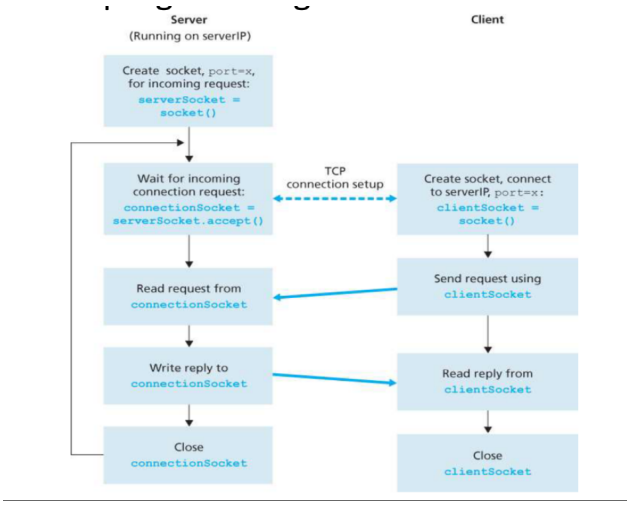

FIGURE 1 – Socket Programming with TCP (CSE232 Slides by Prof. BN Jain)

Some of the Standard Socket types

```
/*Standard socket types */
#define   SOCK_STREAM             1 /*virtual circuit*/
#define   SOCK_DGRAM              2 /*datagram*/
#define   SOCK_RAW                3 /*raw socket*/
#define   SOCK_RDM                4 /*reliably-delivered message*/
#define   SOCK_CONN_DGRAM         5 /*connection datagram*/
```

In the assignment application **SOCKSTREAM** Socket type is used for a TCP connection. It ensures that data packets arrive without loss, duplication, or corruption, which is essential for most web applications. SOCKSTREAM maintains the order of packets, making sure that data is read in the exact sequence it was sent. TCP checks for errors and retransmits lost packets, providing error correction capabilities that UDP (SOCKDGRAM) lacks.

Web servers, FTP servers, and email services all rely on TCP sockets (SOCKSTREAM) to manage data transmission reliably.

```
serverSocket = socket(AF_INET, SOCK_STREAM)
# This snippet creates a socket
# AF_INET specifies the use of IPv4 address family for specifying IP addresses
```

In the context of the server code, serverSocket is the server's listening socket. This socket waits for incoming client connections on a specified port (in this case, port 80, which is commonly used for HTTP traffic).

Once the Server socket is created, it bounds to an IP address and port using the bind(), allowing it to listen for incoming connections on the specified port (e.g., port 80).

This is a overview of how our code send packets using TCP.

1 Server-Side Functioning - Part 1.

The server in this script is a basic web server implemented using Python's socket module. Here's a step-by-step overview of its functioning :

1.1 Server Setup

Socket Creation :

```
serverSocket = socket(AF_INET, SOCK_STREAM)
```

The server creates a socket with AFINET specifying IPv4 addressing and SOCKSTREAM indicating TCP as the transport protocol.

1.2 Binding to Port :

```
serverSocket.bind(('', serverPort))
```

The server binds the socket to port 80 (serverPort). The empty string '' allows the server to accept connections on any network interface.

1.3 Listening for Connections :

```
serverSocket.listen(1)
```

The server begins listening for incoming connections. The argument 1 specifies that only one connection can be queued while waiting to be accepted.

1.4 Accepting Client Connections

Waiting for Incoming Connection :

```
connectionSocket, addr = serverSocket.accept()
```

When a client tries to connect, the server accepts the connection, creating a new socket (connectionSocket) specifically for communicating with the client. The address variable stores the client's address.

1.5 Receiving Data from Client

```
message = connectionSocket.recv(1024)
```

The server reads the incoming data (up to 1024 bytes which is the buffer size) from the client, an HTTP request message.

1.6 Parsing the HTTP Request

The server extracts the HTTP method and filename from the request, typically in the form of GET /filename HTTP/1.1.

1.7 Serving the Requested File

The server attempts to open the requested file. If successful, it reads the file content and sends an HTTP response back to the client.

1.8 Sending Response to Client

Sending HTTP Header :

```
connectionSocket.send('\nHTTP/1.1 200 OK\n\n.encode()')
#HTTP/1.1 specifies the HTTP version used.
```

The server sends a response header indicating that the request was successful. There is a specific format both for HTTP response messages and request messages. 200 OK signifies that the message was successfully received. The client is also informed about the type of the content being received. Encode() is a method in Python that converts a string into a byte stream, which is required for sending data over network connections. Since network communications deal with binary data, converting strings to bytes is essential.

1.9 Handling Errors

If the server fails to find or open the requested file, it catches the IOError and sends an HTTP 404 Not Found response to the client. This can happen if the requested file is not present at the server or the file cannot be opened.

- -

2 Client-Side Part (1)

2.1 Client Sends an HTTP Request

Request Initiation :

The client (like a web browser) sends an HTTP request to the server, typically using a GET request to fetch a webpage or file.

```
GET /index.html HTTP/1.1
Host: server-ip-address
```

2.2 Waiting for Server Response

The client's request goes over the network to the server, and the client waits for the server to respond.

2.3 Receiving the Server Response

Response Processing :

Upon receiving a response from the server, the client processes the incoming data. For a web browser, this means rendering the HTML, CSS, images, etc., received from the server.

2.4 Handling Errors from the Server

If the server responds with an error (like 404 Not Found), the client displays an error message. For a browser, this is usually shown as a "404 Page Not Found" error page.

2.5 Closing the Connection

Once the server finishes sending the response, the connection is closed. The client can then decide to make another request or end the interaction.

3 Part(2) Server - Multi-threaded Web Server Implementation

3.1 Introduction

In this task, a web server was implemented to handle multiple HTTP requests simultaneously using multithreading. The original server was designed to process one client request at a time. The objective was to modify the server to support concurrent connections, allowing it to serve multiple clients at the same time through individual threads. This approach ensures that the web server remains responsive even when multiple clients connect simultaneously.

3.2 Server Overview

The web server is built using the Python `socket` module to establish a TCP connection and the `threading` module to enable concurrent handling of client requests. The server listens on a fixed port for incoming connections. For each client request, a new thread is spawned, which processes the request independently.

3.3 Main Server Loop

The server socket is created using the `socket()` function with parameters `AF_INET` and `SOCK_STREAM` to specify an IPv4, TCP-based connection. The server is then bound to a specific port (in this case, 6789) using the `bind()` method and listens for client connections using the `listen()` method with a backlog of 5 connections.

When a client connects, the server calls `accept()` to accept the connection. At this point, a new thread is created for each connection using Python's `threading.Thread()` function. This thread runs the `handle_client()` function, which processes the HTTP request and sends the appropriate response.

3.4 Handling Client Requests

The `handle_client()` function is responsible for processing each client's HTTP request. Once the connection is established, the server receives the client's request using `recv()`. The requested file name is extracted from the HTTP request message, and the server attempts to open and read the file from the server's file system.

If the file is found, the server sends back an HTTP 200 OK response along with the file's content. The response header includes the content type as `text/html`. The file content is then sent back to the client in a loop using `send()`.

In case the requested file is not found, the server responds with a 404 Not Found message. If there is an issue with the request itself (such as a malformed request), a 400 Bad Request response is sent.

3.5 Multithreading

Multithreading allows the server to handle multiple requests concurrently. Each client connection is processed in its own thread, ensuring that one client does not block the processing of others. This is achieved by creating a new thread for each connection using the following code snippet :

```
client_thread = threading.Thread(target=handle_client, args=(connectionSocket,))
client_thread.start()
```

This starts a new thread for every client, each running the `handle_client()` function. The main server thread remains free to listen for new incoming connections, while individual threads handle the processing of client requests.

3.6 Conclusion

By utilizing multithreading, the server can now handle multiple client connections simultaneously. Each request is serviced in a separate thread, improving the responsiveness and scalability of the server. This implementation is particularly useful for real-world web servers where numerous clients may connect concurrently. Through this design, we ensure efficient use of system resources and minimize the response time for clients.

4 Part (c) HTTP Client Implementation

4.1 Introduction

In this task, we were required to implement a custom HTTP client to test the web server developed in the previous tasks. This client connects to the server via a TCP connection, sends an HTTP GET request, and outputs the server's response. The client program takes three command-line arguments : the server's IP address or hostname, the server's port, and the filename to be requested. The task also involves testing whether the client works with both the single-threaded and multi-threaded server implementations.

4.2 Client Overview

The HTTP client is implemented using Python's `socket` module. This client behaves similarly to a web browser by sending a GET request to the server and then displaying the response (which includes both the HTTP headers and the content of the requested file). The client program takes in the server's hostname, port number, and the requested file as inputs through the command line.

4.3 Socket Creation and Connection

The client first creates a TCP socket to connect to the server. This is done using the `socket()` function with `AF_INET` (IPv4) and `SOCK_STREAM` (TCP) parameters.

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

After the socket is created, the client connects to the server using the `connect()` method, where the server's host (IP address or hostname) and port number are specified.

```
client_socket.connect((server_host, int(server_port)))
```

This establishes a TCP connection between the client and the server, which allows for the transfer of data (HTTP request and response).

4.4 Sending the HTTP GET Request

Once the connection is established, the client prepares an HTTP GET request to be sent to the server. The request includes several components :

1. **Request Line :** Specifies the GET method and the requested file.
2. **Host Header :** Specifies the hostname or IP address of the server.
3. **Connection Header :** Indicates that the connection should be closed after the response is received.
4. **Blank Line :** Indicates the end of the headers.

These components are combined into a single HTTP request string, as shown below :

```
request_line = f"GET /{filename} HTTP/1.1\r\n"
host_header = f"Host: {server_host}\r\n"
connection_header = "Connection: close\r\n"
blank_line = "\r\n"
http_request = request_line + host_header + connection_header + blank_line
```

The client then sends this request to the server using the `sendall()` method, which ensures that all data is sent through the socket.

```
client_socket.sendall(http_request.encode())
```

4.5 Receiving the Server's Response

After sending the HTTP request, the client waits to receive the server's response. The response includes both the HTTP headers and the content of the requested file. The client receives the data in chunks of 1024 bytes using the `recv()` method in a loop until all the data is received.

```
response = b""
while True:
    chunk = client_socket.recv(1024)
    if not chunk:
        break
    response += chunk
```

The received data is stored in a byte string, which is then decoded and printed to the console for the user to see.

4.6 Handling Errors and Edge Cases

The client is designed to handle potential errors such as connection issues and malformed inputs. For example, if there is an error during the connection or data transfer, an exception is caught, and the socket is closed to avoid resource leaks. The following snippet illustrates this :

```
except Exception as e:
    print(f"Error: {e}")
    client_socket.close()
```

Additionally, the client checks for the correct number of command-line arguments before running, ensuring that the user provides all the required inputs (server host, port, and filename). If the user input is incorrect, an error message is displayed, and the program terminates.

4.7 Command-Line Arguments

The program is run from the command line, where the server's IP address or hostname, port number, and filename must be specified. The following command format is used :

```
client.py <server_host> <server_port> <filename>
```

For example, to request a file named `HelloWorld.html` from a server running on `localhost` at port 6789, the command would be :

```
python client.py localhost 6789 HelloWorld.html
```

4.8 Main Conclusion

This custom HTTP client successfully tests the functionality of both the single-threaded and multithreaded server implementations. It simulates a browser by sending an HTTP GET request and receiving the server's response. The client is able to interact with the server, retrieve files, and display the server's response, making it a valuable tool for testing web server implementations.

Answer The client works with both the servers implemented in Part 1 and Part 2.

4.9 Output

5 References

CSE232 Slides by prof BN Jain.
CSE 232 Tut slides.
SocketProgamming-1-Tut1 and Tut2, CSE232
IBM documenation.