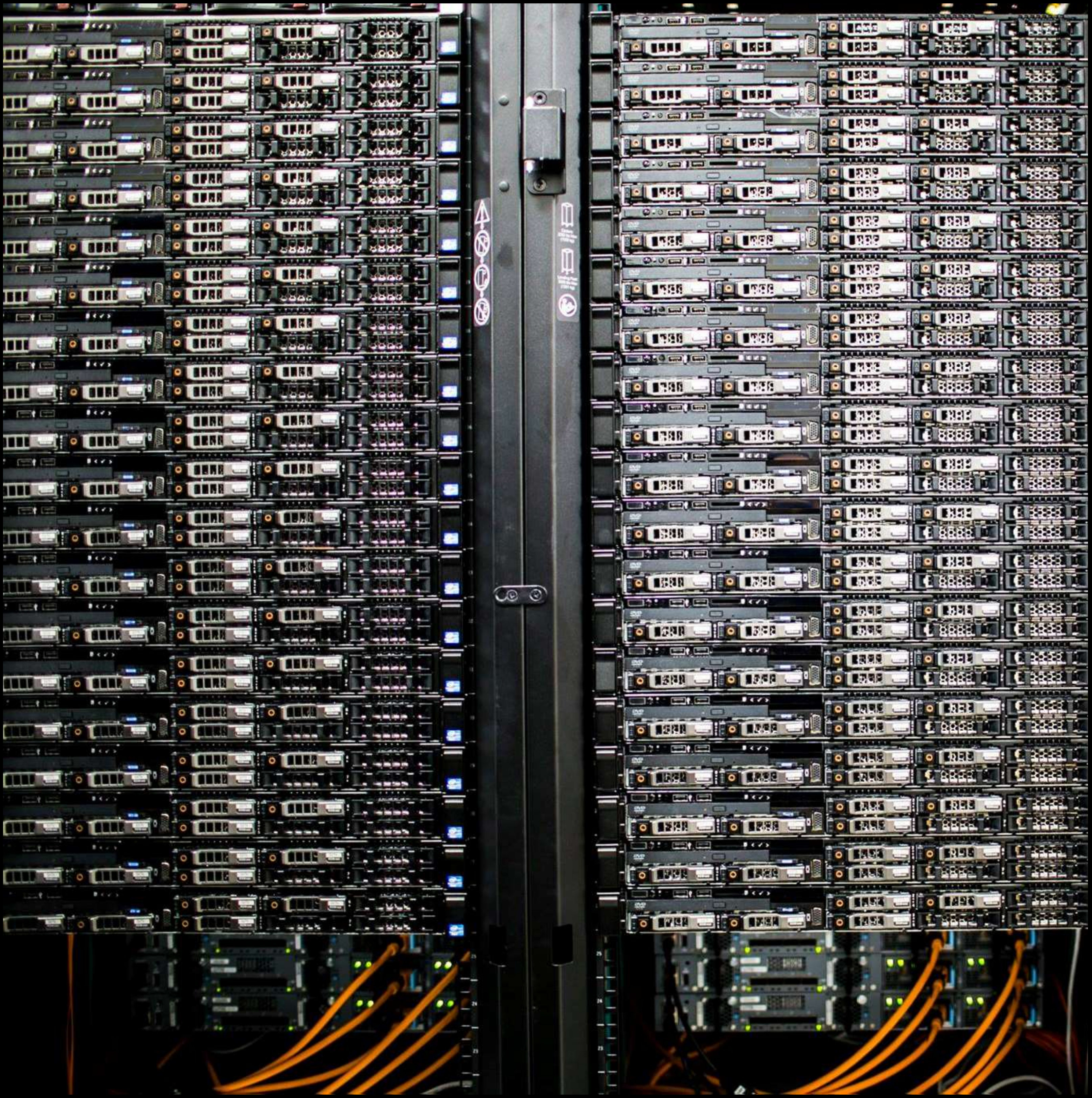


HTTP2 Server

Phase IV

Computer Networks

Ain Shams University
Faculty of Engineering
CESS
CSE351 Computer Networks
Fall 24
Project Document

Project Done by:

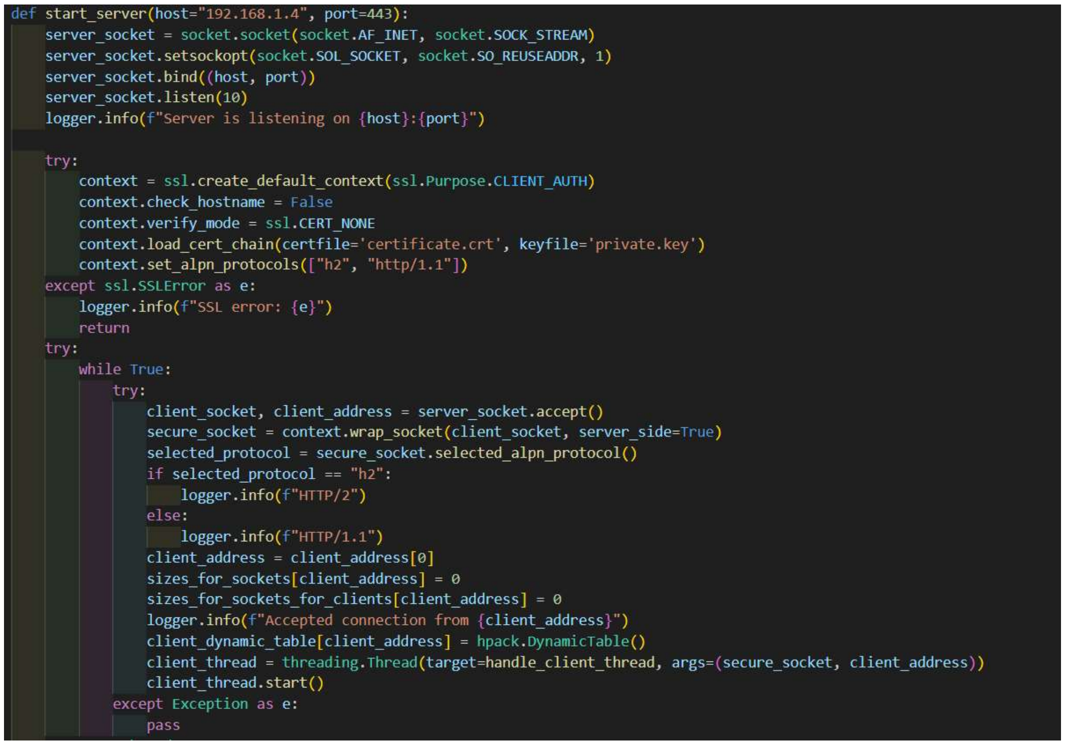
Youssef Mohamed AlSayed	21P0094
Youssef Mohamed Salah	21P0159
Saleh Ahmed ElSayed	21P0324

Table of Contents

SNIPPETS OF IMPORTANT CODE

Connection_handler.py

Accepts TCP connections on port 80 and using threads to handle multiple connections at the same time



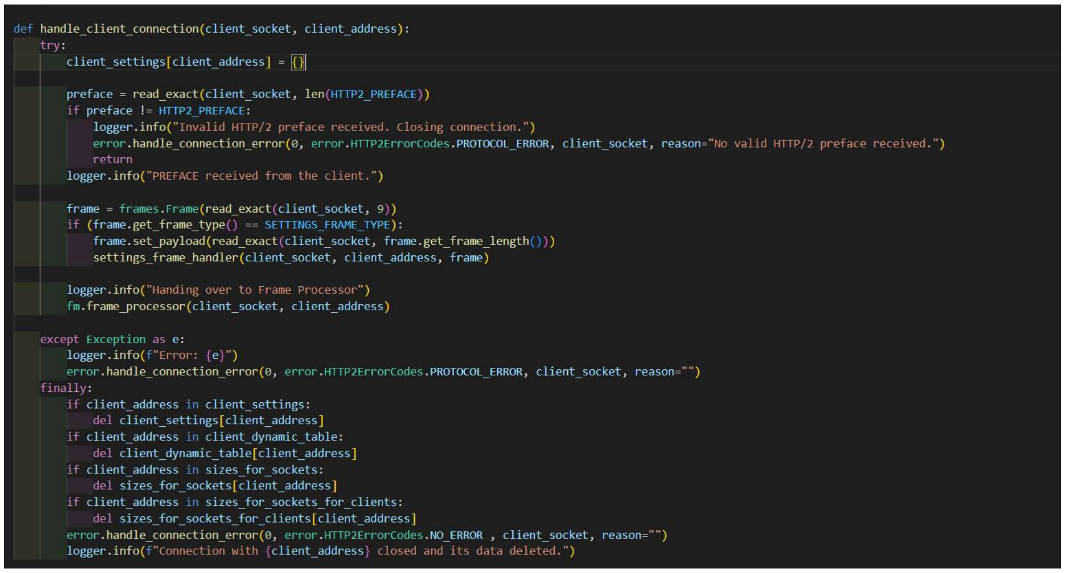
```python
def start_server(host="192.168.1.4", port=443):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server_socket.bind((host, port))
    server_socket.listen(10)
    logger.info(f"Server is listening on {host}:{port}")

    try:
        context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
        context.check_hostname = False
        context.verify_mode = ssl.CERT_NONE
        context.load_cert_chain(certfile='certificate.crt', keyfile='private.key')
        context.set_alpn_protocols(["h2", "http/1.1"])
    except ssl.SSLError as e:
        logger.info(f"SSL error: {e}")
        return
    try:
        while True:
            try:
                client_socket, client_address = server_socket.accept()
                secure_socket = context.wrap_socket(client_socket, server_side=True)
                selected_protocol = secure_socket.selected_alpn_protocol()
                if selected_protocol == "h2":
                    logger.info(f"HTTP/2")
                else:
                    logger.info(f"HTTP/1.1")
                client_address = client_address[0]
                sizes_for_sockets[client_address] = 0
                sizes_for_sockets_for_clients[client_address] = 0
                logger.info(f"Accepted connection from {client_address}")
                client_dynamic_table[client_address] = hpack.DynamicTable()
                client_thread = threading.Thread(target=handle_client_thread, args=(secure_socket, client_address))
                client_thread.start()
            except Exception as e:
                pass
```

This is responsible for creating a secure server that listens for client connections over a specific IP address (192.168.1.4) and port (443). It starts by setting up a socket, which is like a doorway for communication. The server uses SSL/TLS encryption to make sure all data sent and received is secure. It loads a certificate (certificate.crt) and a private key (private.key) that are used to establish trust between the server and clients.

When a client tries to connect, the server checks if the connection is using HTTP/2 or HTTP/1.1, which are two common communication protocols. Once the connection is accepted, it logs the client's information (like their address) and prepares them to communicate securely. It also creates a space (called a dynamic table) to handle specific tasks related to the client's data.

Finally, the server starts a new thread for each client, which allows it to handle multiple clients at the same time without interruptions. Each thread runs a separate function to manage communication with the client, ensuring that messages can be exchanged securely and efficiently. This design makes the server capable of supporting multiple secure connections simultaneously.

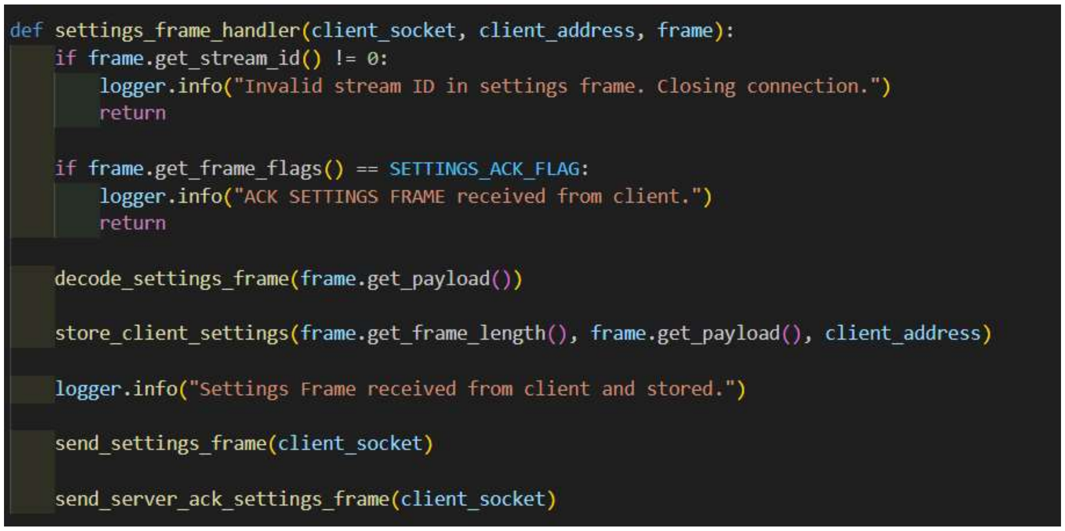
Handling the connection is about receiving the preface for the HTTP2 connection and exchanging the settings frame



```python
def handle_client_connection(client_socket, client_address):
    try:
        client_settings[client_address] = {}

        preface = read_exact(client_socket, len(HTTP2_PREFACE))
        if preface != HTTP2_PREFACE:
            logger.info("Invalid HTTP/2 preface received. Closing connection.")
            error.handle_connection_error(0, error.HTTP2ErrorCodes.PROTOCOL_ERROR, client_socket, reason="No valid HTTP/2 preface received.")
            return
        logger.info("PREFACE received from the client.")

        frame = frames.Frame(read_exact(client_socket, 9))
        if (frame.get_frame_type() == SETTINGS_FRAME_TYPE):
            frame.set_payload(read_exact(client_socket, frame.get_frame_length()))
            settings_frame_handler(client_socket, client_address, frame)

        logger.info("Handing over to Frame Processor")
        fm.frame_processor(client_socket, client_address)

    except Exception as e:
        logger.info(f"Error: {e}")
        error.handle_connection_error(0, error.HTTP2ErrorCodes.PROTOCOL_ERROR, client_socket, reason="")
    finally:
        if client_address in client_settings:
            del client_settings[client_address]
        if client_address in client_dynamic_table:
            del client_dynamic_table[client_address]
        if client_address in sizes_for_sockets:
            del sizes_for_sockets[client_address]
        if client_address in sizes_for_sockets_for_clients:
            del sizes_for_sockets_for_clients[client_address]
        error.handle_connection_error(0, error.HTTP2ErrorCodes.NO_ERROR , client_socket, reason="")
        logger.info(f"Connection with {client_address} closed and its data deleted.")
```

This handles the communication between the server and a client after a connection is established. It begins by creating a space to store settings for the specific client. The code then reads the initial message from the client to check if it is using the correct HTTP/2 protocol. If the message is invalid, the connection is closed with an error. If valid, the server processes additional frames (small chunks of data) sent by the client, such as configuration settings, and prepares them for further handling. In case of any errors during this process, it logs the issue and cleans up any data or resources associated with the client before closing the connection. This ensures smooth and organized communication while managing errors gracefully.

For exchanging the settings frame, it starts by receiving the settings frame of the client and storing it for future usage then sending the server's settings frame then sending the server acknowledgement for receiving the client settings frame and receiving the client acknowledgement

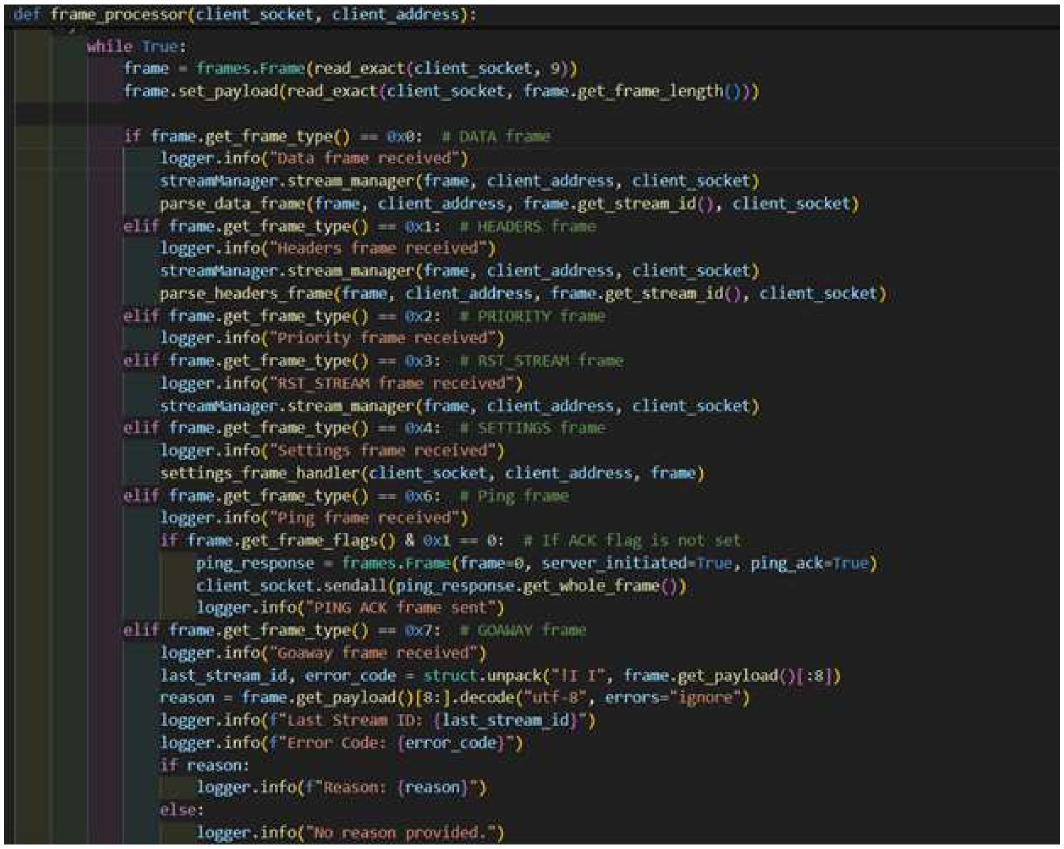


```python
def settings_frame_handler(client_socket, client_address, frame):
    if frame.get_stream_id() != 0:
        logger.info("Invalid stream ID in settings frame. Closing connection.")
        return

    if frame.get_frame_flags() == SETTINGS_ACK_FLAG:
        logger.info("ACK SETTINGS FRAME received from client.")
        return

    decode_settings_frame(frame.get_payload())

    store_client_settings(frame.get_frame_length(), frame.get_payload(), client_address)

    logger.info("Settings Frame received from client and stored.")

    send_settings_frame(client_socket)

    send_server_ack_settings_frame(client_socket)
```

This code processes a settings frame sent by a client in an HTTP/2 connection. It starts by checking if the frame is valid, ensuring the stream ID is zero (as required for settings frames) and that it's not an acknowledgment (ACK) frame; if either check fails, the connection is closed. If the frame is valid, it decodes the settings, stores the client's configuration, and logs that the settings have been successfully received. Finally, the server sends its own settings frame to the client and acknowledges the client's settings, ensuring proper communication setup between the server and client.

Frame_processor.py

Frame processor should handle the frames coming from the client and send it where it belongs, but some frames before that it should go to the stream manager first and we will see that later



```python
def frame_processor(client_socket, client_address):

    while True:
        frame = frames.Frame(read_exact(client_socket, 9))
        frame.set_payload(read_exact(client_socket, frame.get_frame_length()))

        if frame.get_frame_type() == 0x0:  # DATA frame
            logger.info("Data frame received")
            streamManager.stream_manager(frame, client_address, client_socket)
            parse_data_frame(frame, client_address, frame.get_stream_id(), client_socket)
        elif frame.get_frame_type() == 0x1:  # HEADERS frame
            logger.info("Headers frame received")
            streamManager.stream_manager(frame, client_address, client_socket)
            parse_headers_frame(frame, client_address, frame.get_stream_id(), client_socket)
        elif frame.get_frame_type() == 0x2:  # PRIORITY frame
            logger.info("Priority frame received")
        elif frame.get_frame_type() == 0x3:  # RST_STREAM frame
            logger.info("RST_STREAM frame received")
            streamManager.stream_manager(frame, client_address, client_socket)
        elif frame.get_frame_type() == 0x4:  # SETTINGS frame
            logger.info("Settings frame received")
            settings_frame_handler(client_socket, client_address, frame)
        elif frame.get_frame_type() == 0x6:  # Ping frame
            logger.info("Ping frame received")
            if frame.get_frame_flags() & 0x1 == 0:  # If ACK flag is not set
                ping_response = frames.Frame(frame=0, server_initiated=True, ping_ack=True)
                client_socket.sendall(ping_response.get_whole_frame())
                logger.info("PING ACK frame sent")
        elif frame.get_frame_type() == 0x7:  # GOAWAY frame
            logger.info("Goaway frame received")
            last_stream_id, error_code = struct.unpack("!I I", frame.get_payload()[:8])
            reason = frame.get_payload()[8:].decode("utf-8", errors="ignore")
            logger.info(f"Last Stream ID: {last_stream_id}")
            logger.info(f"Error Code: {error_code}")
            if reason:
                logger.info(f"Reason: {reason}")
            else:
                logger.info("No reason provided.")
```

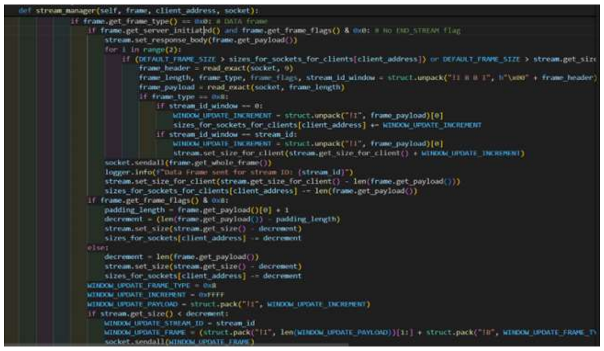
This code processes different types of frames (small packets of data) sent by the client during an HTTP/2 connection. It continuously reads frames and determines their type. For example, it processes data frames (actual client data), headers frames (metadata about the data), and priority frames (information about the importance of streams). It also handles special frames like reset stream frames (to stop a stream), settings frames (to configure communication), ping frames (to check connection health), and goaway frames (to signal the end of the connection). Depending on the frame type, the code performs specific actions like parsing, responding, or logging relevant information. This ensures smooth and structured communication between the client and server.

Stream_manager.py

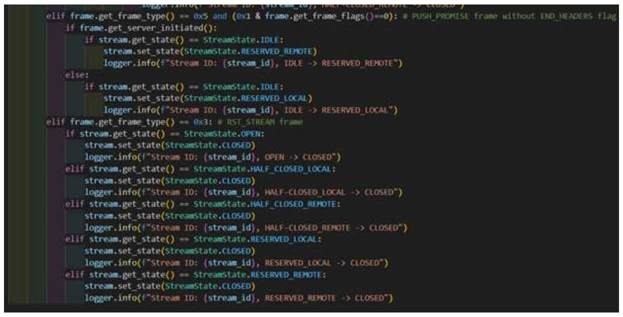
This file is responsible for managing the streams states and also the flow control

The stream manager receives every frame and checks its stream id and if it has new stream id it adds it and if not it manages its state

The other part, which is the flow control it keeps track of window frame of each of connecting and the stream and when the window is full it sends/receive a window update frame



```
def stream_manager(self, frame, client_address, socket):
        if frame.get_frame_type() == 0x0: # DATA frame
            if frame.get_server_initiated() and frame.get_frame_flags() & 0x0: # No END_STREAM flag
                stream.set_response_body(frame.get_payload())
                for i in range(2):
                    if (DEFAULT_FRAME_SIZE > sizes_for_sockets_for_clients[client_address]) or DEFAULT_FRAME_SIZE > stream.get_siz
                        frame_header = read_exact(socket, 9)
                        frame_length, frame_type, frame_flags, stream_id_window = struct.unpack("!I B B I", b"\x00" + frame_header)
                        frame_payload = read_exact(socket, frame_length)
                        if frame_type == 0x8:
                            if stream_id_window == 0:
                                WINDOW_UPDATE_INCREMENT = struct.unpack("!I", frame_payload)[0]
                                sizes_for_sockets_for_clients[client_address] += WINDOW_UPDATE_INCREMENT
                            if stream_id_window == stream_id:
                                WINDOW_UPDATE_INCREMENT = struct.unpack("!I", frame_payload)[0]
                                stream.set_size_for_client(stream.get_size_for_client() + WINDOW_UPDATE_INCREMENT)
                socket.sendall(frame.get_whole_frame())
                logger.info(f"Data Frame sent for stream ID: {stream_id}")
                stream.set_size_for_client(stream.get_size_for_client() - len(frame.get_payload()))
                sizes_for_sockets_for_clients[client_address] -= len(frame.get_payload())
            if frame.get_frame_flags() & 0x8:
                padding_length = frame.get_payload()[0] + 1
                decrement = (len(frame.get_payload()) - padding_length)
                stream.set_size(stream.get_size() - decrement)
                sizes_for_sockets[client_address] -= decrement
            else:
                decrement = len(frame.get_payload())
                stream.set_size(stream.get_size() - decrement)
                sizes_for_sockets[client_address] -= decrement
            WINDOW_UPDATE_FRAME_TYPE = 0x8
            WINDOW_UPDATE_INCREMENT = 0xFFFF
            WINDOW_UPDATE_PAYLOAD = struct.pack("!I", WINDOW_UPDATE_INCREMENT)
            if stream.get_size() < decrement:
                WINDOW_UPDATE_STREAM_ID = stream_id
                WINDOW_UPDATE_FRAME = (struct.pack("!I", len(WINDOW_UPDATE_PAYLOAD))[1:] + struct.pack("!B", WINDOW_UPDATE_FRAME_T
                socket.sendall(WINDOW_UPDATE_FRAME)
```



```
        elif frame.get_frame_type() == 0x5 and (0x1 & frame.get_frame_flags()==0): # PUSH_PROMISE frame without END_HEADERS flag
            if frame.get_server_initiated():
                if stream.get_state() == StreamState.IDLE:
                    stream.set_state(StreamState.RESERVED_REMOTE)
                    logger.info(f"Stream ID: {stream_id}, IDLE -> RESERVED_REMOTE")
            else:
                if stream.get_state() == StreamState.IDLE:
                    stream.set_state(StreamState.RESERVED_LOCAL)
                    logger.info(f"Stream ID: {stream_id}, IDLE -> RESERVED_LOCAL")
        elif frame.get_frame_type() == 0x3: # RST_STREAM frame
            if stream.get_state() == StreamState.OPEN:
                stream.set_state(StreamState.CLOSED)
                logger.info(f"Stream ID: {stream_id}, OPEN -> CLOSED")
            elif stream.get_state() == StreamState.HALF_CLOSED_LOCAL:
                stream.set_state(StreamState.CLOSED)
                logger.info(f"Stream ID: {stream_id}, HALF-CLOSED_LOCAL -> CLOSED")
            elif stream.get_state() == StreamState.HALF_CLOSED_REMOTE:
                stream.set_state(StreamState.CLOSED)
                logger.info(f"Stream ID: {stream_id}, HALF-CLOSED_REMOTE -> CLOSED")
            elif stream.get_state() == StreamState.RESERVED_LOCAL:
                stream.set_state(StreamState.CLOSED)
                logger.info(f"Stream ID: {stream_id}, RESERVED_LOCAL -> CLOSED")
            elif stream.get_state() == StreamState.RESERVED_REMOTE:
                stream.set_state(StreamState.CLOSED)
                logger.info(f"Stream ID: {stream_id}, RESERVED_REMOTE -> CLOSED")
```

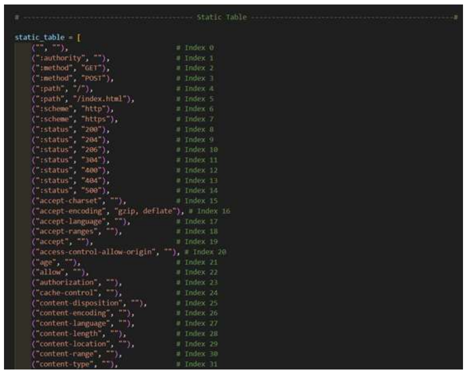
This code manages the handling of data frames in an HTTP/2 stream. When a data frame is received, it checks for specific flags to determine if the frame is complete or part of a larger set. It calculates and updates the size of the data being processed, ensuring it fits within the allowed window size for communication. If the stream reaches its limit, it sends a window update frame to allow more data to be received. Additionally, it processes the payload (content) of the frame and updates the settings for the stream to handle subsequent frames.

If a stream ends or encounters an issue, the code adjusts the flow control by decrementing or resetting window sizes. This ensures efficient and smooth data transfer between the client and server while maintaining proper resource allocation. It handles all these operations dynamically to ensure the connection remains stable and data is processed correctly within the limits set by the HTTP/2 protocol.

HPACK.py

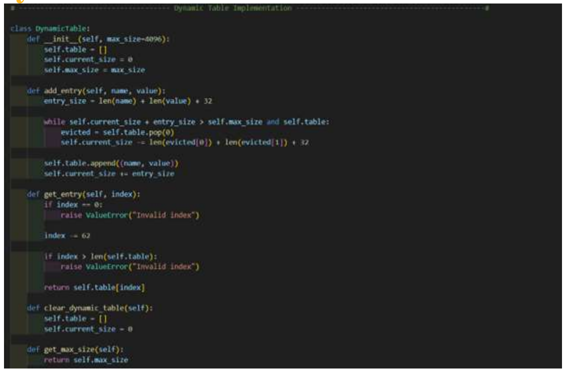
This module is one of the most complex and important modules as it is responsible for receiving the HEADER FRAME as encoded compressed bytes and converting it into readable headers

static table



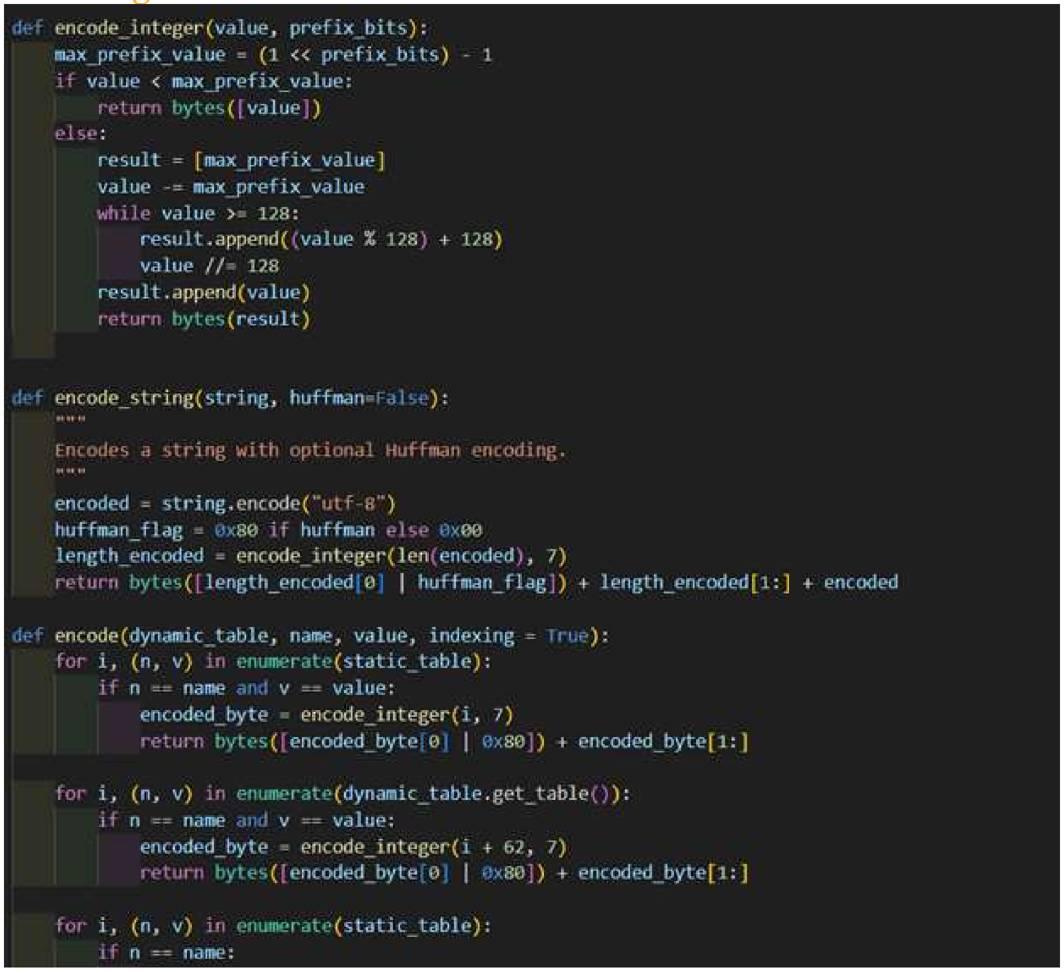
```
# ------------------------------------ Static Table ------------------------------------#

static_table = [
    ("", ""),                           # Index 0
    (":authority", ""),                 # Index 1
    (":method", "GET"),                 # Index 2
    (":method", "POST"),                # Index 3
    (":path", "/"),                     # Index 4
    (":path", "/index.html"),           # Index 5
    (":scheme", "http"),                # Index 6
    (":scheme", "https"),               # Index 7
    (":status", "200"),                 # Index 8
    (":status", "204"),                 # Index 9
    (":status", "206"),                 # Index 10
    (":status", "304"),                 # Index 11
    (":status", "400"),                 # Index 12
    (":status", "404"),                 # Index 13
    (":status", "500"),                 # Index 14
    ("accept-charset", ""),             # Index 15
    ("accept-encoding", "gzip, deflate"), # Index 16
    ("accept-language", ""),            # Index 17
    ("accept-ranges", ""),              # Index 18
    ("accept", ""),                     # Index 19
    ("access-control-allow-origin", ""), # Index 20
    ("age", ""),                        # Index 21
    ("allow", ""),                      # Index 22
    ("authorization", ""),              # Index 23
    ("cache-control", ""),              # Index 24
    ("content-disposition", ""),        # Index 25
    ("content-encoding", ""),           # Index 26
    ("content-language", ""),           # Index 27
    ("content-length", ""),             # Index 28
    ("content-location", ""),           # Index 29
    ("content-range", ""),              # Index 30
    ("content-type", ""),               # Index 31
```

Dynamic table class



```
# ------------------------------ Dynamic Table Implementation ------------------------------#

class DynamicTable:
    def __init__(self, max_size=4096):
        self.table = []
        self.current_size = 0
        self.max_size = max_size

    def add_entry(self, name, value):
        entry_size = len(name) + len(value) + 32

        while self.current_size + entry_size > self.max_size and self.table:
            evicted = self.table.pop(0)
            self.current_size -= len(evicted[0]) + len(evicted[1]) + 32

        self.table.append((name, value))
        self.current_size += entry_size

    def get_entry(self, index):
        if index == 0:
            raise ValueError("Invalid index")

        index -= 62

        if index > len(self.table):
            raise ValueError("Invalid index")

        return self.table[index]

    def clear_dynamic_table(self):
        self.table = []
        self.current_size = 0

    def get_max_size(self):
        return self.max_size
```

Encoding functions



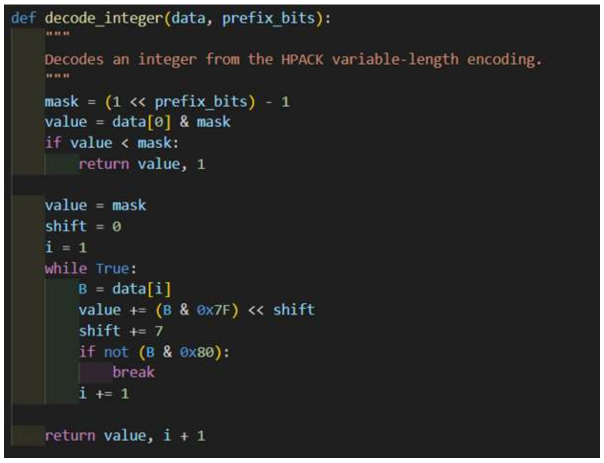
```python
def encode_integer(value, prefix_bits):
    max_prefix_value = (1 << prefix_bits) - 1
    if value < max_prefix_value:
        return bytes([value])
    else:
        result = [max_prefix_value]
        value -= max_prefix_value
        while value >= 128:
            result.append((value % 128) + 128)
            value //= 128
        result.append(value)
        return bytes(result)


def encode_string(string, huffman=False):
    """
    Encodes a string with optional Huffman encoding.
    """
    encoded = string.encode("utf-8")
    huffman_flag = 0x80 if huffman else 0x00
    length_encoded = encode_integer(len(encoded), 7)
    return bytes([length_encoded[0] | huffman_flag]) + length_encoded[1:] + encoded

def encode(dynamic_table, name, value, indexing = True):
    for i, (n, v) in enumerate(static_table):
        if n == name and v == value:
            encoded_byte = encode_integer(i, 7)
            return bytes([encoded_byte[0] | 0x80]) + encoded_byte[1:]

    for i, (n, v) in enumerate(dynamic_table.get_table()):
        if n == name and v == value:
            encoded_byte = encode_integer(i + 62, 7)
            return bytes([encoded_byte[0] | 0x80]) + encoded_byte[1:]

    for i, (n, v) in enumerate(static_table):
        if n == name:
```

This code handles encoding tasks to prepare data for sending in HTTP/2 communication, especially for headers. The first function, encode integer, is used to convert numbers into a format that is compact and efficient. If the number is small, it's directly stored in one byte. For larger numbers, it splits them into multiple parts and stores each part in a sequence of bytes, ensuring the encoding works well for all sizes of numbers.

The second function, encode string, takes a text (string) and prepares it for sending. It can also apply a technique called Huffman encoding, which compresses the text to save space. The function converts the text into bytes, adds the length of the text in a special format at the beginning, and returns everything as a compact package ready to send.

The third function, encode, is for encoding HTTP headers (like names and values). It looks for matches in two tables: one for common headers that are predefined (static) and another for headers that are added during communication (dynamic). If a match is found, it reuses the information, saving space. If not, it creates a new entry and encodes the header efficiently. This process helps reduce the size of HTTP/2 headers, making communication faster and more efficient.

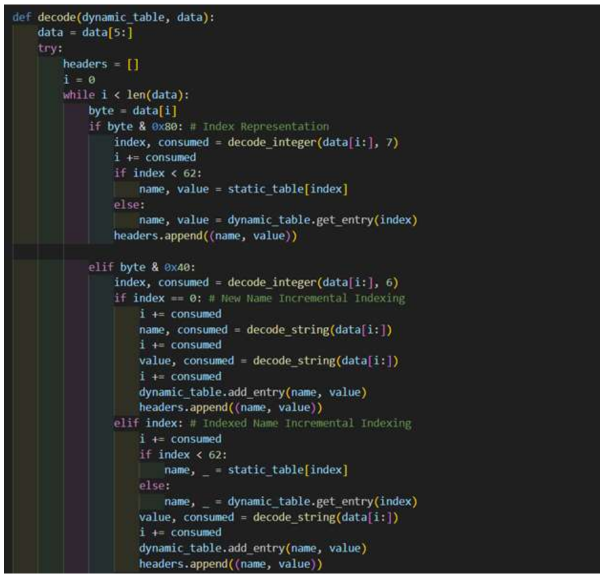
Decoding functions



```python
def decode_integer(data, prefix_bits):
    """
    Decodes an integer from the HPACK variable-length encoding.
    """
    mask = (1 << prefix_bits) - 1
    value = data[0] & mask
    if value < mask:
        return value, 1

    value = mask
    shift = 0
    i = 1
    while True:
        B = data[i]
        value += (B & 0x7F) << shift
        shift += 7
        if not (B & 0x80):
            break
        i += 1

    return value, i + 1
```

This code is responsible for decoding numbers encoded in a special format used in HTTP/2 headers called HPACK. It starts by reading the first byte of data and extracts the number using a prefix (a certain number of bits). If the number is small enough to fit in the prefix, the function simply returns the number along with the count of bytes read (which is just one in this case).

If the number is too large for the prefix, the function keeps reading additional bytes one at a time. It adds each part of the number to the total, adjusting for its position using shifts. The loop continues until it reaches a byte that indicates the end of the number. Finally, the function returns the complete decoded number and the total number of bytes it used to decode it. This approach ensures that even very large numbers can be decoded efficiently.



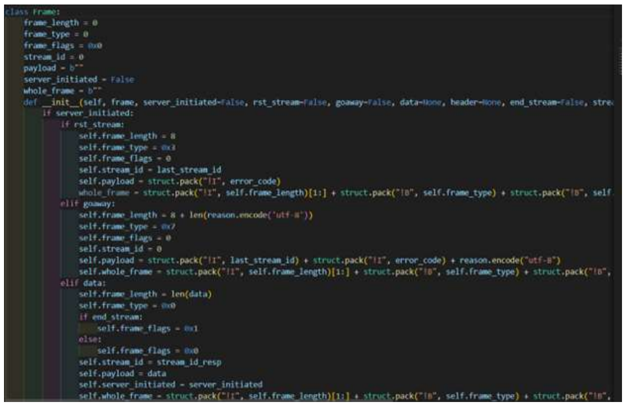
```python
def decode(dynamic_table, data):
    data = data[5:]
    try:
        headers = []
        i = 0
        while i < len(data):
            byte = data[i]
            if byte & 0x80: # Index Representation
                index, consumed = decode_integer(data[i:], 7)
                i += consumed
                if index < 62:
                    name, value = static_table[index]
                else:
                    name, value = dynamic_table.get_entry(index)
                headers.append((name, value))

            elif byte & 0x40:
                index, consumed = decode_integer(data[i:], 6)
                if index == 0: # New Name Incremental Indexing
                    i += consumed
                    name, consumed = decode_string(data[i:])
                    i += consumed
                    value, consumed = decode_string(data[i:])
                    i += consumed
                    dynamic_table.add_entry(name, value)
                    headers.append((name, value))
                elif index: # Indexed Name Incremental Indexing
                    i += consumed
                    if index < 62:
                        name, _ = static_table[index]
                    else:
                        name, _ = dynamic_table.get_entry(index)
                    value, consumed = decode_string(data[i:])
                    i += consumed
                    dynamic_table.add_entry(name, value)
                    headers.append((name, value))
```

This code is responsible for decoding HTTP/2 headers from a compressed format using the HPACK standard. It processes a block of data and extracts header fields (name-value pairs). It starts by reading each byte to determine how the header field is represented, either by referencing an index in a predefined table (static table) or a dynamically updated table. If the header field uses an index, the code retrieves the corresponding name and value directly from the appropriate table.

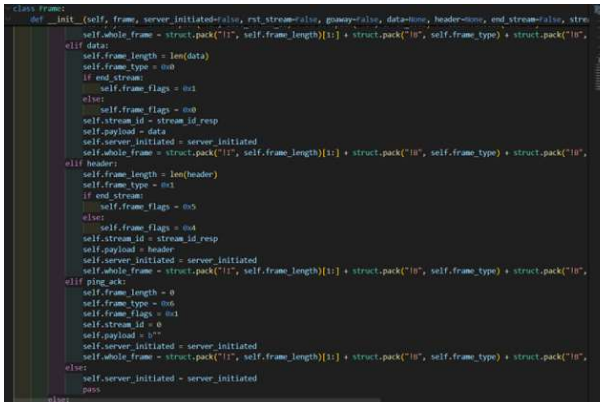
If the header field is not indexed, the code reads the name and value as strings, updates the dynamic table with this new information, and stores the decoded header field. The loop continues until all header fields in the data block are decoded. This ensures efficient handling of HTTP/2 headers by reusing common headers through indexing while allowing new headers to be dynamically added and processed.

Frames.py

Class frame is made managing the frames properly



```
class Frame:
    frame_length = 0
    frame_type = 0
    frame_flags = 0x0
    stream_id = 0
    payload = b""
    server_initiated = False
    whole_frame = b""
    def __init__(self, frame, server_initiated=False, rst_stream=False, goaway=False, data=None, header=None, end_stream=False, stre
        if server_initiated:
            if rst_stream:
                self.frame_length = 8
                self.frame_type = 0x3
                self.frame_flags = 0
                self.stream_id = last_stream_id
                self.payload = struct.pack("!I", error_code)
                whole_frame = struct.pack("!I", self.frame_length)[1:] + struct.pack("!B", self.frame_type) + struct.pack("!B", self
            elif goaway:
                self.frame_length = 8 + len(reason.encode('utf-8'))
                self.frame_type = 0x7
                self.frame_flags = 0
                self.stream_id = 0
                self.payload = struct.pack("!I", last_stream_id) + struct.pack("!I", error_code) + reason.encode("utf-8")
                self.whole_frame = struct.pack("!I", self.frame_length)[1:] + struct.pack("!B", self.frame_type) + struct.pack("!B",
            elif data:
                self.frame_length = len(data)
                self.frame_type = 0x0
                if end_stream:
                    self.frame_flags = 0x1
                else:
                    self.frame_flags = 0x0
                self.stream_id = stream_id_resp
                self.payload = data
                self.server_initiated = server_initiated
                self.whole_frame = struct.pack("!I", self.frame_length)[1:] + struct.pack("!B", self.frame_type) + struct.pack("!B",
```



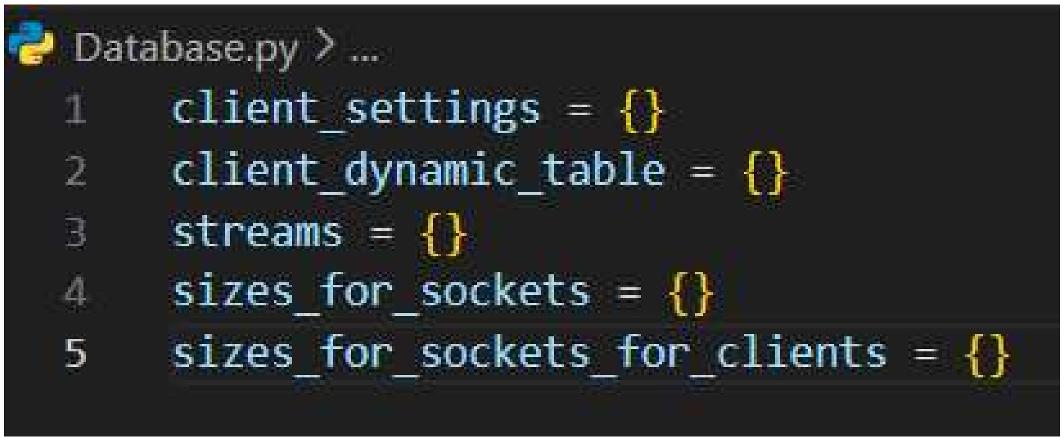
```
class Frame:
    def __init__(self, frame, server_initiated=False, rst_stream=False, goaway=False, data=None, header=None, end_stream=False, stre
                self.whole_frame = struct.pack("!I", self.frame_length)[1:] + struct.pack("!B", self.frame_type) + struct.pack("!B",
            elif data:
                self.frame_length = len(data)
                self.frame_type = 0x0
                if end_stream:
                    self.frame_flags = 0x1
                else:
                    self.frame_flags = 0x0
                self.stream_id = stream_id_resp
                self.payload = data
                self.server_initiated = server_initiated
                self.whole_frame = struct.pack("!I", self.frame_length)[1:] + struct.pack("!B", self.frame_type) + struct.pack("!B",
            elif header:
                self.frame_length = len(header)
                self.frame_type = 0x1
                if end_stream:
                    self.frame_flags = 0x5
                else:
                    self.frame_flags = 0x4
                self.stream_id = stream_id_resp
                self.payload = header
                self.server_initiated = server_initiated
                self.whole_frame = struct.pack("!I", self.frame_length)[1:] + struct.pack("!B", self.frame_type) + struct.pack("!B",
            elif ping_ack:
                self.frame_length = 0
                self.frame_type = 0x6
                self.frame_flags = 0x1
                self.stream_id = 0
                self.payload = b""
                self.server_initiated = server_initiated
                self.whole_frame = struct.pack("!I", self.frame_length)[1:] + struct.pack("!B", self.frame_type) + struct.pack("!B",
            else:
                self.server_initiated = server_initiated
                pass
        else:
```

This code defines a Frame class that creates and organizes different types of HTTP/2 frames, which are small packets of data exchanged between the client and server. Frames can serve various purposes, such as carrying data, headers, control signals, or managing streams. The __init__ method initializes these frames based on their specific roles, like resetting streams, sending data, signaling the end of communication, or acknowledging pings.

For each frame type, the code calculates the frame's length, sets its type and flags, and constructs the payload with all the necessary information. For example, a data frame includes the actual data being transmitted, while a headers frame includes metadata about the request or response. The constructed frame is then packed into a binary format (whole_frame), which can be sent over the network. This ensures that frames are properly structured and compatible with the HTTP/2 protocol.

Database.py

This is made for storing important data about the clients and their connections

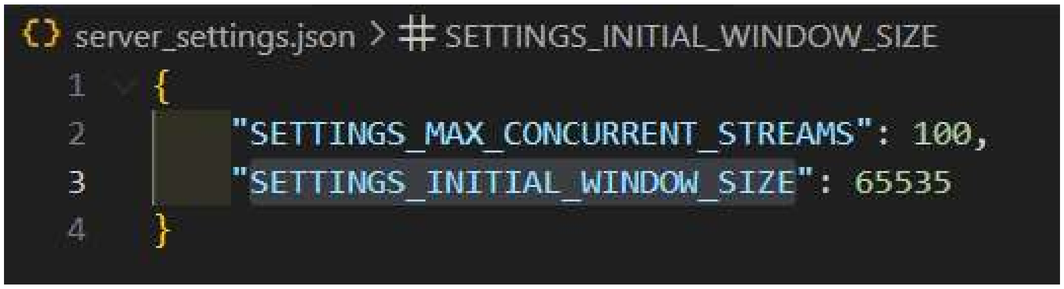


Database.py > ...

```
client_settings = {}
client_dynamic_table = {}
streams = {}
sizes_for_sockets = {}
sizes_for_sockets_for_clients = {}
```

Server_settings.json

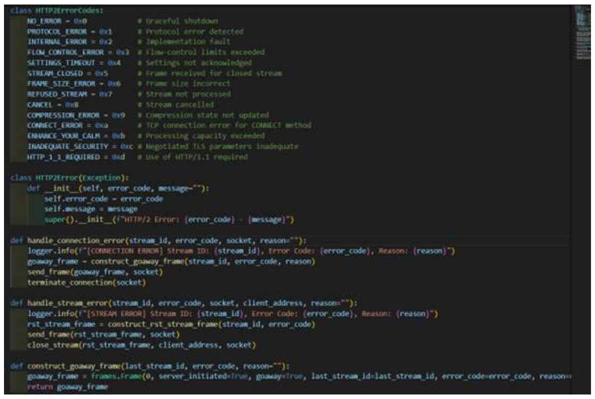
This made for saving the server settings that will be sent while exchanging the SETTINGS FRAME



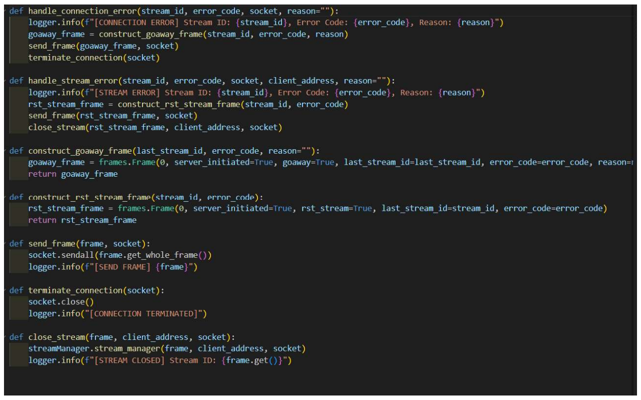
```
server_settings.json > # SETTINGS_INITIAL_WINDOW_SIZE
{
    "SETTINGS_MAX_CONCURRENT_STREAMS": 100,
    "SETTINGS_INITIAL_WINDOW_SIZE": 65535
}
```

Error_handling.py

Provides essential functions for managing HTTP/2 connection and stream errors, constructing appropriate error frames, and ensuring that these frames are sent to the client, followed by proper connection or stream termination.



```python
class HTTP2ErrorCodes:
    NO_ERROR = 0x0              # Graceful shutdown
    PROTOCOL_ERROR = 0x1        # Protocol error detected
    INTERNAL_ERROR = 0x2        # Implementation fault
    FLOW_CONTROL_ERROR = 0x3    # Flow-control limits exceeded
    SETTINGS_TIMEOUT = 0x4      # Settings not acknowledged
    STREAM_CLOSED = 0x5         # Frame received for closed stream
    FRAME_SIZE_ERROR = 0x6      # Frame size incorrect
    REFUSED_STREAM = 0x7        # Stream not processed
    CANCEL = 0x8                # Stream cancelled
    COMPRESSION_ERROR = 0x9     # Compression state not updated
    CONNECT_ERROR = 0xa         # TCP connection error for CONNECT method
    ENHANCE_YOUR_CALM = 0xb     # Processing capacity exceeded
    INADEQUATE_SECURITY = 0xc # Negotiated TLS parameters inadequate
    HTTP_1_1_REQUIRED = 0xd     # Use of HTTP/1.1 required

class HTTP2Error(Exception):
    def __init__(self, error_code, message=""):
        self.error_code = error_code
        self.message = message
        super().__init__(f"HTTP/2 Error: {error_code} - {message}")

def handle_connection_error(stream_id, error_code, socket, reason=""):
    logger.info(f"[CONNECTION ERROR] Stream ID: {stream_id}, Error Code: {error_code}, Reason: {reason}")
    goaway_frame = construct_goaway_frame(stream_id, error_code, reason)
    send_frame(goaway_frame, socket)
    terminate_connection(socket)

def handle_stream_error(stream_id, error_code, socket, client_address, reason=""):
    logger.info(f"[STREAM ERROR] Stream ID: {stream_id}, Error Code: {error_code}, Reason: {reason}")
    rst_stream_frame = construct_rst_stream_frame(stream_id, error_code)
    send_frame(rst_stream_frame, socket)
    close_stream(rst_stream_frame, client_address, socket)

def construct_goaway_frame(last_stream_id, error_code, reason=""):
    goaway_frame = frames.Frame(0, server_initiated=True, goaway=True, last_stream_id=last_stream_id, error_code=error_code, reason=
    return goaway_frame
```



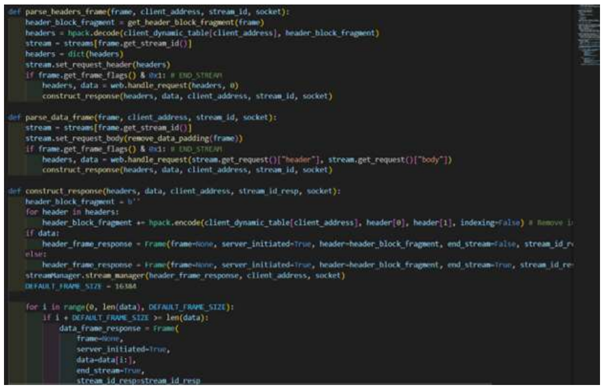
```python
def handle_connection_error(stream_id, error_code, socket, reason=""):
    logger.info(f"[CONNECTION ERROR] Stream ID: {stream_id}, Error Code: {error_code}, Reason: {reason}")
    goaway_frame = construct_goaway_frame(stream_id, error_code, reason)
    send_frame(goaway_frame, socket)
    terminate_connection(socket)

def handle_stream_error(stream_id, error_code, socket, client_address, reason=""):
    logger.info(f"[STREAM ERROR] Stream ID: {stream_id}, Error Code: {error_code}, Reason: {reason}")
    rst_stream_frame = construct_rst_stream_frame(stream_id, error_code)
    send_frame(rst_stream_frame, socket)
    close_stream(rst_stream_frame, client_address, socket)

def construct_goaway_frame(last_stream_id, error_code, reason=""):
    goaway_frame = frames.Frame(0, server_initiated=True, goaway=True, last_stream_id=last_stream_id, error_code=error_code, reason=r
    return goaway_frame

def construct_rst_stream_frame(stream_id, error_code):
    rst_stream_frame = frames.Frame(0, server_initiated=True, rst_stream=True, last_stream_id=stream_id, error_code=error_code)
    return rst_stream_frame

def send_frame(frame, socket):
    socket.sendall(frame.get_whole_frame())
    logger.info(f"[SEND FRAME] {frame}")

def terminate_connection(socket):
    socket.close()
    logger.info("[CONNECTION TERMINATED]")

def close_stream(frame, client_address, socket):
    streamManager.stream_manager(frame, client_address, socket)
    logger.info(f"[STREAM CLOSED] Stream ID: {frame.get()}")
```

This code is responsible for handling HTTP/2 errors and managing connections or streams when something goes wrong. The HTTP2ErrorCodes class defines various error codes like protocol errors, flow control issues, and connection failures, each representing specific problems in the HTTP/2 communication. The HTTP2Error class extends Python's Exception to create custom error messages with these codes.

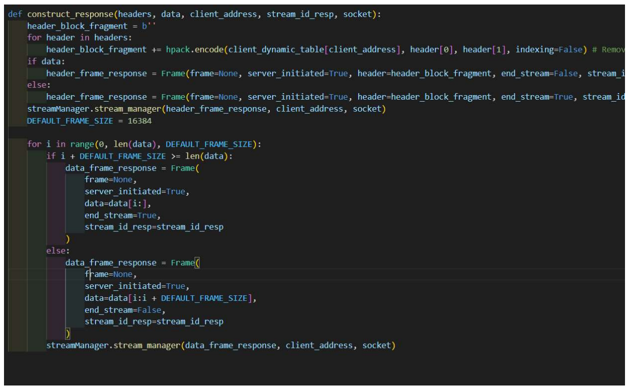
Functions like handle_connection_error and handle_stream_error manage errors by logging the issue and constructing appropriate frames (like "goaway" or "reset stream" frames) to notify the client about the problem. These frames are sent to the client, and if necessary, the connection or stream is terminated. Helper functions like construct_goaway_frame and construct_rst_stream_frame build the specific frames needed for handling these errors, ensuring the server follows the HTTP/2 protocol while managing issues efficiently. This code ensures graceful error handling and proper cleanup when errors occur.

parsing_header_data.py

processes HTTP/2 frames, extracts and handles request headers and bodies, and constructs appropriate HTTP/2 response frames. It includes functions to parse data frames, handle end-of-stream conditions, and manage the construction and sending of response frames. The file ensures proper encoding of headers and segmentation of data for transmission



```python
def parse_headers_frame(frame, client_address, stream_id, socket):
    header_block_fragment = get_header_block_fragment(frame)
    headers = hpack.decode(client_dynamic_table[client_address], header_block_fragment)
    stream = streams[frame.get_stream_id()]
    headers = dict(headers)
    stream.set_request_header(headers)
    if frame.get_frame_flags() & 0x1: # END_STREAM
        headers, data = web.handle_request(headers, 0)
        construct_response(headers, data, client_address, stream_id, socket)

def parse_data_frame(frame, client_address, stream_id, socket):
    stream = streams[frame.get_stream_id()]
    stream.set_request_body(remove_data_padding(frame))
    if frame.get_frame_flags() & 0x1: # END_STREAM
        headers, data = web.handle_request(stream.get_request()["header"], stream.get_request()["body"])
        construct_response(headers, data, client_address, stream_id, socket)

def construct_response(headers, data, client_address, stream_id_resp, socket):
    header_block_fragment = b''
    for header in headers:
        header_block_fragment += hpack.encode(client_dynamic_table[client_address], header[0], header[1], indexing=False) # Remove i
    if data:
        header_frame_response = Frame(frame=None, server_initiated=True, header=header_block_fragment, end_stream=False, stream_id_r
    else:
        header_frame_response = Frame(frame=None, server_initiated=True, header=header_block_fragment, end_stream=True, stream_id_re
    streamManager.stream_manager(header_frame_response, client_address, socket)
    DEFAULT_FRAME_SIZE = 16384

    for i in range(0, len(data), DEFAULT_FRAME_SIZE):
        if i + DEFAULT_FRAME_SIZE >= len(data):
            data_frame_response = Frame(
                frame=None,
                server_initiated=True,
                data=data[i:],
                end_stream=True,
                stream_id_resp=stream_id_resp
```



```python
def construct_response(headers, data, client_address, stream_id_resp, socket):
    header_block_fragment = b''
    for header in headers:
        header_block_fragment += hpack.encode(client_dynamic_table[client_address], header[0], header[1], indexing=False) # Remov
    if data:
        header_frame_response = Frame(frame=None, server_initiated=True, header=header_block_fragment, end_stream=False, stream_i
    else:
        header_frame_response = Frame(frame=None, server_initiated=True, header=header_block_fragment, end_stream=True, stream_id
    streamManager.stream_manager(header_frame_response, client_address, socket)
    DEFAULT_FRAME_SIZE = 16384

    for i in range(0, len(data), DEFAULT_FRAME_SIZE):
        if i + DEFAULT_FRAME_SIZE >= len(data):
            data_frame_response = Frame(
                frame=None,
                server_initiated=True,
                data=data[i:],
                end_stream=True,
                stream_id_resp=stream_id_resp
            )
        else:
            data_frame_response = Frame(
                frame=None,
                server_initiated=True,
                data=data[i:i + DEFAULT_FRAME_SIZE],
                end_stream=False,
                stream_id_resp=stream_id_resp
            )
        streamManager.stream_manager(data_frame_response, client_address, socket)
```

This code is responsible for processing and responding to HTTP/2 frames, specifically headers and data frames. The parse_headers_frame function extracts the header block from a frame, decodes it, and processes the headers to handle requests. If the frame marks the end of a stream, it constructs a response using the headers and sends it back to the client. Similarly, the parse_data_frame function handles frames carrying data. It removes any padding, processes the data, and constructs a response if the frame signals the end of the stream.

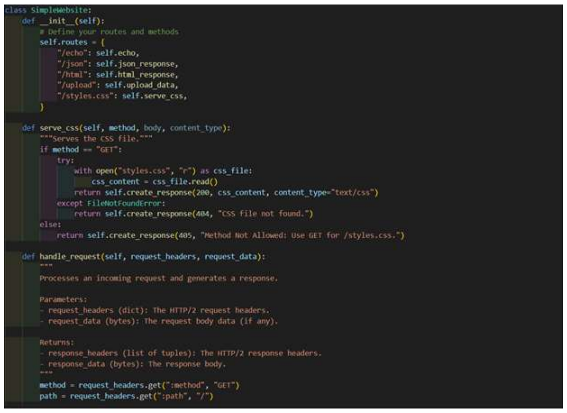
The construct_response function creates the response frames to send back to the client. It encodes the headers into a format suitable for HTTP/2 communication and constructs one or more data frames if there is a body to send. The function ensures the data is split into smaller frames if needed, following size limits, and marks the final frame as the end of the stream. This ensures the server can handle and respond to client requests efficiently while adhering to HTTP/2 protocol rules.

HOW TO HOST YOUR WEBSITE

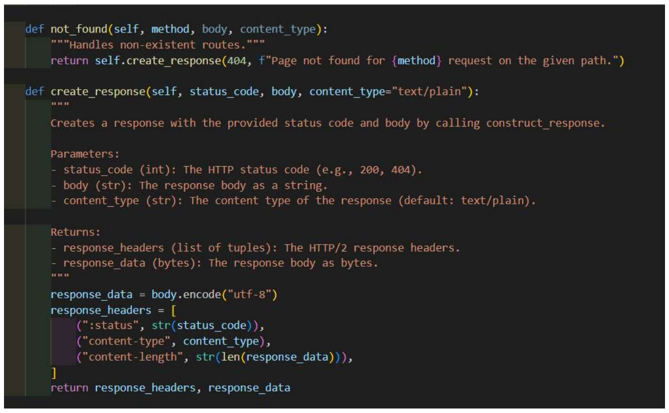
The parsing_header_data.py file processes HTTP/2 frames, extracts request headers and bodies, and constructs response frames.

The SimpleWebsite class in website.py handles web requests and generates responses.

The create_response function in SimpleWebsite creates HTTP/2 response headers and body, which are used by parsing_header_data.py to construct and send responses.



```python
class SimpleWebsite:
    def __init__(self):
        # Define your routes and methods
        self.routes = {
            "/echo": self.echo,
            "/json": self.json_response,
            "/html": self.html_response,
            "/upload": self.upload_data,
            "/styles.css": self.serve_css,
        }

    def serve_css(self, method, body, content_type):
        """Serves the CSS file."""
        if method == "GET":
            try:
                with open("styles.css", "r") as css_file:
                    css_content = css_file.read()
                return self.create_response(200, css_content, content_type="text/css")
            except FileNotFoundError:
                return self.create_response(404, "CSS file not found.")
        else:
            return self.create_response(405, "Method Not Allowed: Use GET for /styles.css.")

    def handle_request(self, request_headers, request_data):
        """
        Processes an incoming request and generates a response.

        Parameters:
        - request_headers (dict): The HTTP/2 request headers.
        - request_data (bytes): The request body data (if any).

        Returns:
        - response_headers (list of tuples): The HTTP/2 response headers.
        - response_data (bytes): The response body.
        """
        method = request_headers.get(":method", "GET")
        path = request_headers.get(":path", "/")
```

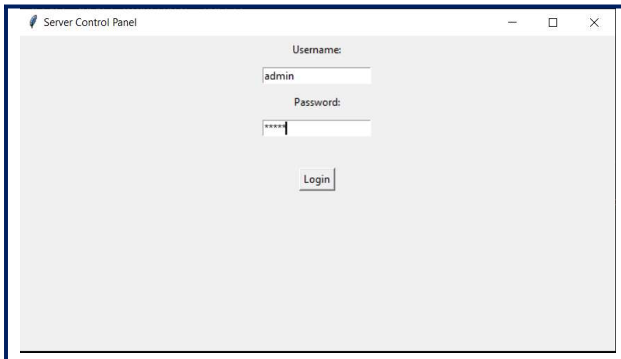


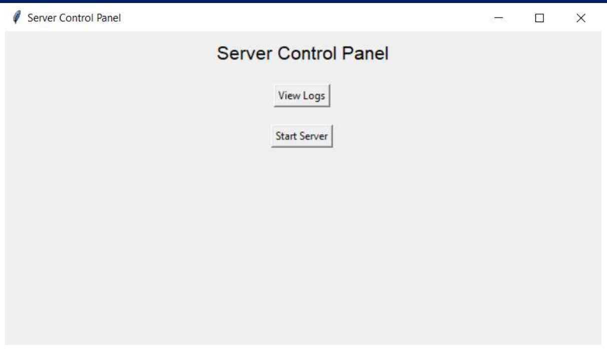
```python
    def not_found(self, method, body, content_type):
        """Handles non-existent routes."""
        return self.create_response(404, f"Page not found for {method} request on the given path.")

    def create_response(self, status_code, body, content_type="text/plain"):
        """
        Creates a response with the provided status code and body by calling construct_response.

        Parameters:
        - status_code (int): The HTTP status code (e.g., 200, 404).
        - body (str): The response body as a string.
        - content_type (str): The content type of the response (default: text/plain).

        Returns:
        - response_headers (list of tuples): The HTTP/2 response headers.
        - response_data (bytes): The response body as bytes.
        """
        response_data = body.encode("utf-8")
        response_headers = [
            (":status", str(status_code)),
            ("content-type", content_type),
            ("content-length", str(len(response_data))),
        ]
        return response_headers, response_data
```

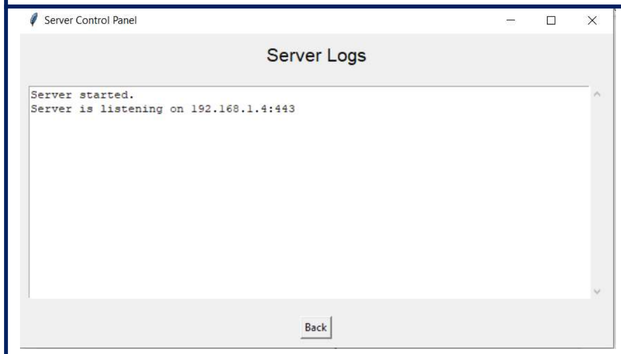
This code is part of a simple web server class (SimpleWebsite) that handles HTTP/2 requests and responses. It defines different routes like /echo, /json, /html, and /styles.css, and each route is linked to a specific method. For example, the serve_css method handles requests to /styles.css, serving a CSS file if the request method is GET. If the file is not found, it returns a 404 error; if the method is incorrect, it returns a 405 error.

The handle_request method processes incoming requests by matching the requested path and method to the defined routes. If a route is not found, the not_found method generates a 404 response. The create_response method helps construct HTTP/2 responses with a status code (like 200 or 404), a response body, and content type. It ensures all responses are correctly formatted for HTTP/2 by including necessary headers like status, content type, and content length. This structure makes it easy to extend the server with additional functionality.

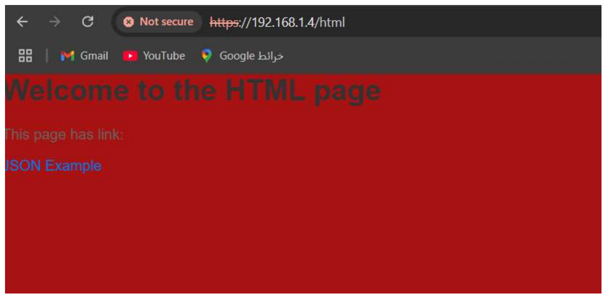
TESTING USING BROWSER







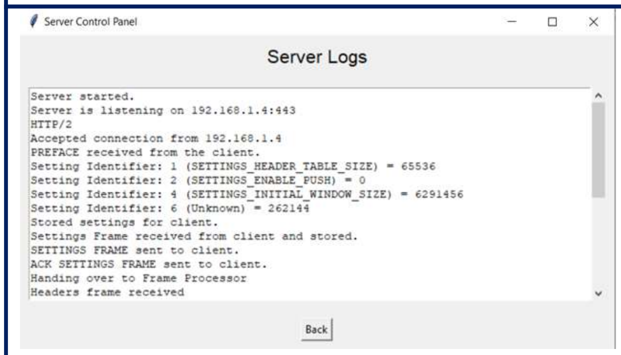
```
Server started.
Server is listening on 192.168.1.4:443
```



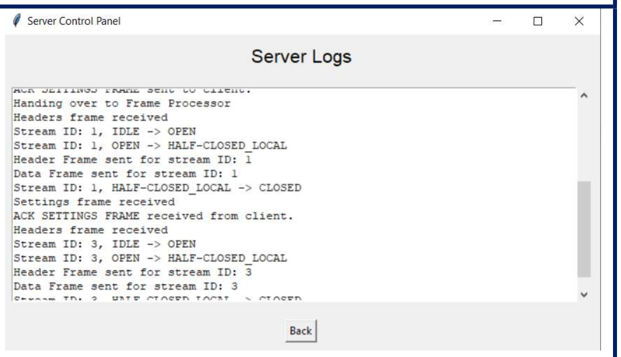
Welcome to the HTML page

This page has link:

JSON Example



```
Server started.
Server is listening on 192.168.1.4:443
HTTP/2
Accepted connection from 192.168.1.4
PREFACE received from the client.
Setting Identifier: 1 (SETTINGS_HEADER_TABLE_SIZE) = 65536
Setting Identifier: 2 (SETTINGS_ENABLE_PUSH) = 0
Setting Identifier: 4 (SETTINGS_INITIAL_WINDOW_SIZE) = 6291456
Setting Identifier: 6 (Unknown) = 262144
Stored settings for client.
Settings Frame received from client and stored.
SETTINGS FRAME sent to client.
ACK SETTINGS FRAME sent to client.
Handing over to Frame Processor
Headers frame received
```



```
Handing over to Frame Processor
Headers frame received
Stream ID: 1, IDLE -> OPEN
Stream ID: 1, OPEN -> HALF-CLOSED_LOCAL
Header Frame sent for stream ID: 1
Data Frame sent for stream ID: 1
Stream ID: 1, HALF-CLOSED_LOCAL -> CLOSED
Settings frame received
ACK SETTINGS FRAME received from client.
Headers frame received
Stream ID: 3, IDLE -> OPEN
Stream ID: 3, OPEN -> HALF-CLOSED_LOCAL
Header Frame sent for stream ID: 3
Data Frame sent for stream ID: 3
```

The previous code successfully tested the web browser by creating an HTTP/2 server that handled client requests and served responses based on specific routes and methods. The server processed requests for HTML pages, JSON responses, and CSS files while logging every interaction to ensure proper functionality. These logs provided valuable insights into the server's ability to manage requests and communicate effectively with the client.

In the first step, the login screen is displayed, requiring the administrator to enter a username and password. This ensures that only authorized users can access the server control panel. Once logged in, the user is directed to the control panel, where they can view logs or start the server, providing easy management of server activities.

After starting the server, it begins listening on a specified IP address and port. The server logs confirm this, showing that it is ready to handle incoming connections. Next, a client interacts with the server using a browser, sending a request for an HTML page. The server processes the request and serves the requested HTML content, which is displayed in the browser.

The server logs show detailed information about each interaction, including the headers and data received, the requests processed, and the responses sent back to the client. The logs also indicate the proper handling of various HTTP/2 frames and the completion of operations, confirming the server's functionality across multiple requests.