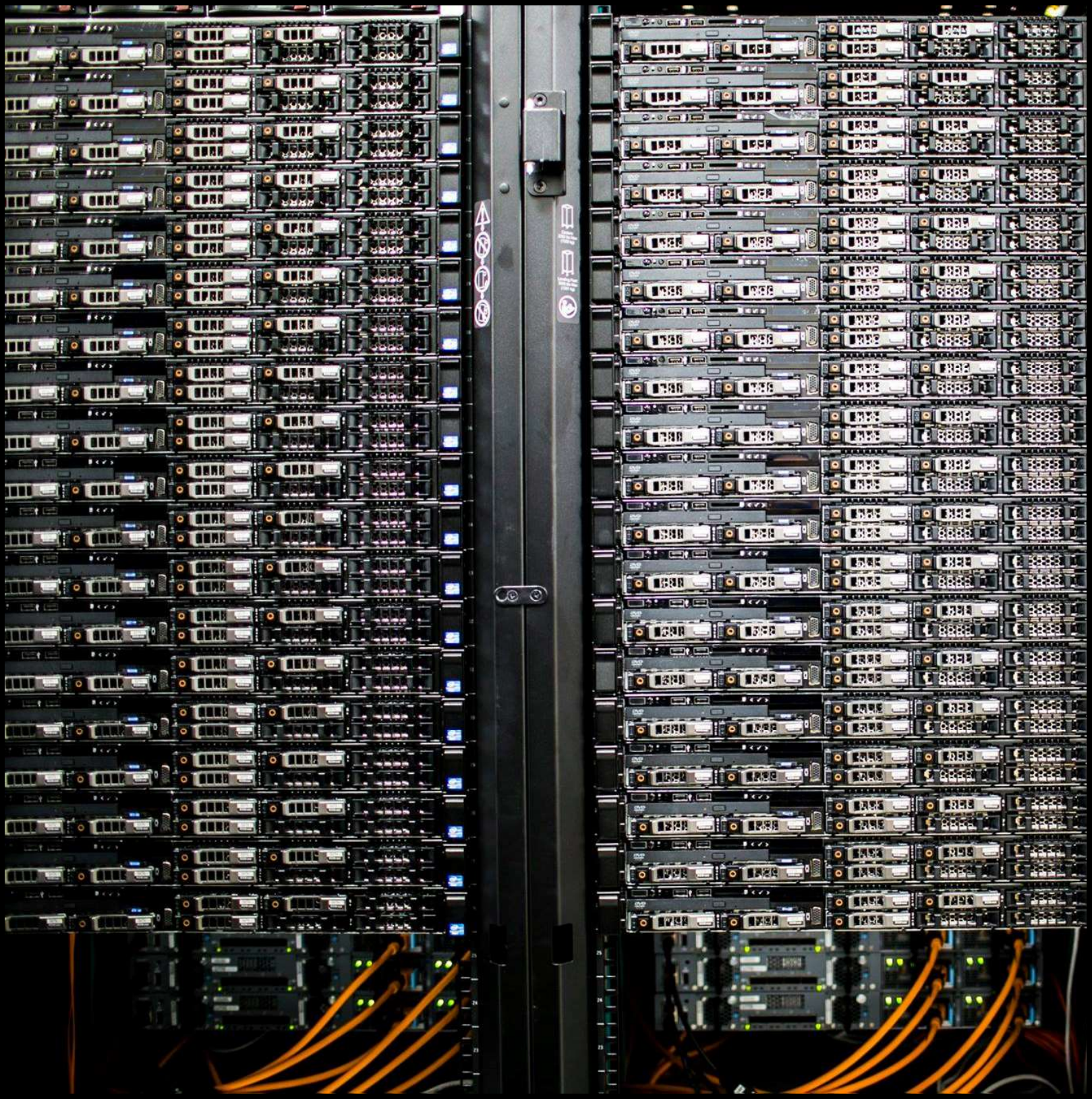
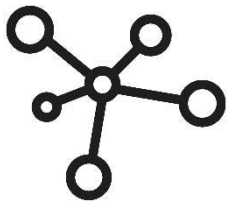


HTTP2 Server

Phase III



Computer Networks

Ain Shams University

Faculty of Engineering

CESS

CSE351 Computer Networks

Fall 24

Proposal Document

Proposal Done by:

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

Table of Contents

CONNECTION_HANDLER.PY

```python
def handle_client_thread(client_socket, client_address):
    handle_client_connection(client_socket, client_address)

def start_server(host="192.168.1.10", port=80):
    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)
    print(f"Server is listening on {host}:{port}")

    try:
        while True:
            client_socket, client_address = server_socket.accept()
            sizes_for_sockets[client_address] = 0
            client_address = client_address[0]
            print(f"Accepted connection from {client_address}")
            client_dynamic_table[client_address] = {}
            client_thread = threading.Thread(target=handle_client_thread, args=(client_socket, client_address))
            client_thread.start()
    except KeyboardInterrupt:
        print("Shutting down server...")
    finally:
        server_socket.close()

if __name__ == "__main__":
    start_server()
```

This code implements a simple multithreaded server in Python. By default, "192.168.1.10" and port 80 are the host and port on which the server listens for incoming client connections. It creates a TCP/IP connection using the socket library and allows for numerous concurrent connections by starting a new thread for every client.

The key component of the server is the start_server function. It begins by creating a socket using the AF_INET (IPv4) and SOCK_STREAM (TCP) protocols. The setsockopt function is called to enable the SO_REUSEADDR option, which allows the socket to reuse the address if the server is restarted. The socket is then bound to the specified host and port, and the server starts listening for connections with a maximum backlog of 10.

The server enters an infinite loop, where it waits for incoming client connections. When a client connects, the server accepts the connection, creating a new socket for communication with the client and retrieving the client's address. Two data structures, sizes_for_sockets and client_dynamic_table, are initialized to manage client-specific information.

Each client is then handled in a separate thread, which is created using the threading module. The thread executes the handle_client_thread function, passing the client socket and address as arguments. This setup ensures the server can handle multiple clients simultaneously. To ensure graceful shutdowns, the code includes exception handling. If a KeyboardInterrupt (e.g., Ctrl+C) is triggered, the server prints a shutdown message and exits the loop. The finally block ensures that the server socket is properly closed, releasing system resources.

For managing client connections, the handle_client_thread function acts as a stand-in. It invokes handle_client_connection, a different method that isn't specified in the given code snippet. The logic for handling client requests, such reading and sending data or carrying out particular activities, is probably contained in this undefined function.

Finally the block at the end of the script makes sure that the server only begins when the script is run directly. The server starts up when this block runs the start_server method. All things considered, this code offers a basic framework for creating a multithreaded server; however, the precise functionality and behavior rely on how the handle_client_connection function is implemented.

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

```
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:
            print("Invalid HTTP/2 preface received. Closing connection.")
            client_socket.close()
            return
        print("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)

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

    except Exception as e:
        print(f"Error: {e}")
        client_socket.close()
    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]
        client_socket.close()
        print(f"Connection with {client_address} closed and its settings deleted.")
```

The `handle_client_connection` function, defined in this code, handles a client's connection to the server. It handles client-sent frames, controls the communication protocol, and makes sure the connection is closed correctly.

The function begins by initializing the client's settings dictionary (`client_settings`) upon connection. It attempts to read the client's preface, which is an initial message expected to match a predefined constant (`HTTP2_PREFACE`). If the preface is invalid, the connection is closed. Next, the function reads a frame from the client, checking its type. If the frame type matches a specified constant (`SETTINGS_FRAME_TYPE`), additional frame payload data is received and handled by the `settings_frame_handler`.

A frame_processor, which probably handles other kinds of frames or client requests, receives additional processing from the function after handling frames related to settings. To detect and record connection errors, the code incorporates exception handling. In the finally block, all client-related data structures are cleaned up, and the client socket is closed. This ensures that the connection is terminated gracefully, and the server resources associated with the client are freed.

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:
        print("Invalid stream ID in settings frame. Closing connection.")
        return

    if frame.get_frame_flags() == SETTINGS_ACK_FLAG:
        print("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)

    print("Settings Frame received from client and stored.")

    send_settings_frame(client_socket)

    send_server_ack_settings_frame(client_socket)
```

The `settings_frame_handler` function, which handles client-sent settings frames, includes this code. The function makes sure that the client and server are communicating and exchanging configurations correctly.

The function first determines whether the stream ID of the frame is legitimate. The settings frame is deemed invalid if a non-zero value is returned by the `get_stream_id()` method. To stop additional processing in these situations, it logs an error message and ends the function.

The method then determines whether the acknowledgment flag (`SETTINGS_ACK_FLAG`) is set in the frame. If this flag is present, it logs a message acknowledging receipt of the client's settings acknowledgment and exits the function, as no further processing is required for acknowledgment frames.

The function uses the decode_settings_frame method to decode the frame's payload if the frame passes these criteria. In this stage, the client's settings are extracted and interpreted. After that, it makes sure the server updates its internal data structures with the client's configuration by storing these settings using the store_client_settings method.

The function logs that the settings frame has been received and stored after processing it successfully. This attests to the server handling the client's settings correctly.

In response, the server uses the send_settings_frame method to send the client its own settings frame. After that, it completes the communication and verifies that the client's settings were received by sending a server acknowledgment frame using the send_server_ack_settings_frame method. This guarantees that the configurations on both sides are in sync.

FRAME_PROCESSOR.PY:

Frame processor should handle the frames coming from the client and send it where it belongs

But every before that it should go to the stream manager first and we will see that later

```python
def frame_processor(client_socket, client_address):
    try:
        while True:
            frame = frames.Frame(read_exact(client_socket, 9))
            frame.set_payload(read_exact(client_socket, frame.get_frame_length()))
            streamManager.stream_manager(frame, client_address, client_socket)

            if frame.get_frame_type() == 0x0:  # DATA frame
                pass
            elif frame.get_frame_type() == 0x1:  # HEADERS frame
                client_dynamic_table[client_address] = hpack.DynamicTable()
                headers = hpack.decode(client_dynamic_table[client_address], frame.get_payload())
                print(headers)
            elif frame.get_frame_type() == 0x4:  # SETTINGS frame
                settings_frame_handler(client_socket, client_address, frame)
            elif frame.get_frame_type() == 0x8:  # WINDOW_UPDATE frame
                #flow_control(stream_id)
                pass
            elif frame.get_frame_type() == 0x3:  # RST_STREAM frame
                #handle_rst_stream_frame(stream_id)
                pass
            elif frame.get_frame_type() == 0x7:  # GOAWAY frame
                #handle_goaway_frame(client_socket)
                pass
            elif frame.get_frame_type() == 0x2:  # PRIORITY frame
                #handle_priority_frame(stream_id)
                pass
            else:
                print(f"Unknown frame type {frame.get_frame_type()} received. Ignoring.")
```

The frame_processor function, defined by this code, processes client-provided frames continuously. It makes sure the right steps are completed and handles various frame kinds according to their type identifier.

In order to continue processing frames while the connection is active, the function initiates an infinite loop (while True). Using the read_exact function, it first reads the frame header before using the frame length to extract the payload. Next, frame.set_payload is used to set the payload into the frame object.

The frame, client address, and client socket are then passed to the stream_manager function. This probably controls the receiving frame's stream-specific activities.

The function determines the kind of frame using frame and conditional statements (if-elif).get_frame_type() and responds appropriately:

- Data Frames (0x0): Saved for payload data; at this time, the code is inactive (pass). HTTP headers are contained in header frames (0x1). The function uses hpack to initialize a dynamic table for the client.dynamicTable and decodes the payload of the header frame. Headers that have been decoded are printed.
- Settings Frames (0x4): These frames are handled by using the settings_frame_handler method.
  The purpose of window update frames (0x8) is to regulate flow. Future handling is indicated with the placeholder (pass).
- Reset Stream Frames (0x3): They show that the stream has ended too soon. Placeholder code indicates that additional treatment is anticipated.
- GoAway Frames (0x7): Indicates a graceful connection closure. The placeholder indicates that further work needs to be done.

Stream priority is set via priority frames (0x2). Future handling is suggested by the placeholder.

The function logs the frame type and disregards it if an unknown frame type is received. This guarantees that unidentified frames won't interfere with the server's functionality.

STREAM_MANAGER.PY:

This file is responsible for managing the streams states and also the flow control. The stream manager receives every frame and check 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 a window update frame

```
class StreamManager:
    def stream_manager(self, frame, client_address, socket):
        if (frame.get_frame_type() == 0x0) or (frame.get_frame_type() == 0x1): # DATA frame or HEADERS frame
            if frame.get_frame_type() == 0x0: # DATA frame
                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:
                    # send update window frame by incerement value = 65535
                    WINDOW_UPDATE_STREAM_ID = stream_id
                    WINDOW_UPDATE_FRAME = (struct.pack("!I", len(WINDOW_UPDATE_PAYLOAD))[1:] + struct.pack("!B", WINDOW_UPDATE_FRAME_TYPE) + struct.pack("!B", 0) +
                    socket.sendall(WINDOW_UPDATE_FRAME)
                    stream.set_size(stream.get_size() + 65535)
                if sizes_for_sockets[client_address] < decrement:
                    WINDOW_UPDATE_STREAM_ID = 0x0
                    WINDOW_UPDATE_FRAME = (struct.pack("!I", len(WINDOW_UPDATE_PAYLOAD))[1:] + struct.pack("!B", WINDOW_UPDATE_FRAME_TYPE) + struct.pack("!B", 0) +
                    socket.sendall(WINDOW_UPDATE_FRAME)
                    sizes_for_sockets[client_address] += 65535
            if 0x1 & frame.get_frame_flags(): # END_STREAM flag
                if frame.get_server_initiated():
                    if stream.get_state() == StreamState.OPEN:
                        stream.set_state(StreamState.HALF_CLOSED_REMOTE)
                    elif stream.get_state() == StreamState.HALF_CLOSED_LOCAL:
                        stream.set_state(StreamState.CLOSED)
                else:
                    if stream.get_state() == StreamState.OPEN:
                        stream.set_state(StreamState.HALF_CLOSED_LOCAL)
                    elif stream.get_state() == StreamState.HALF_CLOSED_REMOTE:
```

First, the function determines the type of frame. When the frame type is either data (`0x0`) or headers (`0x1`), payload and flow control logic are handled. The padding length is collected and the frame's size is reduced if the frame has padding (shown by `flags & 0x8`).

The data or header size is decremented from the stream's window size (`stream.set_size`) in order to manage window size. The client receives a window update frame if the final size hits or drops below a threshold (such as `65535`). This guarantees that the client is informed about the server's ability to receive additional data.

Next, a global window size (`sizes_for_sockets`) is checked by the function. The same threshold check is carried out if the size of the frame requires that this value be decremented. The `socket.sendall()` method is used to send a window update frame in order to keep flow control and avoid congestion.

The stream's state is also updated if the frame's flags indicate that the stream has ended (`flags & 0x1`). To ensure the correct stream lifespan, the stream can be set to either `CLOSED` or one of two half-closed states (`HALF_CLOSED_LOCAL` or `HALF_CLOSED_REMOTE`), depending on the situation.

The stream state defaults to `OPEN` if the frame type or conditions do not fit any of the aforementioned scenarios. This guarantees that continuous communication will continue until new instructions are given.

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

It consists of the 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

```python
# ---------------------------------- 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
```

The `DynamicTable` class, defined by this code, implements a dynamic data structure that is frequently used for header compression in HTTP/2 protocols. When the table reaches its limit, it handles the eviction of older items and manages key-value pairs with a set maximum size. The `__init__` method is used to initialize the class, creating an empty list for the table (`self.table`). While `max_size` specifies the maximum permitted size for the table, `current_size` maintains track of the entire size of the entries in the table. `max_size` is set to 4096 by default.

An additional name-value pair can be added to the table using the `add_entry` method. It determines the new entry's size, taking into account the value, the name's length, and a 32-byte overhead. The technique removes older entries from the front of the table (FIFO) until there is sufficient room if adding the new entry would make the table larger than it can be. The size of the evicted items is deducted from the `current_size` during the eviction process. By using its index, the `get_entry` method fetches an entry from the table. A `ValueError` is raised if the index is invalid (for example, `0` or longer than the table's length). If not, the entry at the designated index is returned.

By setting `current_size` to 0 and resetting the table to an empty list, the `clear_dynamic_table` method completely clears the table. When necessary, this can be used to reset or reinitialize the table.
Lastly, the `get_max_size` method returns the maximum allowable size of the table, providing access to the `max_size` property.

Encoding function

Encodes headers into encoded compressed bytes

```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:
```

encode_integer function

This function uses a prefix size (prefix_bits) to encode an integer into a byte sequence. First, it determines the maximum value (max_prefix_value) that the prefix can represent. It immediately returns the value as a byte if the integer (value) is smaller than this upper limit. If not, max_prefix_value is subtracted from the integer, and a loop is started to encode the remaining value in 7-bit chunks. If further bytes are needed, a continuation flag (128) is added. Until the value is contained in a single byte, which is added as the last byte, the process is repeated. When encoding compact variable-length integers, this function is frequently utilized.

encode_string Function

This function uses an optional Huffman encoding to convert a string into bytes. The string is first converted to UTF-8 bytes. It sets a flag (huffman_flag) to 0x80 if Huffman encoding is enabled and to 0x00 otherwise. The encode_integer function is then used to encode the length of the encoded string with a 7-bit prefix. The function then produces a single byte sequence that contains the length-encoded bytes, the Huffman flag, and the UTF-8-encoded string.

encode the function

A name-value pair is encoded using the encode function so that it can be added to a dynamic table. First, it determines whether the name-value pair is present in a static_table that has been predefined. If the pair is located, it changes a particular flag in the first byte to identify its source and encodes the pair's index. It looks for the pair in the dynamic_table if it cannot be located in the static table. The same procedure is followed: if the pair is located, a suitable flag is set and its index is encoded. Lastly, it only encodes the name if the pair is not present in either table but the name is in the static table.

The function would normally add the name-value pair to the dynamic table (not shown here) if it is not already in any table and indexing is enabled. Either inline values or table references can be used to efficiently encode header fields using the returned bytes.

Decoding functions

decodes into normal headers

```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
```

The `decode_integer` function, defined by this code, decodes an integer that has been encoded using HPACK variable-length encoding. In the first byte of the data, it first extracts the integer value from the designated prefix bits. The decoded value is returned if the value falls inside the prefix. If not, it goes into a loop to handle continuation flags (`0x80}) and process further bytes, adding their contributions to the integer. The number of bytes read and the decoded integer are returned by the function.

```python
def decode_string(data):
    """
    Decodes a string from HPACK format.
    """
    huffman = data[0] & 0x80
    if huffman:
        length, consumed = decode_integer(data, 7)
        string = ht.decode_huffman(data[consumed:consumed + length])
        return string, consumed + length
    length, consumed = decode_integer(data, 7)
    string = data[consumed:consumed + length].decode("utf-8")
    return string, consumed + length
```

The `decode_string` function, which decodes a string from HPACK format, is defined by this code. By looking at a flag in the first byte, it first determines whether the string is Huffman-encoded. When Huffman encoding is applied, the string's length is decoded, and the string is then retrieved using a Huffman decoder. If not, the length is decoded, and the string is extracted as UTF-8. The decoded string and the total bytes used for decoding are returned by the function.

FRAMES.PY

Class frame is made managing the frames properly

```python
class Frame:
    frame_length = 0
    frame_type = 0
    frame_flags = 0x0
    stream_id = 0
    payload = b""
    server_initiated = False
    def __init__(self, frame, server_initiated=False):
        frame_length, frame_type, frame_flags, stream_id = struct.unpack("!I B B I", b"\x00" + frame)
        self.frame_length = frame_length
        self.frame_type = frame_type
        self.frame_flags = frame_flags
        self.stream_id = stream_id
        self.server_initiated = server_initiated

    def __str__(self):
        return f"Frame Length: {self.frame_length}, Frame Type: {self.frame_type}, Frame Flags: {self.frame_flags}, Stream ID: {self.stream_id}, Payload: {self.payload}"

    def __repr__(self):
        return f"Frame Length: {self.frame_length}, Frame Type: {self.frame_type}, Frame Flags: {self.frame_flags}, Stream ID: {self.stream_id}, Payload: {self.payload}"

    def get_frame_length(self):
        return self.frame_length

    def get_frame_type(self):
        return self.frame_type

    def get_frame_flags(self):
        return self.frame_flags

    def get_stream_id(self):
        return self.stream_id

    def get_payload(self):
        return self.payload
```

The `Frame` class defined by this code denotes a frame in a communication protocol, most likely HTTP/2. The class is in charge of handling and interpreting frame-related data, such as payload, stream ID, length, type, and flags.

The `__init__` method unpacks the frame's metadata from a binary representation in order to initialize the frame object. From the supplied frame data, it extracts the frame length, type, flags, and stream ID using `struct.unpack`. The instance variables `frame_length`, `frame_type`, `frame_flags`, and `stream_id` contain these values. `server_initiated` is an extra parameter that indicates if the server started the frame.

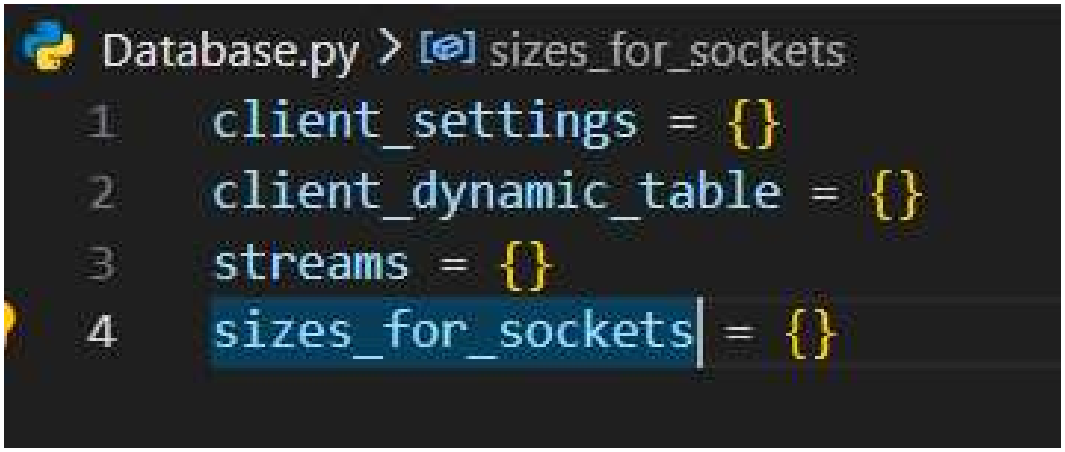
To facilitate debugging and logging of frame details, the `__str__` and `__repr__` methods establish string representations for the `Frame` object. Key attributes of the frame, such as its length, type, flags, stream ID, and payload, are returned in a prepared string.

To access particular frame properties, the class contains getter methods (`get_frame_length`, `get_frame_type`, `get_frame_flags`, `get_stream_id`, and `get_payload`). By offering regulated access to the frame's attributes, these techniques guarantee encapsulation.

All things considered, this class captures the behavior and structure of a protocol frame, making it simple to parse, describe, and retrieve frame-related data. It functions as a fundamental building piece for the protocol's communication processing.

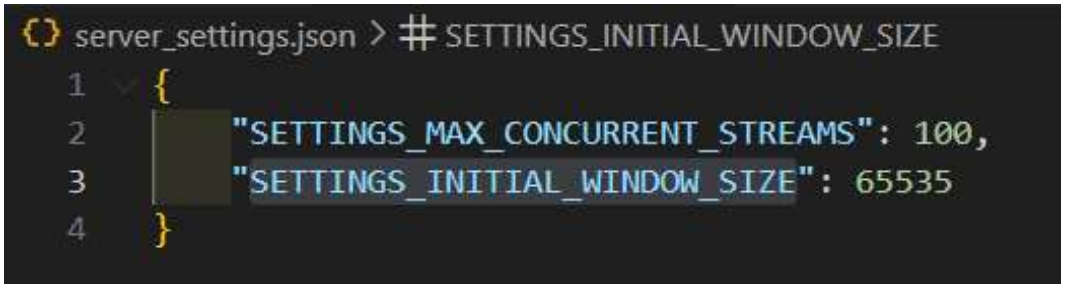
DATABASE.PY

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



SERVER_SETTINGS.JSON

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



TESTING

We will use nghttp to test by sending text file of 944KB:

The client sent the preface and the settings frame exchange completed

Headers frame is sent to the server as well as the first four data frame and because of the text file is very large it will be sent on many data frames and we can also see the window update that the server sends for both the stream level and the connection level

The server also show the same in its output as we can see the receiving of the preface and exchanging the settings frame and decoding the headers and printing the dynamic table and receiving data frames and sending window update