HTTP Library :

HTTP Request and Response Protocol :

Structure of an HTTP Request
Request Line: This specifies the HTTP method, the path of the resource being requested, and the version of HTTP being used.

HTTP Method: Defines the action to be performed (e.g., GET, POST, PUT, DELETE).
Path: The URL or resource path on the server.

HTTP Version: Typically HTTP/1.1 or HTTP/2.0.

Headers: Key-value pairs that provide meta-information about the request.

Headers like Content-Type, User-Agent, and Host specify details about the client and the content being sent.

Empty Line: Separates headers from the body (if any).

Body (Optional): Contains data sent to the server (mostly used in POST, PUT requests). For instance, form data, JSON, etc.

HTTP Request Example:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0
.8
Connection: keep-alive
```

Common HTTP Methods :

GET: Retrieve data from the server. The request usually has no body.
Example: Fetching a webpage.

POST: Send data to the server (often in the body) to create or update a resource.
Example: Submitting a form, uploading data.

PUT: Send data to the server to replace or update a resource.
Example: Updating a user profile.

DELETE: Remove a resource on the server.
Example: Deleting a file from the server.

HEAD: Similar to GET, but only fetches headers, not the body.
Example: Checking the last modified date of a file.

OPTIONS: Describes the communication options for the target resource.

PATCH: Partially update an existing resource.

Common HTTP Request Headers :

Host: The domain name of the server (or IP address).
Content-Type: Specifies the media type of the body (e.g., application/json, text/html).
Authorization: Provides credentials for authentication (e.g., Bearer tokens, Basic Auth).
User-Agent: Details about the client making the request (e.g., browser and OS details).
Accept: Specifies the types of media that the client can accept (e.g., application/json).

HTTP Response Protocol:

Structure of an HTTP Response :
Status Line: Specifies the HTTP version, a status code, and a status message.

HTTP Version: Typically HTTP/1.1 or HTTP/2.0.
Status Code: A 3-digit number indicating the result of the request (e.g., 200, 404).
Status Message: A brief explanation of the status code (e.g., "OK", "Not Found").
Headers: Key-value pairs providing meta-information about the response, such as content type, length, and caching.

Empty Line: Separates headers from the body.

Body (Optional): Contains the resource being returned, such as an HTML page, JSON data, etc.

HTTP Response Example:

```
HTTP/1.1 200 OK
Date: Tue, 06 Oct 2024 12:34:56 GMT
Server: Apache/2.4.1 (Unix)
Content-Type: text/html
Content-Length: 1024

<html>
  <head><title>Example Page</title></head>
  <body><h1>Welcome to the Example Page</h1></body>
</html>
```

Common HTTP Status Codes :
1xx (Informational): The server acknowledges a request, but no action has been taken.
100 Continue: The server is ready to receive the body of the request.

2xx (Success): The request was successfully processed.
200 OK: The request succeeded.
201 Created: The request succeeded, and a new resource was created.

3xx (Redirection): The client must take additional action to complete the request.
301 Moved Permanently: The resource has moved to a new URL.
302 Found: The resource is temporarily located elsewhere.

4xx (Client Error): The request contains incorrect syntax or cannot be fulfilled.
400 Bad Request: The server could not understand the request due to invalid syntax.
401 Unauthorized: Authentication is required.
403 Forbidden: The client is not authorized to access the resource.
404 Not Found: The requested resource could not be found.

5xx (Server Error): The server encountered an error while processing the request.
500 Internal Server Error: The server encountered an unexpected condition.
503 Service Unavailable: The server is temporarily unavailable.

Common HTTP Response Headers :

```
Content-Type: Specifies the media type of the returned content (e.g.,
application/json, text/html).
Content-Length: The length (in bytes) of the body content.
Date: The date and time when the response was generated.
Server: Information about the server software.
Set-Cookie: Sends a cookie to the client (used for session management).
```

Design :

Flow:
TCPServer accepts a new connection → creates a Connection object → registers callbacks for recv/send completion.
Connection receives data until headers and body are fully received → triggers a callback to notify the higher layer (which is passed down when registering with TCPServer).
HTTPServer creates the HTTPRequest and HTTPResponse once notified by the Connection and processes the request via the registered handler.

Things that can be implemented for proper http compliant :

Routing: Map URLs to handlers (e.g., /users/:id) and support query params (?key=value)
Headers: how to set, read, and use HTTP headers properly.
Persistent Connections: Implement keep-alive connections.
Security: SSL/TLS, HTTPS, and security headers.
Concurrency: Use multithreading or asynchronous I/O.(done)
Sessions and Cookies: Managing user state.
File Uploads: Handle multipart data.
Rate Limiting: Prevent abuse and overload.
Advanced HTTP: Support for WebSockets and HTTP/2.
Logging and Monitoring : to keep track of requests

Summary of the key differences between HTTP/1.0 and HTTP/1.1:

Persistent Connections:
HTTP/1.0: A new connection is opened for each request.
HTTP/1.1: Connections stay open by default, allowing multiple requests/responses on the same connection (keep-alive).

Chunked Transfer Encoding:
HTTP/1.0: Requires the server to know the full content size before sending.
HTTP/1.1: Supports sending data in chunks when the size is not known in advance.

Host Header:
HTTP/1.0: Optional, does not support multiple domains on the same IP.
HTTP/1.1: Mandatory, allowing virtual hosting (multiple domains on one server).

Pipelining:
HTTP/1.0: Sends one request at a time.
HTTP/1.1: Allows sending multiple requests without waiting for responses (pipelining).

Caching:
HTTP/1.0: Basic caching using Expires header.
HTTP/1.1: Advanced caching controls with Cache-Control headers.

Range Requests:
HTTP/1.0: Does not support partial content requests.
HTTP/1.1: Supports partial content requests, useful for resuming downloads.

Connection Management:
HTTP/1.0: Limited control over connection reuse.
HTTP/1.1: Provides control over connection management with headers like Connection: keep-alive and Connection: close.

Error Handling:
HTTP/1.0: Fewer status codes for handling errors.
HTTP/1.1: More detailed error status codes (e.g., 100 Continue, 206 Partial Content).

Key Differences Between HTTP/1.1 and HTTP/2:

Binary Protocol:
HTTP/1.1: Uses a text-based protocol. Requests and responses are plain text and human-readable.
HTTP/2: Is a binary protocol. All communication is done in binary format, which makes it more efficient for parsing and transmitting.

Multiplexing:
HTTP/1.1: Only allows one request per TCP connection at a time. If multiple requests are made, the server has to process them sequentially or create multiple connections (which introduces overhead).
HTTP/2: Supports multiplexing, meaning multiple requests and responses can be sent over a single TCP connection simultaneously. This removes the need for multiple connections and significantly reduces latency.

Header Compression:
HTTP/1.1: Headers are sent in full with every request and response, leading to redundancy and larger message sizes.
HTTP/2: Uses HPACK header compression to reduce the size of the headers and avoid redundant transmission of the same headers (like User-Agent and Cookie) across multiple requests.

Server Push:
HTTP/1.1: The client must request all resources (e.g., HTML, CSS, JavaScript).
HTTP/2: Allows the server to “push” resources to the client before it even requests them. For example, if the server knows that a CSS file is needed when sending an HTML file, it can push the CSS file immediately.

Stream Prioritization:
HTTP/1.1: No built-in prioritization of requests. Browsers typically open multiple connections to handle this manually.
HTTP/2: Requests can be prioritized, allowing critical resources to load faster.
Single Connection:
HTTP/1.1: Typically uses multiple TCP connections to handle multiple requests, which increases resource consumption on both client and server.
HTTP/2: Uses one TCP connection to handle multiple streams (requests), reducing the overhead of setting up multiple connections.

How to implement the HTTP 2.0 PROTOCOL :

Steps to Implement HTTP/2 Multiplexing:

Frame Layer Implementation: Instead of using a text-based request-response model (like HTTP/1.1), We will need to implement a frame-based communication model. Each piece of data sent over the connection is encapsulated in frames. For multiplexing, we will need to support multiple stream identifiers within these frames.

Stream Management: Manage multiple streams per connection. Each stream will have its own identifier, and we must manage its state.

Concurrency and Flow Control: Streams should be independent, and the server needs to handle multiple streams concurrently over the same TCP connection. We must also manage flow control for these streams to avoid overloading the client or server.

State Handling for Each Stream: Streams in HTTP/2 can be in different states: idle, open, half-closed, closed. We need to manage state transitions based on the frames received and sent.

Detailed Explanation of HTTP/2 Multiplexing :

Binary Framing Protocol:

Every HTTP/2 communication is based on frames. Each frame is part of a stream and contains either control information (like headers) or actual data (like a portion of the body).

Frame Structure: Each frame consists of:
Length (3 bytes)
Type (1 byte, e.g., HEADERS, DATA, etc.)
Flags (1 byte)
Stream Identifier (4 bytes)
Frame-specific payload (the rest)
You'll need to parse and generate these frames as binary data, ensuring that frames can be multiplexed on different streams.

HTTP/2.0 operates fundamentally differently from HTTP/1.x in terms of how requests and responses are transmitted. Instead of relying on a text-based format with clear line delimiters, HTTP/2 breaks all communication into smaller units called frames, which are transmitted over streams. Here’s an in-depth look at how the binary protocol works, along with examples to explain the flow.

HTTP/2 Concepts: Streams, Frames, and Multiplexing

Streams: A stream is a bidirectional flow of frames that represents a single logical request-response transaction. Each HTTP/2 connection can have many streams open simultaneously.

Frames: Frames are the smallest unit of communication in HTTP/2. Each frame belongs to a particular stream and carries a specific type of information (e.g., headers, data, settings). Frames are binary structures that are much more efficient to parse and process compared to the text-based format of HTTP/1.x.

Multiplexing: Multiple streams can exist on a single HTTP/2 connection, and frames from different streams can be interleaved. This allows multiple requests and responses to be transmitted concurrently over the same connection.

Structure of an HTTP/2 Frame
All HTTP/2 frames share the same basic structure:

Length (24 bits): Specifies the length of the frame payload (up to 16 MB).
Type (8 bits): Specifies the type of frame (e.g., HEADERS, DATA, SETTINGS).
Flags (8 bits): Used for additional control information.
Stream Identifier (31 bits): Identifies the stream to which the frame belongs. A stream ID of 0 is used for frames that apply to the entire connection (e.g., SETTINGS frames).

Types of HTTP/2 Frames :
HEADERS: Contains HTTP headers and marks the start of a stream.
DATA: Carries the payload of a request or response (e.g., the body of an HTTP message).
SETTINGS: Used to negotiate connection settings between the client and server.
PRIORITY: Allows prioritization of streams.
RST_STREAM: Resets a stream.
PUSH_PROMISE: Used by the server to "push" resources to the client before they are requested.
WINDOW_UPDATE: Used for flow control.

Stream Handling:

HTTP/2 streams are identified by stream IDs, and multiple streams can exist at the same time on a single TCP connection. For instance, you might be serving two different requests simultaneously over stream IDs 1 and 3.
The client and server must manage the state of each stream independently. For example, stream 1 might be sending headers while stream 3 is sending data. Your server must be able to differentiate and handle them concurrently.

```
struct HTTP2Stream {
    int stream_id;  // Unique identifier for the stream
    std::string headers;  // HTTP headers
    std::string body;     // Data for this stream
    StreamState state;    // Open, half-closed, closed
    // Other fields like flow control, priority, etc.
};
```

Frame Handling:

We need to define how to parse and handle frames based on their types (e.g., HEADERS, DATA, etc.).

```
void handleFrame(const std::vector<uint8_t>& frameData) {
    FrameType frameType = static_cast<FrameType>(frameData[3]); // Type
byte
    int streamId = extractStreamId(frameData);

    switch (frameType) {
        case FrameType::HEADERS:
            handleHeadersFrame(streamId, frameData);
            break;
        case FrameType::DATA:
            handleDataFrame(streamId, frameData);
            break;
        case FrameType::SETTINGS:
            handleSettingsFrame(frameData);
            break;
        // Handle other frame types like PING, WINDOW_UPDATE, etc.
    }
}

void handleHeadersFrame(int streamId, const std::vector<uint8_t>&
frameData) {
    // Parse headers for the stream and store them
    streams[streamId].headers = parseHeaders(frameData);
    streams[streamId].state = StreamState::OPEN;
}

void handleDataFrame(int streamId, const std::vector<uint8_t>&
frameData) {
    // Append data to the stream's body
    streams[streamId].body.append(parseData(frameData));
}
```

Multiplexing Streams:

We need to ensure that your server can manage multiple concurrent streams. Here's a basic structure to handle this:

Each connection can manage multiple streams, with each stream represented by an identifier (stream_id).
Streams are processed independently. This requires concurrency—each stream could be handled in a separate thread or via an asynchronous mechanism.

```
std::unordered_map<int, HTTP2Stream> streams;

void handleIncomingFrames(int connectionId, const std::vector<uint8_t>&
frameData) {
    // Parse incoming frames
    while (hasMoreFrames(frameData)) {
        handleFrame(extractFrame(frameData));
    }
}

// Simulate asynchronous handling of multiple streams
void handleStreams() {
    for (auto& [streamId, stream] : streams) {
        if (stream.state == StreamState::OPEN) {
            // Handle the stream's request in a separate thread or task
            processStream(stream);
        }
    }
}
```

Concurrency:

To fully support multiplexing, each stream should be processed asynchronously, meaning that responses for one stream don't block others. You could use threads, coroutines, or an event-driven model (e.g., epoll on Linux) to achieve this.

```
void processStream(HTTP2Stream &stream) {
    // Process the request, generate a response
    std::thread([&]() {
        std::string responseBody = generateResponse(stream);
        sendDataFrame(stream.stream_id, responseBody);
    }).detach();
}
```

Handling Flow Control:

In HTTP/2, flow control applies to both the connection and individual streams. We will need to manage WINDOW_UPDATE frames to ensure that no stream overwhelms the client or server.

```
void handleWindowUpdateFrame(int streamId, const std::vector<uint8_t>&
frameData) {
    // Parse the new window size and update flow control settings
    streams[streamId].windowSize = parseWindowSize(frameData);
}
```

How HTTP/2 Recognizes the End of Requests and Responses :

HEADERS and DATA frames are part of the same stream: Both the client and server track streams by their stream identifiers. Frames with the same stream ID are part of the same request-response pair.
END_STREAM flag: The final frame for any given request or response is marked with the END_STREAM flag. This tells the receiving side that all frames for the given stream (and thus the request or response) have been sent.

Sample code for handling data frames on server side :

```
#include <iostream>
#include <sys/socket.h>
#include <unistd.h>
#include <vector>
#include <cstring>

// Constants for HTTP/2
const int HTTP2_FRAME_HEADER_SIZE = 9;  // Fixed size of the HTTP/2
frame header

// Function to receive exactly `len` bytes from a socket
ssize_t recv_exactly(int sockfd, char* buffer, size_t len) {
    size_t total_received = 0;
    ssize_t received;

    while (total_received < len) {
        received = recv(sockfd, buffer + total_received, len -
total_received, 0);
        if (received <= 0) {
            // Error or connection closed
```

```cpp
            return received;
        }
        total_received += received;
    }

    return total_received;
}

// Struct to represent an HTTP/2 frame
struct HTTP2Frame {
    uint32_t length;      // Length of the frame payload (24 bits)
    uint8_t type;         // Type of the frame
    uint8_t flags;        // Flags
    uint32_t stream_id;  // Stream identifier (31 bits)
    std::vector<char> payload;  // Payload data

    // Parse the frame header
    bool parse_header(const char* buffer) {
        length = (buffer[0] << 16) | (buffer[1] << 8) | buffer[2];
        type = buffer[3];
        flags = buffer[4];
        stream_id = (buffer[5] & 0x7F) << 24 | (buffer[6] << 16) |
(buffer[7] << 8) | buffer[8];
        return true;
    }

    // Print the frame details
    void print() {
        std::cout << "Frame Length: " << length << ", Type: " <<
static_cast<int>(type)
                  << ", Flags: " << static_cast<int>(flags) << ", Stream
ID: " << stream_id << std::endl;
    }
};

// Function to handle receiving and processing frames
bool handle_http2_frame(int sockfd) {
    char header_buffer[HTTP2_FRAME_HEADER_SIZE];

    // Step 1: Read the frame header
    ssize_t received = recv_exactly(sockfd, header_buffer,
HTTP2_FRAME_HEADER_SIZE);
    if (received <= 0) {
        std::cerr << "Failed to receive frame header." << std::endl;
        return false;
    }
    // Step 2: Parse the frame header
    HTTP2Frame frame;
    if (!frame.parse_header(header_buffer)) {
        std::cerr << "Failed to parse frame header." << std::endl;
        return false;
    }

    // Step 3: Read the frame payload
    frame.payload.resize(frame.length);  // Adjust buffer size for
payload
    received = recv_exactly(sockfd, frame.payload.data(), frame.length);
    if (received <= 0) {
        std::cerr << "Failed to receive frame payload." << std::endl;
        return false;
    }

    // Step 4: Process the frame (printing for now)
    frame.print();

    return true;
}

int main() {
    // Dummy socket file descriptor for the example
    int sockfd = 0;  // This would be a valid socket in a real server

    // Example loop to handle multiple frames
    while (true) {
        if (!handle_http2_frame(sockfd)) {
            std::cerr << "Error handling HTTP/2 frame, closing
connection." << std::endl;
            break;
        }
    }

    // Close socket (cleanup)
    close(sockfd);
    return 0;
}
```

Summary of the differences between HTTP/2 and HTTP/3:

Transport Protocol:
HTTP/2: Uses TCP for reliable data transmission.
HTTP/3: Uses QUIC, which is built on UDP and faster.

Head-of-Line Blocking:
HTTP/2: If one packet is lost, all streams wait (TCP issue).
HTTP/3: Only the affected stream waits, others continue.

Connection Setup:
HTTP/2: Requires separate TCP and TLS handshakes, which takes more time.
HTTP/3: Combines handshakes into one step, reducing connection setup time.

Multiplexing:
HTTP/2: Multiple requests can be sent over a single connection, but all can be delayed by one packet loss.
HTTP/3: Multiple independent streams can continue even if one is delayed.

Security:
HTTP/2: Encryption (TLS) is optional.
HTTP/3: Encryption (TLS 1.3) is required.

Performance:
HTTP/2: Latency can increase due to separate handshakes and packet loss.
HTTP/3: Faster connection, better performance in bad networks due to quick handshakes and no stream blocking.

Packet Loss Handling:
HTTP/2: All streams are delayed if one packet is lost.
HTTP/3: Only the stream with the lost packet is delayed.

Adoption:
HTTP/2: Widely used and supported by most servers and browsers.
HTTP/3: Newer, still growing in adoption, but gaining support.

HTTPS PROTOCOL :
HTTPS is essentially HTTP over TLS/SSL (Transport Layer Security / Secure Sockets Layer), which provides encryption, authentication, and data integrity.

Here's what happens behind the scenes, step-by-step:

1. Establishing a TCP Connection
First, the client initiates a TCP connection with the server on port 443 (the standard HTTPS port).
This step is identical to HTTP — the 3-way TCP handshake (SYN, SYN-ACK, ACK) is completed.

2. TLS/SSL Handshake Begins
After the TCP connection, the TLS/SSL handshake starts. This is where the magic happens.
The handshake ensures:
Both parties (client and server) agree on encryption algorithms.
The server's identity is verified.
A shared key is generated for encrypting data.

Step-by-Step TLS Handshake:
1. Client Hello
The client sends a "Client Hello" message. It contains:
A list of encryption algorithms (ciphers) it supports (e.g., AES, RSA, etc.).
A random number used to generate encryption keys.
The TLS version it wants to use (e.g., TLS 1.2, TLS 1.3).

2. Server Hello
The server responds with a "Server Hello" message. It contains:
The encryption algorithm it has chosen (from the client's list).
Another random number for key generation.
A digital certificate to prove the server's identity.

3. Server Certificate Exchange & Authentication
The server sends its certificate, which includes:
The server's public key.
The domain name.
The CA (Certificate Authority) that issued the certificate.
The client verifies the certificate:

Checks the certificate's validity (e.g., is it expired?).
Verifies the certificate's chain of trust (checks if it was issued by a trusted CA).
Validates the domain name in the certificate (e.g., the certificate should be issued for example.com if the client is accessing https://example.com).
If any of these checks fail, the connection is rejected.

4. Key Exchange
In modern TLS (TLS 1.2 or 1.3), Ephemeral Diffie-Hellman (ECDHE) is often used for key exchange.
Both the client and server use their random numbers and the server’s public key to compute a shared secret.
This shared secret becomes the "session key", which will be used to encrypt all further communication.

5. Session Key Generation
Both the client and server now have a shared session key.
They use this key to encrypt the actual HTTP data (e.g., GET or POST requests).

6. Encrypted Communication Begins
All further communication is encrypted using the session key.
The client sends an encrypted HTTP request (e.g., GET /index.html), and the server responds with an encrypted HTTP response.

7. Data Integrity via MAC
TLS ensures data integrity by attaching a Message Authentication Code (MAC) to every message.
If a message is tampered with in transit, the MAC will not match, and the connection will be terminated.

8. Connection Closure (Optional)
Once the communication is complete, the client and server can close the connection.
If session resumption is enabled (via TLS session tickets), the client and server can reuse the session key for future connections, reducing handshake overhead.

Summary of Steps in HTTPS Handshake :
TCP Handshake: Establish a basic connection.
Client Hello: Client proposes encryption algorithms, TLS version, and sends a random number.(AES, RSA, ECDSA) (TLS 1.2, TLS 1.3) .
Server Hello: Server agrees on the encryption algorithm, sends its random number.
Server Certificate: Server sends its digital certificate for authentication.
Key Exchange: Client and server generate a shared session key (via RSA or ECDHE).
Finished Messages: Both parties confirm the handshake.
Encrypted Communication: All further data is encrypted with the session key.
Connection Close: Optionally, the connection is closed or reused.

Example Cipher Suite :
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

This suite contains:

TLS: The protocol (can be TLS 1.2, TLS 1.3, etc.).
ECDHE: Key exchange algorithm (Elliptic Curve Diffie-Hellman Ephemeral).
RSA: Authentication algorithm (RSA-based certificates are used).

AES_256_GCM: Symmetric encryption algorithm (AES with 256-bit key in Galois/Counter Mode).
SHA384: MAC algorithm (HMAC with SHA-384).

What is a CA Certificate in HTTPS?
A CA (Certificate Authority) certificate is a digital certificate issued by a trusted organization (the CA) that serves as the basis for verifying the identity of websites or servers in HTTPS. When a browser connects to an HTTPS site, it trusts the certificate because it was issued by a CA that is part of the browser's trusted CA store.

How CA Certificates Work in HTTPS :
Certificate Hierarchy / Chain of Trust: HTTPS uses a hierarchical trust model consisting of:

Root CA certificates: These belong to top-level Certificate Authorities.
Intermediate CA certificates: Issued by root CAs or other intermediates to delegate signing authority.
Server certificates: Issued by a CA to identify a specific domain (e.g., example.com).
A browser verifies the server’s certificate by following the chain of trust from the server's certificate up to a trusted root CA certificate.

What a CA Certificate Contains :

A typical CA certificate contains the following information:

Subject Information:
Name of the entity (CA or organization) the certificate was issued to.
Domain name or Common Name (CN), e.g., www.example.com

Issuer Information:
Identifies the CA that issued this certificate.
For a root CA, the issuer will be the same as the subject (self-signed).

Public Key:
A public key (as part of the Public Key Infrastructure, or PKI) that corresponds to the private key held by the certificate owner.

Validity Period:
Start and end dates specifying when the certificate is valid.

Serial Number:
A unique identifier for the certificate.

Signature Algorithm:
The algorithm used to sign the certificate (e.g., SHA-256 with RSA).

CA’s Digital Signature:
A signature by the issuing CA that verifies the certificate's authenticity.

Certificate Extensions:
Additional metadata, such as:
Basic Constraints: Whether the certificate is a CA certificate.
Key Usage: How the certificate can be used (e.g., for signing or encryption).

Subject Alternative Names (SAN): Lists all domain names covered by the certificate.

Use openssl library to implement https on top of http :
command to install openssl in linux :
command :
```
sudo apt-get update
sudo apt-get install libssl-dev
```

Create a Self-Signed Certificate (for Testing) :
command :
```
openssl req -x509 -newkey rsa:2048 -keyout server.key -out server.crt -days 365 -nodes
```

This command creates:

server.crt: Certificate.
server.key: Private key.

checking whether your https server is running properly :
command :
```
openssl s_client -connect localhost:8000
```

CMakeLists.txt file changes to use openssl library :
```
# Find the OpenSSL library.
find_package(OpenSSL REQUIRED)

# Add tcpserver.cpp
add_library(tcpserver STATIC tcpserver.cpp ${SERVER_OS_SRC})

# Link the OpenSSL libraries to the target.
target_link_libraries(tcpserver PUBLIC OpenSSL::SSL OpenSSL::Crypto)
```

Analyse the crash on Linux using coredump file :

1. Enable Core Dumps on your machine
Command : ulimit -c unlimited

2. Run Your Program :
Run your program as usual, and if it crashes, a core dump file will be generated (usually named core or core.<pid>).

3. Use GDB to Analyze the Core Dump :
gdb <your-executable> <core-dump-file>

4. Use GDB Commands to analyse :
Command : bt
Backtrace: This shows the function call stack leading to the crash:

Command : print <variable_name>
examine variables at the time of the crash:

Command : info threads
thread apply all bt

Examine Threads (if using multithreading):

5. Change the path to current directory where coredump will be generated :
Command : sudo sysctl -w kernel.core_pattern=core.%e.%p

To make the above change permanent :
add the following line to /etc/sysctl.conf :
kernel.core_pattern=core.%e.%p

And then apply the changes with this command :
sudo sysctl -p

Run the tool such as Valgrind to check for memory leaks and invalid memory access issues :
Command : valgrind ./myserver
This will give you a detailed report of any memory-related issues that could cause a crash.

Creating HTTP SERVER using uwebsocket library :

Link to uwebsocket github -> https://github.com/uNetworking/uWebSockets

Steps to install uwebsocket library :

steps to use uWebsocket on linux :

Clone the Repository with Submodules :
commands :

```
git clone --recurse-submodules https://github.com/uNetworking/uWebSockets.git
cd uWebSockets
sudo WITH_OPENSSL=1 make examples
sudo make
sudo make install (this will only copy the uWebsockets header files to
/usr/local/include/uWebsocket. need to copy the uSockets header and library files too )

sudo cp uSockets/*.a /usr/local/lib/ (uSockets folder is inside the uWebSockets folder )
sudo cp uSockets/src/*.h /usr/local/include/
sudo ldconfig (update the cache)
```

then add these libraries as the linker dependencies in the CMakeLists.txt file :

```
target_link_libraries(httpserver PRIVATE -l:uSockets.a ssl crypto z pthread)
```

If You Already Cloned the Repository :
command :

```
cd uWebSockets
git submodule update --init --recursive
```

install zlib development package :
command :

```
sudo apt install -y zlib1g-dev
```

install openssl library too :
command :

```
sudo apt-get install libssl-dev
```

steps to use uWebSockets on Windows(works when project is directly build using visual studio) :

Steps to build and run the uwebsockets library on windows :

1. Install vcpkg manager on windows.

To install vcpkg manager on windows :
a. go to website - https://vcpkg.io/en/getting-started.html
b. and run commands as mentioned there .

```
   git clone https://github.com/Microsoft/vcpkg.git

   .\vcpkg\bootstrap-vcpkg.bat

   vcpkg integrate install(will be added in the visual studio)

   also set in the path of the env variable
```

2. Now install the uwebockets library using vcpkg manager

```
set VCPKG_DEFAULT_TRIPLET=x64-windows (optional)
vcpkg install uwebsockets
```

install openssl -

```
vcpkg install openssl
```

install zlib :

```
vcpkg install zlib
vcpkg install zlib --triplet x64-windows-static (for debug version)
```

Designated initializers in c++ 20 :
Designated initializers allow you to initialize specific members of a struct like this:

```
struct PerSocketData {
    std::vector<std::string> topics;
    int nr;
};

PerSocketData data = {
    .topics = {"chat", "news"},
    .nr = 42
};
```

Sample code for http server using uwebsocket library :

```cpp
#include <App.h>
#include <string>

int main() {
    uWS::App()
        .get("/index", [](auto *res, auto *req) {
            res->writeHeader("Content-Type", "application/json");
            res->end(R"({"message": "Welcome to the index page!"})");
        })
        .get("/shops", [](auto *res, auto *req) {
            res->writeHeader("Content-Type", "application/json");
            res->end(R"({"shops": ["Shop1", "Shop2", "Shop3"]})");
        })
        .get("/users", [](auto *res, auto *req) {
            res->writeHeader("Content-Type", "application/json");
            res->end(R"({"users": ["Alice", "Bob", "Charlie"]})");
        })
        .listen(9001, [](auto *token) {
            if (token) {
                std::cout << "Server listening on port 9001" <<
std::endl;
            } else {
                std::cerr << "Failed to listen on port 9001" <<
std::endl;
            }
        })
        .run();

    return 0;
}
```

Websocket Protocol :

The WebSocket protocol provides full-duplex communication over a single TCP connection, allowing data to flow simultaneously in both directions between a client (like a browser) and a server. This makes it suitable for real-time applications such as chat, gaming, or financial tickers.

Unlike HTTP, which is request-response-based, WebSocket is persistent and allows bi-directional communication without the overhead of multiple HTTP requests.

How WebSocket Protocol Works :

Handshake (Upgrade HTTP to WebSocket):
The connection starts as an HTTP request and then upgrades to WebSocket.
The client sends a WebSocket handshake request to the server.
If the server accepts, it responds with a 101 Switching Protocols status, and the connection switches to WebSocket.

Data Frames:
After a successful handshake, both the client and server exchange frames instead of HTTP requests. A frame contains structured data (like text or binary) and is minimal in size compared to HTTP.

Connection Lifecycle:
Open: After the handshake, the connection is established.
Message: Both the client and server can send messages freely at any time.
Ping/Pong: WebSocket supports ping-pong frames to check if the connection is still alive.
Close: Either side can gracefully close the connection by sending a Close frame.

WebSocket Handshake Example :

Client request:
The client initiates a connection by sending an HTTP Upgrade request to the server.

example request :

```
GET /chat HTTP/1.1
Host: example.com:80
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

Upgrade: Informs the server that the client wants to upgrade from HTTP to WebSocket.
Sec-WebSocket-Key: A unique key used to verify the handshake.

Server response:
If the server accepts the request, it responds with:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Sec-WebSocket-Accept: A hashed response derived from the Sec-WebSocket-Key. It proves the handshake is successful.

WebSocket Frames Structure :

After the handshake, data is exchanged in frames. Each frame has a header and payload.

1. Frame Header Fields:
FIN (1 bit): Indicates if this is the last frame of a message.
RSV1, RSV2, RSV3 (1 bit each): Reserved for future use.
Opcode (4 bits): Specifies the type of frame (e.g., text, binary, ping, pong).
0x1: Text frame
0x2: Binary frame
0x8: Close frame
0x9: Ping frame
0xA: Pong frame
Mask (1 bit): Indicates if the payload is masked (client must always mask data).
Payload length (7/16/64 bits): Length of the payload data.
Masking key (32 bits, optional): Used to mask the payload (required for client frames).

2. Frame Payload:
The payload contains the actual data (text or binary).

example of websocket frame :

```
FIN  | Opcode=0x1 | Mask | Payload Length | Masking Key | Payload Data ("Hello")
1    |   0x1      |  1   |       5        |  32 bits    | Masked Hello
```

Security in WebSockets :

TLS (wss://):
WebSocket connections can be secured using TLS, just like HTTPS. The wss:// scheme ensures encrypted communication.

Cross-Origin Requests:
Since WebSocket does not enforce same-origin policies like HTTP, careful validation of origin headers and authentication mechanisms are necessary.

Create Collaborative Document editor(using uwebsockets library) :

server.cpp

```cpp
#include <App.h>
#include <iostream>
#include <fstream>
#include <string>
#include <unordered_set>
#include <mutex>

using namespace std;

// Struct to store user data for each WebSocket connection
struct PerSocketData {};

// Correct template instantiation for WebSocket
using WebSocket = uWS::WebSocket<false, true, PerSocketData>;

// Store active WebSocket connections
std::unordered_set<WebSocket*> clients;

// Shared document content
std::string shared_document = "Welcome to the Collaborative Document
Editor!";
std::mutex doc_mutex;

// Helper function to read files (HTML, JS) into strings
std::string readFile(const std::string& filepath) {
    std::ifstream file(filepath);
    if (!file) {
        std::cerr << "Error opening file: " << filepath << std::endl;
        return "";
    }
    return std::string((std::istreambuf_iterator<char>(file)),
std::istreambuf_iterator<char>());
}

int main() {
    uWS::App()
        .get("/*", [](auto* res, auto* req) {
            std::string url = std::string(req->getUrl());  // Convert
string_view to string
            std::string content;

            // Serve files based on the URL
            if (url == "/") {
                cout<<"came to index.html"<<endl;
                content = readFile("../index.html");
                res->writeHeader("Content-Type", "text/html");
            } else if (url == "main.js") {
                cout<<"came to main.js"<<endl;
                content = readFile("../main.js");
                res->writeHeader("Content-Type",
"application/javascript");
            } else {
                res->writeStatus("404 Not Found")->end("404: File not
found");
                return;
            }

            res->end(content);
        })
        .ws<PerSocketData>("/*", {
            .open = [](WebSocket* ws) {
                std::cout << "Client connected!" << std::endl;

                // Send current document state to the new client
                std::lock_guard<std::mutex> lock(doc_mutex);
                ws->send(shared_document, uWS::OpCode::TEXT);

                clients.insert(ws);
            },
            .message = [](WebSocket* ws, std::string_view message,
uWS::OpCode opCode) {
                // Update shared document and broadcast to all other
clients
                {
                    std::lock_guard<std::mutex> lock(doc_mutex);
                    shared_document = std::string(message);  // Convert
string_view to string
```

```
                }
                //cout<<shared_document<<endl;
                for (auto* client : clients) {
                    if (client != ws) {
                        client->send(message, opCode);
                    }
                }
            },
            .close = [](WebSocket* ws, int code, std::string_view
message) {
                std::cout << "Client disconnected!" << std::endl;
                clients.erase(ws);
            }
        })
        .listen(9001, [](auto* listen_socket) {
            if (listen_socket) {
                std::cout << "Server listening on port 9001" <<
std::endl;
            } else {
                std::cerr << "Failed to start server!" << std::endl;
            }
        })
        .run();

    return 0;
}
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>Collaborative Document Editor</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            padding: 20px;
        }
        textarea {
            width: 100%;
            height: 400px;
        }
    </style>
</head>
<body>
    <h1>Collaborative Document Editor</h1>
    <textarea id="editor" placeholder="Start typing..."></textarea>

    <script>
        const ws = new WebSocket(`ws://${location.host}`);

        // Sync text changes across clients
        const editor = document.getElementById('editor');

        editor.addEventListener('input', () => {
            ws.send(JSON.stringify({ type: 'update', content: 
editor.value }));
        });

        ws.onmessage = (event) => {
            const data = JSON.parse(event.data);
            if (data.type === 'update') {
                editor.value = data.content;
            }
        };

        ws.onopen = () => console.log('Connected to WebSocket server');
        ws.onclose = () => console.log('Disconnected from WebSocket 
server');
    </script>
</body>
</html>
```

CMakeLists.txt :

```
cmake_minimum_required(VERSION 3.10)

# Project Name
project(onlinedoc)

# Set C++ Standard
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

```cmake
set(VCPKG_DIR "C:/vcpkg/vcpkg")

# Include vcpkg toolchain file if it exists
if(EXISTS "${VCPKG_DIR}/scripts/buildsystems/vcpkg.cmake")
    set(CMAKE_TOOLCHAIN_FILE
"${VCPKG_DIR}/scripts/buildsystems/vcpkg.cmake")
endif()

# Detect Operating System
if (WIN32)
    message(STATUS "Configuring for Windows")
    include_directories("${VCPKG_DIR}/installed/x64-windows/include")
elseif (UNIX)
    message(STATUS "Configuring for Linux")
    # Include directories for uSockets and uWebSockets
    include_directories(/usr/local/include/uWebSockets)
else()
    message(FATAL_ERROR "Unsupported OS")
endif()

# Add Executable
add_executable(onlinedoc server.cpp)



# Link Libraries
if (WIN32)
target_link_libraries(onlinedoc PRIVATE
"${VCPKG_DIR}/installed/x64-windows/lib/libssl.lib"
"${VCPKG_DIR}/installed/x64-windows/lib/libcrypto.lib"
"${VCPKG_DIR}/installed/x64-windows/lib/uSockets.lib"
"${VCPKG_DIR}/installed/x64-windows/lib/zlib.lib"
)
elseif (UNIX)
    # Linking against uWebSockets, OpenSSL, Zlib, and pthread
    target_link_libraries(onlinedoc PRIVATE
        -l:uSockets.a
        ssl
        crypto
        z
        pthread
    )
endif()
```

Webrtc Communication protocol :

WebRTC (Web Real-Time Communication) is a protocol that allows real-time peer-to-peer communication between browsers and mobile applications. It facilitates the exchange of audio, video, and data between users without requiring an intermediary server for data transfer. Here’s a detailed breakdown of how WebRTC works, including the steps for establishing a connection and transferring data.

Key Components of WebRTC

PeerConnection: Manages the connection between two peers.
MediaStream: Represents a stream of audio or video.
DataChannel: Allows for bidirectional data communication between peers.
STUN/TURN Servers: Help in NAT traversal to connect peers behind routers or firewalls.

Steps for Establishing a Connection in WebRTC
1. Signalling :
Signalling is the initial communication between peers to exchange connection metadata. This process is not defined by the WebRTC standard, so developers can use any signalling method (e.g., WebSocket, HTTP, etc.). The signalling process involves exchanging:

Session control messages
Network information (ICE candidates)
Media capabilities (SDP – Session Description Protocol)

STUN (Session Traversal Utilities for NAT)

Signalling Steps:
Peer A wants to connect to Peer B. Peer A sends a signalling message to its signalling server to initiate the connection.
The signalling server forwards this message to Peer B.
Peer B responds to Peer A with its own signalling message.
2. ICE Candidate Gathering
ICE (Interactive Connectivity Establishment) is a framework used to find the best path to connect peers.

Each peer gathers ICE candidates, which include the network configurations needed to connect.
Candidates can include local IP addresses, STUN/TURN server addresses, and more.
Gathering Steps:

Each peer creates an RTCPeerConnection and listens for ICE candidate events.
When candidates are found, they are sent through the signaling server to the other peer.
3. SDP Exchange
SDP (Session Description Protocol) is a format for describing multimedia communication sessions.

Peer A creates an SDP offer, including information about media formats and ICE candidates.
Peer A sends this SDP offer to Peer B via the signaling server.
Peer B receives the offer, creates an SDP answer, and sends it back to Peer A.
SDP Exchange Steps:

Peer A: peerConnection.createOffer() → peerConnection.setLocalDescription(offer) → send offer to Peer B
Peer B: peerConnection.setRemoteDescription(offer) → createAnswer() → setLocalDescription(answer) → send answer to Peer A
4. Connection Establishment
Once both peers have exchanged their SDP offers and answers and ICE candidates, they attempt to establish a connection.

Each peer will use the ICE candidates to try to connect to the other peer.
If a candidate connection is successful, the peers can start sending and receiving media/data.
Connection Steps:

Peers use the gathered ICE candidates to connect.
When a connection is established, a connectionstatechange event is fired on both peers.
5. Data Transfer
Once the connection is established, WebRTC allows for data transfer using:

Media Streams: For audio and video. You can send media streams directly using addTrack() or addStream() on the RTCPeerConnection.
Data Channels: For arbitrary data. You can send text, binary data, etc.
Data Transfer Steps:

For DataChannel:
Create a DataChannel on one peer: const dataChannel = peerConnection.createDataChannel("chat");
Send data: dataChannel.send("Hello, Peer!");
For Media Streams:
Capture media: navigator.mediaDevices.getUserMedia({ audio: true, video: true })
Add media to the peer connection: peerConnection.addTrack(track, stream);
6. Closing the Connection
When the communication is finished, the connection can be closed.

Closing Steps:

Call peerConnection.close() on both peers.
Clean up any resources, such as media tracks and data channels.

Webrtc code example :

server.cpp :

```
#include <uwebsockets/App.h>
#include <unordered_map>
#include <iostream>
#include <fstream>
#include <sstream>

// Struct to store user data for each WebSocket connection
struct PerSocketData {
    std::string role;
};

// Store connected WebSocket clients
std::unordered_map<uWS::WebSocket<false, true, PerSocketData>*,
std::string> clients;

int main() {
    // Define WebSocket behavior with PerSocketData as template
    uWS::App()
        .ws<PerSocketData>("/*", {
            .open = [](auto* ws) {
                std::cout << "New WebSocket connection\n";
                ws->getUserData()->role = "";  // Initialize role with
empty string
                clients[ws] = "unknown";  // Add client to the map
            },
            .message = [](auto* ws, std::string_view message,
uWS::OpCode opCode) {
                std::cout << "Received message: " << message << "\n";
                ws->getUserData()->role = std::string(message);  //
Store role

                // Broadcast the message to other clients
                for (auto& [client, role] : clients) {
                    if (client != ws) {  // Avoid sending message to the
same client
                        client->send(message, opCode);
                    }
                }
            },
            .close = [](auto* ws, int code, std::string_view message) {
                std::cout << "WebSocket connection closed\n";
                clients.erase(ws);  // Remove client from the map
            }
        })

        // Serve static HTML files
        .get("/*", [](auto* res, auto* req) {
            std::ifstream file("../public/index.html");
            if (!file.is_open()) {
                res->writeStatus("404 Not Found")->end("HTML file not
found.");
                return;
            }

            std::stringstream buffer;
            buffer << file.rdbuf();
            std::string html = buffer.str();
            res->writeHeader("Content-Type", "text/html")->end(html);
        })

        // Start the server and listen on port 9001
        .listen(9001, [](auto* listen_socket) {
            if (listen_socket) {
                std::cout << "Server is listening on port 9001\n";
            }
        })
        .run();

    return 0;
}
```

index.html :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <title>WebRTC Demo</title>
</head>
<body>
    <h1>WebRTC Video Call Demo</h1>
    <video id="localVideo" autoplay muted></video>
    <video id="remoteVideo" autoplay></video>
    <button id="startCall">Start Call</button>

    <script>
        const localVideo = document.getElementById('localVideo');
        const remoteVideo = document.getElementById('remoteVideo');
        const startCallButton = document.getElementById('startCall');

        const ws = new WebSocket('ws://' + window.location.hostname +
':9001');
        const peerConnection = new RTCPeerConnection();

        // Get local video stream
        navigator.mediaDevices.getUserMedia({ video: true, audio: true
})
            .then(stream => {
                localVideo.srcObject = stream;
                stream.getTracks().forEach(track =>
peerConnection.addTrack(track, stream));
            });

        // Handle WebSocket messages (signaling)
        ws.onmessage = async (event) => {
            const { type, data } = JSON.parse(event.data);

            if (type === 'offer') {
                await peerConnection.setRemoteDescription(new
RTCSessionDescription(data));
                const answer = await peerConnection.createAnswer();
                await peerConnection.setLocalDescription(answer);
                ws.send(JSON.stringify({ type: 'answer', data: answer
}));
            } else if (type === 'answer') {
                await peerConnection.setRemoteDescription(new
RTCSessionDescription(data));
            } else if (type === 'ice-candidate') {
                await peerConnection.addIceCandidate(new
RTCIceCandidate(data));
            }
        };

        // Send ICE candidates to the other peer
        peerConnection.onicecandidate = (event) => {
            if (event.candidate) {
                ws.send(JSON.stringify({ type: 'ice-candidate', data:
event.candidate }));
```

```
            }
        };

        // Display remote stream
        peerConnection.ontrack = (event) => {
            remoteVideo.srcObject = event.streams[0];
        };

        // Start call and send offer
        startCallButton.onclick = async () => {
            const offer = await peerConnection.createOffer();
            await peerConnection.setLocalDescription(offer);
            ws.send(JSON.stringify({ type: 'offer', data: offer }));
        };
    </script>
</body>
</html>
```

CMakeLists.txt :

```
cmake_minimum_required(VERSION 3.10)

# Project Name
project(webrtc)

# Set C++ Standard
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

set(VCPKG_DIR "C:/vcpkg/vcpkg")

# Include vcpkg toolchain file if it exists
if(EXISTS "${VCPKG_DIR}/scripts/buildsystems/vcpkg.cmake")
    set(CMAKE_TOOLCHAIN_FILE
"${VCPKG_DIR}/scripts/buildsystems/vcpkg.cmake")
endif()

# Detect Operating System
if (WIN32)
    message(STATUS "Configuring for Windows")
    include_directories("${VCPKG_DIR}/installed/x64-windows/include")
elseif (UNIX)
    message(STATUS "Configuring for Linux")
    # Include directories for uSockets and uWebSockets
    include_directories(/usr/local/include/uWebSockets)
else()
    message(FATAL_ERROR "Unsupported OS")
endif()

# Add Executable
add_executable(webrtc server.cpp)



# Link Libraries
if (WIN32)
target_link_libraries(webrtc PRIVATE
"${VCPKG_DIR}/installed/x64-windows/lib/libssl.lib"
"${VCPKG_DIR}/installed/x64-windows/lib/libcrypto.lib"
"${VCPKG_DIR}/installed/x64-windows/lib/uSockets.lib"
"${VCPKG_DIR}/installed/x64-windows/lib/zlib.lib"
)
elseif (UNIX)
    # Linking against uWebSockets, OpenSSL, Zlib, and pthread
    target_link_libraries(webrtc PRIVATE
        -l:uSockets.a
        ssl
        crypto
        z
        pthread
    )
endif()
```

If you want to explicitly provide the STUN server in the above code(change in index.html code) :

```
const configuration = {
    iceServers: [
        { urls: 'stun:stun.l.google.com:19302' }  // Google's public
STUN server
    ]
};

// Create the RTCPeerConnection with the STUN server configuration
const peerConnection = new RTCPeerConnection(configuration);
```