What functionalities should I add for my http server in c

Nice! Building an HTTP server in C is a great project. Here's a list of functionalities you can consider implementing, from basic to advanced:

Basic Functionalities

1. **Handle GET Requests**

   Serve static files like HTML, CSS, JS, images, etc.

2. **Parse HTTP Headers**

   Extract headers like `Host`, `User-Agent`, `Content-Type`, etc.

3. **Return Proper HTTP Status Codes**

   e.g., `200 OK`, `404 Not Found`, `500 Internal Server Error`.

4. **Multithreading or Multiprocessing**

   Handle multiple clients using threads or `fork()`.

5. **Logging**

   Log requests with method, path, status code, and timestamp.

Intermediate Functionalities

1. **POST Request Handling**

   Accept form data or JSON payloads.

2. **MIME Type Detection**

   Send the appropriate `Content-Type` based on file extension.

3. **Directory Listing**

   Optionally list files if a directory is requested.

4. **URL Decoding**

   Decode URL-encoded paths and parameters.

5. **Simple Routing**

   Map paths like `/api/hello` or `/user/profile` to handler functions.

Advanced Functionalities

1. **Persistent Connections (Keep-Alive)**

   Keep the connection open for multiple requests.

2. **HTTP/1.1 Compliance**

   Handle chunked transfer encoding, Host headers, etc.

3. **CGI Support**

   Run external programs/scripts and return their output.

4. **TLS/SSL Support**

   Add HTTPS using libraries like OpenSSL.

5. **Authentication Support**

   Basic Auth or token-based header checking.

6. **Rate Limiting**

   Prevent abuse by limiting the number of requests per IP.

7. **Cache Control**

   Add support for `ETag`, `Last-Modified`, and `Cache-Control`.

Want suggestions based on your current progress or specific use cases (e.g., serving a web app or API)?

Is creating an http server in c a good projext

Finished reasoning

Absolutely—a custom HTTP server in C is an excellent project, especially if you're looking to deepen your understanding of:

1. **Networking Fundamentals**

   You'll work directly with sockets (BSD sockets API), learn how TCP connections are established (the three-way handshake), and handle low-level I/O.

2. **Protocol Design and Parsing**

   You'll implement the HTTP protocol yourself—parsing request lines and headers, formatting responses, handling status codes, and managing persistent (keep-alive) versus non-persistent connections.

3. **Concurrency Models**

   You can experiment with different ways to handle multiple clients:

   4. **Forking** a new process per connection

   5. **Thread pools**

   6. **Non-blocking I/O** with `select()`/`poll()`/`epoll()`

      Each approach has trade-offs in complexity, performance, and resource usage.

7. **Memory Management and Security**

   Writing a robust server in C forces you to manage buffers carefully, avoid buffer overflows, and think about edge-case inputs (e.g., very long headers or malformed requests).

8. **Extensibility**

   Once the basics are in place, you can layer on features like:

9. Serving static files with correct MIME types

10. Simple routing or a CGI interface for dynamic content

11. TLS/SSL using OpenSSL for HTTPS support

12. Caching headers (`ETag`, `Last-Modified`) and compression (gzip)

13. **Performance Tuning**

    C gives you fine-grained control. You can measure and optimize latency, throughput, and resource usage (memory, file descriptors).

Potential Challenges

- **Complexity**: HTTP/1.1 has a lot of corner cases (chunked encoding, connection reuse).
- **Security**: You'll need to be vigilant about buffer overflows, injection attacks, and properly validating client input.
- **Maintenance**: Without higher-level abstractions, adding new features can get messy unless you structure your code cleanly.

Why It’s Worthwhile

- **Deep Learning**: You’ll come away with a far better grasp of how web servers (like Nginx or Apache) actually work under the hood.

- **Portfolio Impact**: A home-grown HTTP server is an impressive showcase of low-level systems skills.

- **Extensible Base**: You can turn your server into a sandbox for other protocols (WebSocket, HTTP/2) or custom embedded-device applications.

If you enjoy systems programming and want a project that scales in complexity as you add features, building an HTTP server in C is both challenging and rewarding. Go for it!

Why do most http requests are 1.1 even when http 3 has been invented

Great question—and it shows you're thinking like a systems-level dev. So here’s **why HTTP/1.1 still dominates**, even though HTTP/2 and HTTP/3 exist:

---

1. Massive Backward Compatibility

- **HTTP/1.1 has been around since 1997**, and nearly all browsers, servers, proxies, CDNs, and IoT devices support it.

- It's the "lowest common denominator" that **everything understands**—so it’s the safe default if there's any uncertainty about the client's capabilities.

---

2. Middleboxes and Proxies

- Routers, corporate firewalls, and load balancers often **assume and parse HTTP/1.1**.
- Many proxies **break or block** newer protocols like HTTP/2 or QUIC (which HTTP/3 is based on), especially in enterprise or ISP networks.
- This makes fallback to 1.1 essential for reliability.

---

3. HTTP/3 Uses UDP (QUIC)

- HTTP/3 is based on QUIC, which runs over **UDP**, not TCP.
- A lot of infrastructure (firewalls, NATs, etc.) is still **optimized for TCP and might restrict or throttle UDP traffic**, especially on port 443.
- Some ISPs or enterprise networks may even **block UDP altogether**, forcing a fallback to HTTP/1.1 or 2.

---

4. Deployment Complexity

- Implementing HTTP/2 or HTTP/3 often requires **TLS and ALPN (Application-Layer Protocol Negotiation)**.
- Not every server is updated or configured to support these protocols. Especially on shared hosting or older infrastructure, HTTP/1.1 is just easier to maintain.

---

5. Client-Driven Negotiation

- Browsers like Chrome, Firefox, etc. **attempt HTTP/2 or 3**, but if the server or network can’t handle it, they **gracefully fall back to 1.1**.
- So from the server’s perspective, **most requests might still come in as HTTP/1.1** unless both ends are fully modernized.

---

6. Content Doesn’t Always Need Speed

- For simple HTML/CSS sites or APIs that don't stream or multiplex much data, the performance difference between 1.1 and 2/3 **is negligible**.
- Many developers and companies don't feel the need to upgrade yet.

TL;DR

HTTP/1.1 is still everywhere because it's universally supported, reliable in all environments, and newer protocols face adoption challenges with infrastructure, compatibility, and complexity.

Want to try setting up HTTP/2 or 3 support in your own C server next? It's totally doable (especially with libraries like `nghttp2` or `lsquic`).

Why does http 3 use udp. Explain difference bw versions of http in detail

Finished reasoning

Below is a deep dive into why HTTP/3 moved to UDP (via QUIC) and how each major HTTP version differs under the hood.

Evolution of HTTP Versions

Feature / Version	HTTP/0.9	HTTP/1.0	HTTP/1.1	HTTP/2	HTTP/3 (QUIC)
Year Introduced	1991	1996	1997	2015	2022
Transport Layer	TCP (single request)	TCP (one request/conn)	TCP (persistent connections)	TCP (multiplexed streams)	UDP (QUIC multiplexed)
Connection Model	One request, one TCP	One request per conn	Keep-alive, pipelining	Single TCP conn, streams	Single UDP "conn," streams
Header Encoding	None	Plain-text headers	Plain-text headers	HPACK (binary, compressed)	QPACK (improved compression)
Multiplexing	No	No	Pipelining (head-of-line)	True multiplexing	True multiplexing
TLS Support	N/A	Optional	Optional, widely used	Mandatory in browsers	Mandatory
Flow Control	No	No	No	Stream- & connection-level	Stream- & connection-level
Prioritization	No	No	No	Yes	Yes

Feature / Version	HTTP/0.9	HTTP/1.0	HTTP/1.1	HTTP/2	HTTP/3 (QUIC)
Binary Framing	No	No	No	Yes	Yes

Key Differences and Motivations

1. HTTP/0.9 → HTTP/1.0

- **HTTP/0.9** was a toy protocol: client sent `GET /path\n`, server returned raw HTML, then closed.

- **HTTP/1.0** standardized requests and responses with versioned status lines and headers, but still opened a new TCP connection per request.

2. HTTP/1.0 → HTTP/1.1

- **Persistent Connections**: HTTP/1.1 introduced `Connection: keep-alive` so a single TCP connection could serve multiple requests, cutting TCP handshake overhead.

- **Pipelining**: Clients could send multiple requests back-to-back without waiting for responses, but responses had to arrive in order. This created the **head-of-line blocking** problem: one slow response held up all later ones.

- **Chunked Transfer Encoding**, additional cache controls, more robust error codes, and host headers (virtual hosting) rounded out the spec.

3. HTTP/1.1 → HTTP/2

- **Binary Framing Layer**: HTTP/2 moved from plain-text to a binary protocol (frames), enabling more efficient parsing.
- **HPACK Compression**: Headers are compressed to reduce repeated header bloat.
- **Multiplexing**: Multiple independent "streams" can coexist over one TCP connection, eliminating head-of-line blocking *within* HTTP.
- **Prioritization**: Streams can be prioritized so, e.g., CSS loads before images if needed.

But… TCP's own head-of-line blocking remains: if a TCP packet is lost, *all* streams stall until that packet is retransmitted and received.

4. HTTP/2 → HTTP/3 (QUIC over UDP)

- **Why UDP?**
    - **Avoid TCP HOL Blocking**: QUIC implements its own framing and reliability on top of UDP, so packet loss on one stream only stalls that stream, not *all* streams on the connection.
    - **Faster Connection Establishment**: QUIC combines the TCP three-way handshake and TLS handshake into a single "0-RTT" or "1-RTT" process, shaving round trips.
    - **Built-In Encryption**: QUIC integrates TLS 1.3 directly, so every connection is encrypted by default.
    - **Connection Migration**: Since QUIC connections are identified by a connection ID rather than the 4-tuple of IP:port, clients can roam between networks (Wi-Fi → cellular) without tearing down the connection.

- **QUIC Features**
  - **Multiplexed Streams** with per-stream flow control and independent loss recovery.
  - **0-RTT Resumption**: Repeat clients can start sending data immediately using cached session keys.
  - **Path MTU Discovery**, packet pacing, and pluggable congestion control algorithms at the user level.

Why TCP Alone Wasn't Enough

1. **Head-of-Line Blocking at TCP Level**

   Even with HTTP/2's multiplexing, a single lost TCP segment blocks *all* streams until recovery.

2. **Handshake Latency**

   Traditional TCP + TLS requires at least two round trips (TCP SYN/SYN-ACK + TLS ClientHello/ServerHello). QUIC merges these.

3. **Deployment Flexibility**

   QUIC's user-space implementation means rapid iteration and feature rollout, without waiting for kernel updates.

Summary

- **HTTP/1.1** (over TCP): ubiquitous, simple, but suffers from HOL blocking and per-request overhead.
- **HTTP/2** (over TCP): binary, multiplexed, header-compressed—but still bound by TCP's recovery semantics.
- **HTTP/3** (over UDP + QUIC): multiplexed with independent reliability, faster handshakes, mandatory encryption, and connection migration.

By moving to UDP and implementing reliability in user space, HTTP/3 (QUIC) sidesteps long-standing TCP limitations and lays the groundwork for a more responsive, flexible web.

I think quic also supports server side events

Finished reasoning

Server-Side Events (SSE) is a pattern built on top of HTTP's streaming capabilities: the client opens a long-lived HTTP request (usually a GET to an endpoint with `Accept: text/event-stream`), and the server keeps that connection open, pushing newline-delimited "event" data whenever it has updates.

Because SSE lives entirely at the HTTP layer, **any transport that fully implements HTTP semantics can carry SSE**—that includes HTTP/1.1, HTTP/2, and HTTP/3 (QUIC). The differences are all underneath:

1. **Over HTTP/1.1 (TCP):**
    2. A single TCP connection carries the SSE stream.
    3. If a TCP segment is lost, the entire stream stalls until recovery.

4. **Over HTTP/2 (TCP):**
   5. SSE is just one HTTP/2 stream within the binary-framed connection.
   6. Other streams (e.g., resource loads) can continue even if the SSE stream stalls, but a lost TCP packet still blocks all streams on that connection until it's retransmitted.
7. **Over HTTP/3 (QUIC/UDP):**
   8. SSE is one QUIC stream.
   9. Packet loss only stalls that one stream; other streams (and control frames) proceed independently.
   10. The lower handshake latency and built-in 0-RTT mean you can reestablish SSE connections faster after a network change or reconnection.
   11. Connection migration (e.g., switching from Wi-Fi to cellular) can keep the SSE stream alive without a full reconnect.

So yes—you can absolutely implement SSE endpoints over HTTP/3. From your server's point of view, you still:

c

```c
// Pseudocode for an SSE handler
send_response_header("HTTP/3 200 OK");
send_header("Content-Type", "text/event-stream");
send_header("Cache-Control", "no-cache");
send_header("Connection", "keep-alive");
end_headers();

// Then in a loop, whenever there's an update:
while (client_is_connected) {
    write("data: { \"time\": 123456789 }\n\n");
    flush();
    sleep(1);
}
```

Under the hood your QUIC library (e.g. lsquic or quiche) handles splitting those bytes into QUIC frames, retransmitting lost packets per-stream, and keeping the connection alive across network changes.

Key takeaway: SSE isn't a new transport protocol—it's just a convention over HTTP. Since HTTP/3 fully implements HTTP semantics on top of QUIC, SSE works just as well (and more robustly) over HTTP/3 as it did over HTTP/1.1 or HTTP/2.

Are these the only difference between http 3 quic and the rest

Finished reasoning

No—there's more under the hood in QUIC/HTTP-3 beyond just "UDP + independent streams." Here are several additional key differences and enhancements:

---

1. Connection Identifiers

- **QUIC uses a Connection ID** that's independent of IP/port tuples.
- This lets endpoints **migrate** (e.g. Wi-Fi → LTE) without tearing down the session—something TCP can't do without a full reconnect.

2. Version Negotiation

- QUIC packets include a **version field**, so client and server can negotiate or upgrade the QUIC version mid-session.
- You don't need to fall back to TCP if you don't support a given QUIC version—just negotiate to one you both do.

3. Pluggable Congestion Control & Loss Recovery

- QUIC's loss detection and congestion control live in **user space** (your library), not the kernel.
- You can choose or tune algorithms (e.g., BBR, CUBIC, Reno) per-connection without waiting for OS updates.

4. Stream Prioritization & Flow Control Granularity

- QUIC supports **per-stream flow control windows**, so a fast producer can't overwhelm a slow consumer on a different stream.
- HTTP/3's QPACK (its header-compression) is designed to avoid head-of-line blocking between header blocks and data frames.

5. Built-In Multipath (Future)

- The QUIC spec is designed to allow **multipath** (sending packets over multiple network paths simultaneously) in a way TCP isn't.
- While not universally implemented yet, it's in the QUIC roadmap.

6. Cryptographic Agility & Integration

- QUIC integrates **TLS 1.3** directly into its handshake, not layered on top.
- This tight coupling lets QUIC negotiate crypto parameters and establish encryption keys in fewer round trips and swap out crypto primitives more easily.

7. Packet Framing & Header Formats

- QUIC packets have a **self-describing header** (with packet number length, flags, etc.) that makes parsing more flexible.
- There are **Short** and **Long** header formats for different stages of the handshake, improving efficiency.

8. Stateless Retry & DoS Mitigation

- QUIC servers can send a **stateless retry** during the handshake (using a cookie) to verify a client's IP before allocating state, helping mitigate amplification and DoS attacks.

In short, HTTP/3 over QUIC isn't just "HTTP/2 on UDP." QUIC rethinks connection management, reliability, security, and transport-layer extensibility—all in user space—while HTTP/3 layers its framing and header compression on top. This combination yields faster handshakes, more robust loss recovery, seamless migration, and a platform for future transport innovations (like multipath).

Since http 3 used udp does it have socket dgram in its code. How do I implement http server over udp

Finished reasoning

Because HTTP/3 (over QUIC) runs atop UDP, your "socket" calls will indeed use datagram sockets (`SOCK_DGRAM`) at the OS level. However, you **won't** simply read/write HTTP messages on that socket as you would with TCP+HTTP/1.x. Instead, you embed your HTTP/3 logic inside a QUIC implementation, which itself uses a UDP socket for packet I/O.

Here's the high-level approach to building an HTTP/3 server in C:

---

1. Create and bind a UDP socket

c

```c
int fd = socket(AF_INET6, SOCK_DGRAM, 0);
// or AF_INET for IPv4
struct sockaddr_in6 addr = { .sin6_family = AF_INET6,
                             .sin6_port   = htons(443),
                             .sin6_addr   = in6addr_any };
bind(fd, (struct sockaddr*)&addr, sizeof(addr));
```

This socket will receive and send QUIC packets.

---

2. Integrate a QUIC library

Implementing QUIC from scratch is extremely complex. Instead, pick a battle-tested library:

- **quiche** (Cloudflare)
- **lsquic** (LiteSpeed)
- **ngtcp2** + **nghttp3** (IETF reference)

These libraries will:

1. **Parse incoming UDP datagrams** into QUIC packets
2. **Perform the QUIC handshake** (including TLS 1.3 crypto)
3. **Demultiplex QUIC streams**, delivering ordered byte-streams to you
4. **Handle retransmissions, congestion control, flow control**
5. **Expose an API** where you can read/write on individual QUIC streams

---

3. Implement the HTTP/3 layer

Once you have QUIC streams, you layer on HTTP/3:

1. **Initialize an HTTP/3 connection context** with your QUIC session.
2. **On a new incoming stream**, parse the HTTP/3 frame sequence (HEADERS, DATA, etc.) using your library's HTTP/3 API.
3. **Dispatch request headers** to your request handler.
4. **Write response frames** (HEADERS frame + DATA frames) back into that same stream.

Each QUIC stream maps to one HTTP request/response exchange (or can be reused for push streams).

---

4. Event loop

Because UDP is message-oriented and QUIC is asynchronous, you'll typically run an event loop that:

1. **Waits on your UDP socket** (e.g., with `epoll`/`select`/`poll`).
2. **Reads a packet**, passes it into the QUIC library (e.g., `quiche_conn_recv()`).
3. **Drains any readable QUIC streams** (e.g., `quiche_conn_stream_recv()`) and handles HTTP/3 frames.
4. **Generates outgoing QUIC packets** (e.g., `quiche_conn_send()`) and `sendto()` them on the UDP socket.
5. **Handles timers** for retransmissions and connection timeouts.

---

5. Example pseudocode sketch

c

```c
// 1. Setup UDP socket `fd` and QUIC config/context
quiche_config *config = quiche_config_new(QUIC_VERSION);
quiche_config_load_cert_chain_from_pem_file(config, "cert.pem");
quiche_config_load_priv_key_from_pem_file(config, "key.pem");

while (1) {
    // 2. Wait for events on `fd` (readable, writable, timeout)
    int n = recvfrom(fd, buf, sizeof(buf), 0, &peer_addr, &peer_len);
```

```
    if (n > 0) {
        // 3. Pass packet into QUIC
        quiche_conn *conn = quiche_accept(...);
        quiche_conn_recv(conn, buf, n);
    }

    // 4. Process all streams
    quiche_stream_iter it;
    while (quiche_conn_stream_iter(conn, &it)) {
        uint64_t stream_id;
        ssize_t read = quiche_conn_stream_recv(conn, &stream_id, buf, sizeof(buf), 
        if (read > 0) {
            // parse HTTP/3 HEADERS/DATA frames from `buf`
            handle_http3_frames(conn, stream_id, buf, read);
        }
    }

    // 5. Generate outgoing QUIC packets
    uint8_t out[1500];
    ssize_t sent = quiche_conn_send(conn, out, sizeof(out));
    if (sent > 0) {
        sendto(fd, out, sent, 0, (struct sockaddr*)&peer_addr, peer_len);
    }

    // 6. Drive timers (e.g., retransmit)
    quiche_conn_on_timeout(conn);
}
```

Why you don’t “just” use `recvfrom`/`sendto` for HTTP

- QUIC packets include framing, encryption, packet numbers, congestion signals, etc.
- HTTP/3 frames ride **inside** QUIC streams.
- All of the complexity (TLS, retransmits, stream multiplexing) is handled by your QUIC library.

Next steps

1. **Pick a QUIC library** (e.g., quiche is C-friendly and well documented).
2. **Follow its examples**—they almost always include an HTTP/3 “server” demo.
3. **Layer your application logic** on top of their stream-read/stream-write APIs.

This approach lets you focus on your HTTP semantics and leave the transport-layer details to a mature QUIC implementation.

How to handle packet loss in udp for http server

Finished reasoning

When you build an HTTP/3 server over UDP, you don't handle packet loss yourself at the application level—instead you rely on QUIC's built-in reliability mechanisms. Here's how QUIC (and thus your HTTP/3 server) copes with lost UDP datagrams:

1. **Packet Numbers & ACKs**

   2. Every QUIC packet carries a unique packet number.

   3. Receivers periodically send ACK frames listing which packet numbers they've received.

4. **Retransmission Timers**

   5. When you send a packet, the QUIC library starts a timer.

   6. If no ACK arrives for that packet before the timer expires, the library automatically retransmits it.

7. **Per-Stream Loss Recovery**

   8. QUIC tracks which streams' data was acknowledged.

   9. Only the lost stream's data is resent; other streams continue unimpeded.

10

Congestion Control

11

QUIC implements congestion algorithms (e.g., CUBIC, BBR) in user space.

12

When it detects loss, it reduces its sending rate to avoid overwhelming the network.

13

Adaptive Timeouts

14

QUIC continually measures round-trip time (RTT) and adjusts its retransmission timeout (RTO).

15

This keeps loss recovery responsive but not overly aggressive.

In Practice

If you're using a library like **quiche**, **lsquic**, or **ngtcp2**, you simply:

1

Feed incoming UDP datagrams into the library (`quiche_conn_recv()`).

2

Call its "send" function (`quiche_conn_send()`) whenever it has packets queued (including retransmissions).

3

Let the library's event loop drive timers (e.g., `quiche_conn_on_timeout()`), which triggers retransmits when needed.

All of the complexity—tracking packet numbers, scheduling retransmissions, running congestion control, and delivering in-order byte streams to you—is encapsulated in the QUIC implementation. Your job is simply to read from and write to the QUIC streams; the library ensures reliable delivery over the unreliable UDP socket.

How does https differ from https in both theory and code..how can I write an https server in c

Finished reasoning

Below is a two-part deep dive:

1. **Theory**: What HTTPS adds on top of HTTP
2. **Code**: Sketch of an HTTPS server in C using OpenSSL

---

1. Theory: HTTP vs HTTPS

Aspect	HTTP (Hypertext Transfer Protocol)	HTTPS (HTTP over TLS)
Transport	Plain-text over TCP	Encrypted over TCP via TLS
Security Goals	None	Confidentiality, Integrity, Authentication
Handshake	TCP 3-way handshake only	TCP 3-way + TLS handshake (X.509 cert exchange, key derivation)
Port	80	443 (by convention)
Data Protection	Vulnerable to eavesdropping and tampering	Protected by symmetric encryption (AES, ChaCha20), MAC/AEAD
Certificate	N/A	Server (and optionally client) presents X.509 certificate
Library Support	BSD sockets only	BSD sockets + TLS library (OpenSSL, mbedTLS, wolfSSL, etc.)

What TLS Adds

1. **Authentication**
    2. Server proves its identity by sending a certificate signed by a trusted CA.

3. (Optional) Client can present a certificate for mutual TLS.

4. **Key Exchange**

5. TLS uses asymmetric crypto (e.g. ECDHE) to agree on a shared secret.

6. Protects against passive and active attackers.

7. **Encryption & Integrity**

8. Subsequent application data is encrypted and integrity-protected (AEAD ciphers).

9. **Session Resumption**

10. TLS can resume previous sessions (via session IDs or tickets) to save handshake round trips.

---

2. Code: Minimal HTTPS Server in C with OpenSSL

Below is a simplified outline. Real servers must handle errors, concurrency, renegotiation, and more.

c

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#define PORT 4433
#define CERT_FILE "server.crt"
#define KEY_FILE  "server.key"

// 1. Initialize OpenSSL
void init_openssl()
{
    SSL_load_error_strings();
    OpenSSL_add_ssl_algorithms();
}

// 2. Create SSL_CTX
SSL_CTX* create_context()
{
    const SSL_METHOD *method = TLS_server_method();
    SSL_CTX *ctx = SSL_CTX_new(method);
    if (!ctx) {
        perror("Unable to create SSL context");
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
    }
    return ctx;
}

// 3. Configure context with cert and key
void configure_context(SSL_CTX *ctx)
{
    // Load server certificate
    if (SSL_CTX_use_certificate_file(ctx, CERT_FILE, SSL_FILETYPE_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
    }
    // Load private key
    if (SSL_CTX_use_PrivateKey_file(ctx, KEY_FILE, SSL_FILETYPE_PEM) <= 0 ) {
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
```

```c
    }
}

int main()
{
    int sock;
    struct sockaddr_in addr;

    // Initialize SSL library
    init_openssl();
    SSL_CTX *ctx = create_context();
    configure_context(ctx);

    // 4. Create and bind TCP socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port   = htons(PORT);
    addr.sin_addr.s_addr = INADDR_ANY;
    bind(sock, (struct sockaddr*)&addr, sizeof(addr));
    listen(sock, 1);
    printf("HTTPS server listening on port %d\n", PORT);

    // 5. Accept connections
    while (1) {
        struct sockaddr_in client_addr;
        uint len = sizeof(client_addr);
        int client = accept(sock, (struct sockaddr*)&client_addr, &len);
        if (client < 0) {
            perror("Unable to accept");
            continue;
        }

        // 6. Create SSL object and perform handshake
        SSL *ssl = SSL_new(ctx);
        SSL_set_fd(ssl, client);
        if (SSL_accept(ssl) <= 0) {
            ERR_print_errors_fp(stderr);
        } else {
            // 7. Read HTTP request
            char buf[4096];
            int bytes = SSL_read(ssl, buf, sizeof(buf) - 1);
            if (bytes > 0) {
                buf[bytes] = '\0';
                printf("Received request:\n%s\n", buf);

                // 8. Send HTTP response
                const char *response =
                    "HTTP/1.1 200 OK\r\n"
                    "Content-Type: text/plain\r\n"
                    "Content-Length: 12\r\n"
                    "\r\n"
                    "Hello HTTPS";
                SSL_write(ssl, response, strlen(response));
            }
        }

        // 9. Clean up
        SSL_shutdown(ssl);
        SSL_free(ssl);
        close(client);
    }

    // 10. Tear down
    close(sock);
    SSL_CTX_free(ctx);
    EVP_cleanup();
    return 0;
}
```

Key Points in the Code

1. **OpenSSL Initialization**
   2. `SSL_library_init()` (implicit in modern versions)
   3. `SSL_load_error_strings()` and `OpenSSL_add_ssl_algorithms()`
4. **Context Setup**
   5. `SSL_CTX_new(TLS_server_method())` creates a server-side TLS context supporting TLS 1.2+ and TLS 1.3.
   6. Load your PEM-encoded certificate and private key into the context.
7. **Socket Handling**
   8. Create a normal TCP socket (`SOCK_STREAM`), bind, listen, and accept.
9. **TLS Handshake per Connection**
   10. Wrap the client socket FD in an `SSL*` object with `SSL_new()`.
   11. Call `SSL_accept()` to perform the TLS handshake (including certificate exchange and key derivation).
12. **I/O over SSL**
   13. Use `SSL_read()` / `SSL_write()` instead of `read()` / `write()`. These functions handle decryption/encryption, buffering, and renegotiation.
14. **Cleanup**
   15. `SSL_shutdown()` to send the “close_notify” alert.
   16. Free the `SSL*` and eventually the `SSL_CTX`.

Next Steps & Production Considerations

- **Error Handling**: Check return values at every step and log errors.
- **Concurrency**: Spawn threads or use an event loop (e.g., `select`/`epoll`) to handle multiple clients.
- **Certificate Management**: Automate cert renewal (e.g., Let’s Encrypt).
- **TLS Settings**: Harden your TLS context (disable old ciphers, enable HTTP Strict Transport Security).
- **Performance**: Consider session tickets for 0-RTT resumption and OCSP stapling for faster cert validation.

With this foundation, you’ve bridged the gap between HTTP and HTTPS in both theory and code.