Sistemas Operativos **Ano lectivo 2025/2026**

Network Programming Tutorial[1]

Building HTTP Servers in C

From Socket Basics to Complete Web Server

This tutorial teaches network programming in C from the ground up. You'll build increasingly complex servers, culminating in a complete HTTP web server. Each section builds on the previous one, so work through them in order.

Part 1: Socket Programming Basics

Understanding Sockets

A socket is an endpoint for network communication. Think of it as a telephone - you need to create it, configure it, dial (connect), and then talk (send/receive data).

Function	Purpose
socket()	Create a new socket
bind()	Assign an address (IP + port) to the socket
listen()	Mark socket as passive (ready to accept connections)
accept()	Accept an incoming connection (blocks until client connects)
send()/recv()	Send and receive data
close()	Close the connection

Exercise 1: Echo Server

Let's build the simplest possible server - one that echoes back whatever the client sends.

```
// echo_server.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

[1] The text of this project proposal had AI contributions to its completion.

```
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    // Step 1: Create socket
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Step 2: Set socket options (allow immediate reuse)
    int opt = 1;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    // Step 3: Bind to address
    struct sockaddr_in address;
    address.sin_family = AF_INET;           // IPv4
    address.sin_addr.s_addr = INADDR_ANY;   // All interfaces
    address.sin_port = htons(PORT);         // Port (network byte order)

    if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    // Step 4: Listen for connections
    if (listen(server_fd, 10) < 0) {
        perror("listen failed");
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d...\n", PORT);

    // Step 5: Accept and handle connections
    while (1) {
        int client_fd = accept(server_fd, NULL, NULL);
        if (client_fd < 0) {
            perror("accept failed");
            continue;
        }

        printf("Client connected\n");
```

```
            // Read and echo back
            char buffer[BUFFER_SIZE];
            ssize_t bytes_read = recv(client_fd, buffer, BUFFER_SIZE - 1, 0);
            if (bytes_read > 0) {
                buffer[bytes_read] = '\0';
                printf("Received: %s\n", buffer);
                send(client_fd, buffer, bytes_read, 0);
            }

            close(client_fd);
        }

        close(server_fd);
        return 0;
    }
```

Compile and test:

```
gcc -o echo_server echo_server.c
./echo_server

# In another terminal:
telnet localhost 8080
```

Part 2: HTTP Basics

Understanding HTTP Protocol

HTTP is a text-based protocol. The client sends a request, the server sends a response.

HTTP Request Format

```
GET /index.html HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0
Accept: text/html
```

HTTP Response Format

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1234
Server: MyServer/1.0

<html>...</html>
```

Exercise 2: Minimal HTTP Server

Build a server that responds with a simple HTML page.

```
// simple_http.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

const char* response =
    "HTTP/1.1 200 OK\r\n"
    "Content-Type: text/html\r\n"
    "\r\n"
    "<html><body><h1>Hello, World!</h1></body></html>";

int main() {
    int server_fd = socket(AF_INET, SOCK_STREAM, 0);
    int opt = 1;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    struct sockaddr_in addr = {0};
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(8080);
```

```
        bind(server_fd, (struct sockaddr*)&addr, sizeof(addr));
        listen(server_fd, 10);

        printf("HTTP server running on http://localhost:8080\n");

        while (1) {
            int client_fd = accept(server_fd, NULL, NULL);

            char buffer[4096];
            recv(client_fd, buffer, sizeof(buffer), 0);

            send(client_fd, response, strlen(response), 0);
            close(client_fd);
        }
        return 0;
    }
```

Test it:

```
gcc -o simple_http simple_http.c
./simple_http
# Open browser: http://localhost:8080
```

Part 3: File Serving

Exercise 3: Static File Server

Parse the request path and serve actual files from disk.

```
// file_server.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

void send_file(int client_fd, const char* path) {
    // Open file
    FILE* file = fopen(path, "rb");
    if (!file) {
        // Send 404
        const char* response =
            "HTTP/1.1 404 Not Found\r\n"
            "Content-Type: text/html\r\n"
            "\r\n"
            "<h1>404 Not Found</h1>";
        send(client_fd, response, strlen(response), 0);
        return;
    }

    // Get file size
    fseek(file, 0, SEEK_END);
    long file_size = ftell(file);
    fseek(file, 0, SEEK_SET);

    // Send headers
    char header[512];
    snprintf(header, sizeof(header),
        "HTTP/1.1 200 OK\r\n"
        "Content-Type: text/html\r\n"
        "Content-Length: %ld\r\n"
        "\r\n", file_size);
    send(client_fd, header, strlen(header), 0);

    // Send file content
    char buffer[4096];
    size_t bytes;
    while ((bytes = fread(buffer, 1, sizeof(buffer), file)) > 0) {
        send(client_fd, buffer, bytes, 0);
    }
```

```
        fclose(file);
    }
```

Part 4: Multi-Threading

Why Threading?

Our current server handles one client at a time. While serving one client, others must wait. Threading allows concurrent handling of multiple clients.

Exercise 4: Threaded Server

```
// threaded_server.c
#include <pthread.h>

void* handle_client(void* arg) {
    int client_fd = *(int*)arg;
    free(arg);

    // Read request, send response
    char buffer[4096];
    recv(client_fd, buffer, sizeof(buffer), 0);

    const char* response = "HTTP/1.1 200 OK\r\n\r\n<h1>Hello</h1>";
    send(client_fd, response, strlen(response), 0);

    close(client_fd);
    return NULL;
}

int main() {
    // ... socket setup as before ...

    while (1) {
        int* client_fd = malloc(sizeof(int));
        *client_fd = accept(server_fd, NULL, NULL);

        pthread_t thread;
        pthread_create(&thread, NULL, handle_client, client_fd);
        pthread_detach(thread);  // Don't wait for thread to finish
    }
    return 0;
}
```

Compile with pthread:

```
gcc -pthread -o threaded_server threaded_server.c
```

Key Concepts Summary

Common Pitfalls

- **Not checking return values:** Every system call can fail
- **Buffer overflows:** Always leave room for null terminator
- **Not handling partial reads/writes:** recv/send may return less than requested
- **Memory leaks:** Free allocated memory, close file descriptors
- **Byte order:** Use htons/ntohs for port numbers

Testing Tools

```
# Test with curl
curl -v http://localhost:8080

# Load testing with Apache Bench
ab -n 1000 -c 50 http://localhost:8080/

# Monitor connections
netstat -an | grep 8080
```

Next Steps

You now understand the basics of network programming. For your project, you'll need to add:

- Process architecture (master-worker with fork)
- Thread pools (reuse threads instead of creating new ones)
- Synchronization (semaphores, mutexes, condition variables)
- Shared memory for IPC
- Full HTTP parsing (handle different methods, headers, status codes)
- Proper MIME type detection, logging, statistics

Practice these examples before starting your project!

For complete project templates and advanced topics, refer to the Code Templates document.