TCP Client-Server Stack Implementation

COSC 414 - Professor Trivedi

Department of Computer Science: Operating Systems

Ihab Ashkar

Chibueze Eburuoh

Xannia Simpson

Content

Project Overview

This project implements a simplified client–server communication model to demonstrate how data moves through a basic network stack in a UNIX-based operating system. Using TCP sockets, the system shows how processes exchange messages through network-based inter-process communication (IPC). Although minimal by design, the implementation highlights key networking concepts such as socket creation, binding, listening, connecting, and data transmission—mirroring the essential behavior of real OS-level network layers.

System Architecture

Client to Server Communication

1. Client creates socket and requests connection.
2. Server listens on port 8080 using socket(), bind(), and listen().
3. Server accepts client and creates dedicated connection socket.
4. Client sends message, server processes, and server responds.
5. Both sides close connection.

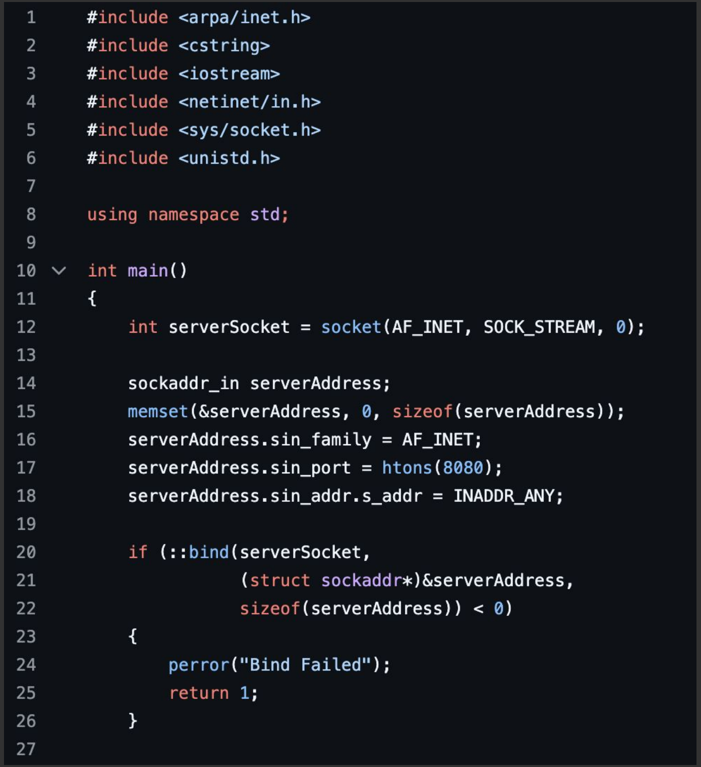
Files

- server.cpp handles socket setup, client handling, and the message response
- client.cpp creates the connection, sends the connection request, and prints the server reply.
- Our Makefile simply automates compilation for both files, rather than manually compiling each.

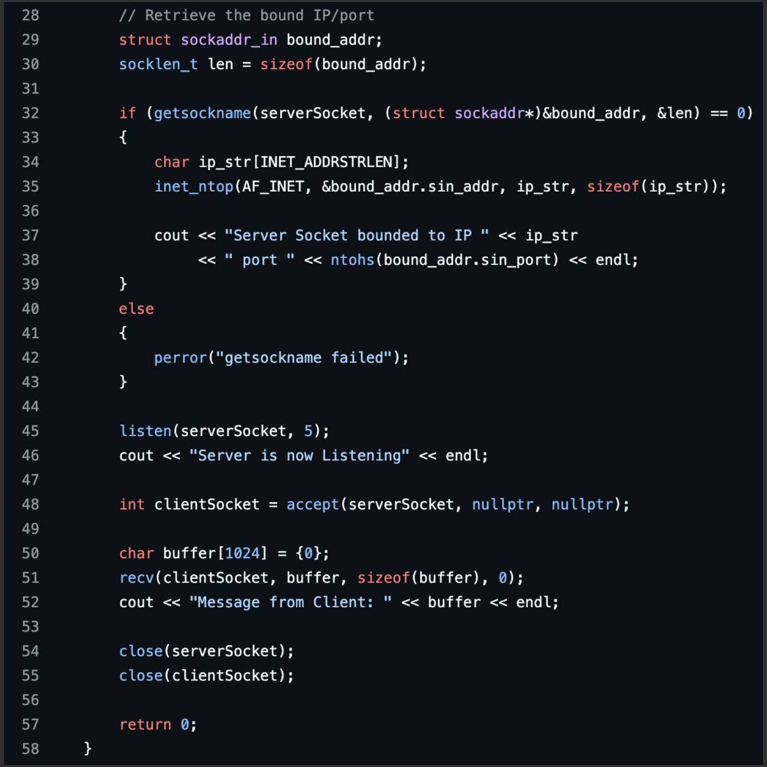
Key Implementation Details

- **Server Setup:**
  - Create TCP socket
  - Bind to a port
  - Listen for a connection request
- **Client Handling:**
  - Accept a client socket
  - Use **recv()** (receive) and **send()** for communication
  - Loop until client disconnects
- **Client Operations:**
  - Create socket
  - Connect to server using **connect()**
  - Send message and wait for server reply

Server Implementation

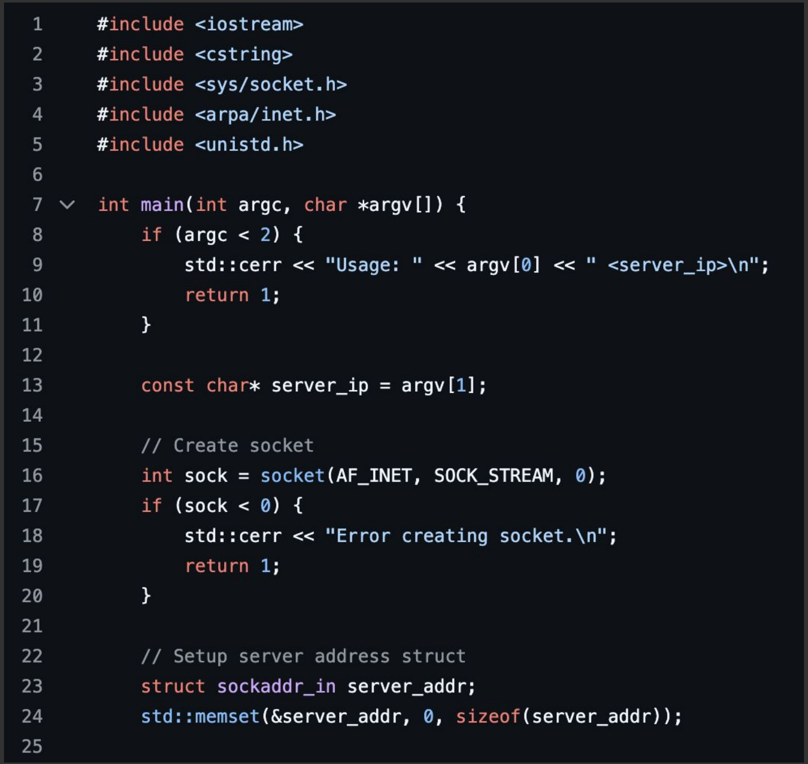


```cpp
#include <arpa/inet.h>
#include <cstring>
#include <iostream>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>

using namespace std;

int main()
{
    int serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    sockaddr_in serverAddress;
    memset(&serverAddress, 0, sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(8080);
    serverAddress.sin_addr.s_addr = INADDR_ANY;

    if (::bind(serverSocket,
               (struct sockaddr*)&serverAddress,
               sizeof(serverAddress)) < 0)
    {
        perror("Bind Failed");
        return 1;
    }

```

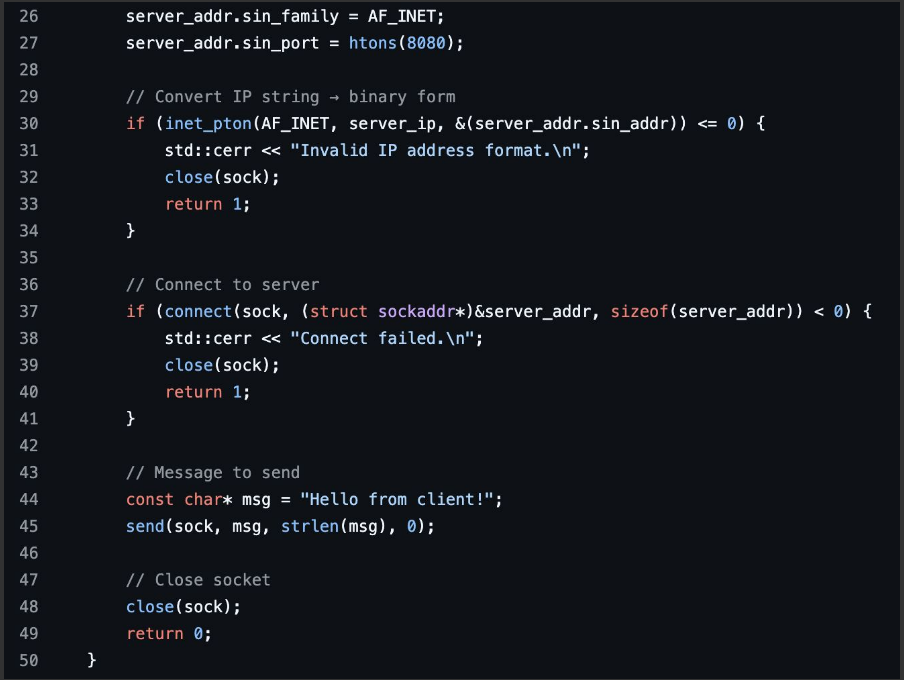


```cpp
    // Retrieve the bound IP/port
    struct sockaddr_in bound_addr;
    socklen_t len = sizeof(bound_addr);

    if (getsockname(serverSocket, (struct sockaddr*)&bound_addr, &len) == 0)
    {
        char ip_str[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &bound_addr.sin_addr, ip_str, sizeof(ip_str));

        cout << "Server Socket bounded to IP " << ip_str
             << " port " << ntohs(bound_addr.sin_port) << endl;
    }
    else
    {
        perror("getsockname failed");
    }

    listen(serverSocket, 5);
    cout << "Server is now Listening" << endl;

    int clientSocket = accept(serverSocket, nullptr, nullptr);

    char buffer[1024] = {0};
    recv(clientSocket, buffer, sizeof(buffer), 0);
    cout << "Message from Client: " << buffer << endl;

    close(serverSocket);
    close(clientSocket);

    return 0;
}
```

Client Implementation



```cpp
#include <iostream>
#include <cstring>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <server_ip>\n";
        return 1;
    }

    const char* server_ip = argv[1];

    // Create socket
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        std::cerr << "Error creating socket.\n";
        return 1;
    }

    // Setup server address struct
    struct sockaddr_in server_addr;
    std::memset(&server_addr, 0, sizeof(server_addr));

```



```cpp
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(8080);

    // Convert IP string → binary form
    if (inet_pton(AF_INET, server_ip, &(server_addr.sin_addr)) <= 0) {
        std::cerr << "Invalid IP address format.\n";
        close(sock);
        return 1;
    }

    // Connect to server
    if (connect(sock, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        std::cerr << "Connect failed.\n";
        close(sock);
        return 1;
    }

    // Message to send
    const char* msg = "Hello from client!";
    send(sock, msg, strlen(msg), 0);

    // Close socket
    close(sock);
    return 0;
}
```

Build Process

Function Calls:

all - compiles all the programs concurrently

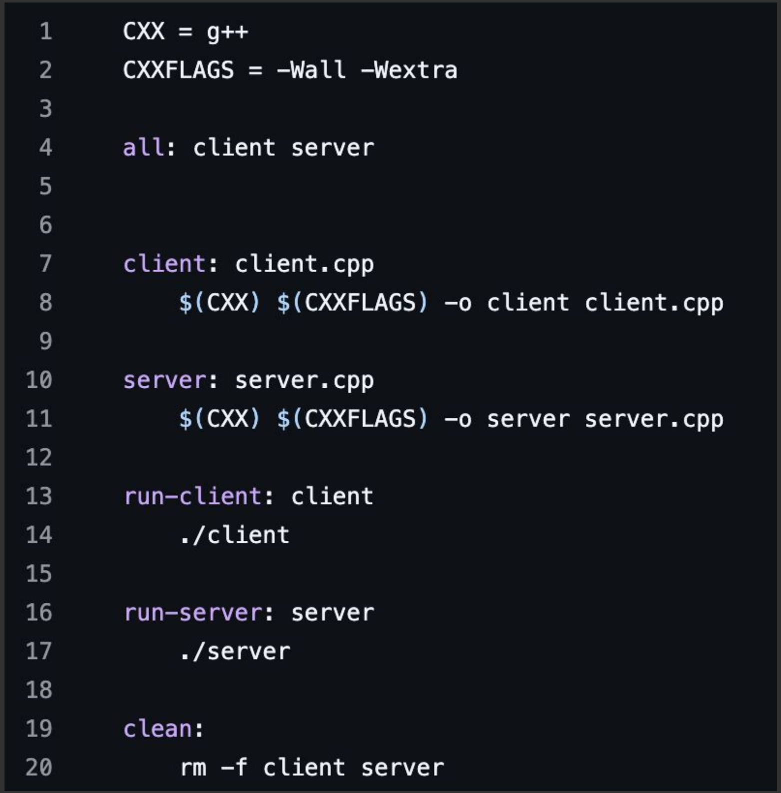
server - builds the server executable

client - builds the client executable

clean - removes compiled binaries

Purpose:

- The Makefile automates compilation.
- Ensures consistent build commands.
- Streamlines process to run the program.



```
CXX = g++
CXXFLAGS = -Wall -Wextra

all: client server



client: client.cpp
	$(CXX) $(CXXFLAGS) -o client client.cpp

server: server.cpp
	$(CXX) $(CXXFLAGS) -o server server.cpp

run-client: client
	./client

run-server: server
	./server

clean:
	rm -f client server
```

OS IPC Breakdown

1

socket()

Requests the OS kernel to create a communication endpoint.

2

bind()

Associates the socket with a local IP address + port,

3

getsockname()

This call queries the kernel for information about the socket's bound address

4

listen(), connect()

Marks socket as server within the kernel; creates backlog queue for incoming connections

Establishes a communication path between a client process and a remote process

5

send(), recv()

Transfers data from one process to another, through the OS's TCP/IP stack

Transfers data from another process (possibly on another machine) into your process.

Future Enhancements

We plan to implement the following improvements as we continue to build on this project:

1 Improved Interprocess Communication

As of now, we have implemented network-based message passing through TCP sockets, rather than using traditional pipes to convey message passing. Implementing pipe-based message passing, which would demonstrate communication across different address spaces

2 Add Multi-Client Support (Concurrency)

Right now, our server can only manage a single client connection at once. To improve scalability, we plan to implement multiprocessing so that each new client triggers a fork(). This will allow a separate child process to handle that client while the main server continues accepting others

3 Error Handling and Logging

Improved error handling and logging would make the system more robust and easier to maintain. Adding structured logs with timestamps and clear error messages would help diagnose issues more effectively. Implementing consistent error-recovery steps would bring the project closer to real-world network server reliability.

Thank you

Questions ?