Victor Hugo Silva
RED ID 826603798
COMPE560 - Spring 2025
Dr. Yusuf Ozturk

REPORT: SOCKET PROGRAMMING

Table of Contents

Introduction

This report documents the development and validation of a simple TCP‑based echo server and client, implemented in C on a Unix environment (SDSU's Jason/Volta servers). This assignment covers two C programs—server.c and client.c—that exchange and reverse text over TCP. All socket and I/O calls are checked via a die() helper that reports errors with perror(). What follows are the complete source listings, execution screenshots, and a detailed function summary.

Summary

The server creates a blocking IPv4 stream socket, binds to a chosen port, and listens with a backlog of five. In its main loop, it calls accept(), wraps the new socket in FILE* streams, then repeatedly reads up to 4096 bytes via fread(), prints that data to its console, reverses it in place with reverse_buffer(), and sends it back with fwrite().

The client creates and connects a socket to the server's IP and port (using inet_pton() and connect()), wraps it in FILE* streams, reads one line from stdin via fgets(), writes it out, and calls shutdown(..., SHUT_WR) to signal end-of-input. It then reads the reversed text in 4096-byte chunks until EOF and prints each chunk to stdout.

The following table contrasts the core behavior of the two programs. The server runs indefinitely, accepting one client at a time, reading and reversing incoming data in fixed-size chunks; the client connects once, sends a single line of input, and prints the server's reversed reply before exiting.

SERVER	**CLIENT**
Creates a blocking IPv4 stream socket (socket(AF_INET, SOCK_STREAM, 0))	Creates a blocking IPv4 stream socket and immediately calls connect()
Binds to INADDR_ANY and the specified port (bind() + listen() with backlog of 5)	No bind/listen—uses connect() to initiate a connection to the server's IP and port
In an infinite loop: accept() a new connection, wrap in FILE* streams for fread/fwrite	Wraps the connected socket in FILE* streams for fwrite (send) and fread (receive)
Repeatedly fread() up to 4096 bytes, print to stdout, reverse in place, then fwrite() back to client	Reads one line from stdin via fgets(), sends it with fwrite(), and calls shutdown()
Cleans up by fclose() on both streams and loops back to accept()	After sending, reads the reversed data in 4096-byte chunks until EOF, then fclose() and exit
Reports any errors immediately via die() → perror() + exit()	Reports any errors immediately via die() → perror() + exit()

Code

server.c

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

```
#include <arpa/inet.h>

/*
*   COMPE560 - Spring 2025
*   Victor Hugo Silva - 826603798
*   Assignment: Introduction to Socket Programming
*
*/


// Buffer size for reading from socket and stdout
#define BUF_SIZE 4096

/**
 * Prints an error message and exits.
 * @param msg  Context string to print before the system error.
 */
void die(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

/**
 * Reverse the contents of a buffer in place.
 * @param buf  Pointer to the data buffer.
 * @param len  Number of bytes in the buffer to reverse.
 */
void reverse_buffer(char *buf, ssize_t len) {
    for (ssize_t i = 0; i < len / 2; ++i) {
        char tmp = buf[i];              // store front element
        buf[i] = buf[len - 1 - i];      // copy back element to front
        buf[len - 1 - i] = tmp;         // restore front element to back
    }
}

/**
 * Main server entry point.
 * Usage: ./server <port>
 * Listens on the given TCP port, accepts a connection,
 * echoes and reverses any incoming data back to the client.
 */

int main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    int port = atoi(argv[1]);  // Convert port argument to integer

    // Create a TCP socket (IPv4)
    int serv_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (serv_sock < 0) {
        die("socket");
    }
```

```
// Prepare server address structure (bind to all interfaces)
struct sockaddr_in addr = {0};
addr.sin_family = AF_INET;               // IPv4
addr.sin_addr.s_addr = htonl(INADDR_ANY); // 0.0.0.0 (all interfaces)
addr.sin_port = htons(port);             // port in network byte order

// Bind the socket to our address and port
if (bind(serv_sock, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
   die("bind");
}

// Start listening, allow up to 5 pending connections
if (listen(serv_sock, 5) < 0) {
   die("listen");
}
printf("Server listening on port %d\n", port);

// Accept and handle clients in a loop
while (1) {
   // accept() blocks until a client connects
   int clnt_sock = accept(serv_sock, NULL, NULL);
   if (clnt_sock < 0) {
      die("accept");
   }
   printf("Client connected\n");

   // Convert socket descriptors to FILE* streams for fread/fwrite
   FILE *in = fdopen(clnt_sock, "r");
   FILE *out = fdopen(dup(clnt_sock), "w");
   if (!in || !out) {
      die("fdopen");
   }

   char buf[BUF_SIZE];
   ssize_t n;

   // Read chunks until EOF (client closed write end)
   while ((n = fread(buf, 1, BUF_SIZE, in)) > 0) {
      // Print received data to server's stdout
      fwrite(buf, 1, n, stdout);
      fflush(stdout);

      // Reverse the data in place
      reverse_buffer(buf, n);

      // Send reversed data back to the client
      fwrite(buf, 1, n, out);
      fflush(out);
   }

   // Clean up this connection
   fclose(in);
   fclose(out);
   printf("Client disconnected\n");
}
```

```
    // Close the listening socket
    close(serv_sock);
    return 0;
}
```

client.c

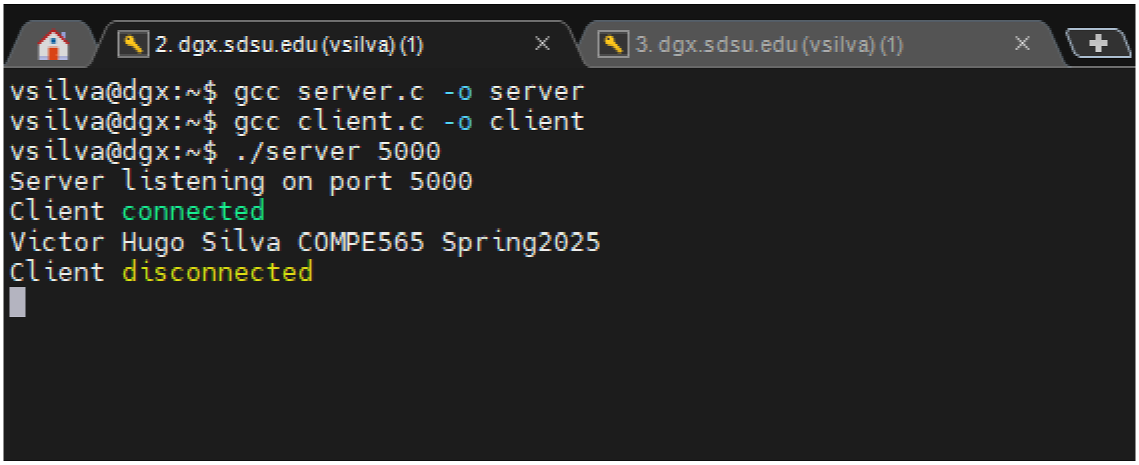
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/*
*   COMPE560 - Spring 2025
*   Victor Hugo Silva - 826603798
*   Assignment: Introduction to Socket Programming
*
*/


// Buffer size for stdin and socket I/O
#define BUF_SIZE 4096

/**
 * Prints an error message and exits.
 * @param msg  Context string to print before the system error.
 */
void die(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

/**
 * Main client entry point.
 * Usage: ./client <server-IP> <port>
 * Connects to the server, sends one line from stdin,
 * receives reversed data, prints it, and exits.
 */
int main(int argc, char **argv) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <server-IP> <port>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    const char *server_ip = argv[1];
    int port = atoi(argv[2]);

    // Create a TCP socket (IPv4)
```

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
    die("socket");
}

// Specify server address
struct sockaddr_in serv_addr = {0};
serv_addr.sin_family = AF_INET;            // IPv4
serv_addr.sin_port = htons(port);          // port in network byte order
if (inet_pton(AF_INET, server_ip, &serv_addr.sin_addr) <= 0) {
    die("inet_pton");
}

// Connect to the server
if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
    die("connect");
}
printf("Connected to %s:%d\n", server_ip, port);

// Wrap socket in FILE* streams for fread/fwrite
FILE *out = fdopen(sock, "w");      // to send data
FILE *in  = fdopen(dup(sock), "r"); // to receive data
if (!in || !out) {
    die("fdopen");
}

// Read one line from stdin
char buf[BUF_SIZE];
if (fgets(buf, BUF_SIZE, stdin) == NULL) {
    fprintf(stderr, "No input provided\n");
    exit(EXIT_FAILURE);
}
size_t len = strlen(buf);

// Send the input line to the server
if (fwrite(buf, 1, len, out) != len) {
    die("fwrite to socket");
}
fflush(out);

// Signal end-of-file to server (no more data)
shutdown(sock, SHUT_WR);

// Read reversed response and print to stdout
ssize_t n;
while ((n = fread(buf, 1, BUF_SIZE, in)) > 0) {
    if (fwrite(buf, 1, n, stdout) != n) {
        die("fwrite to stdout");
    }
    fflush(stdout);
}

// Clean up
fclose(in);
fclose(out);
```

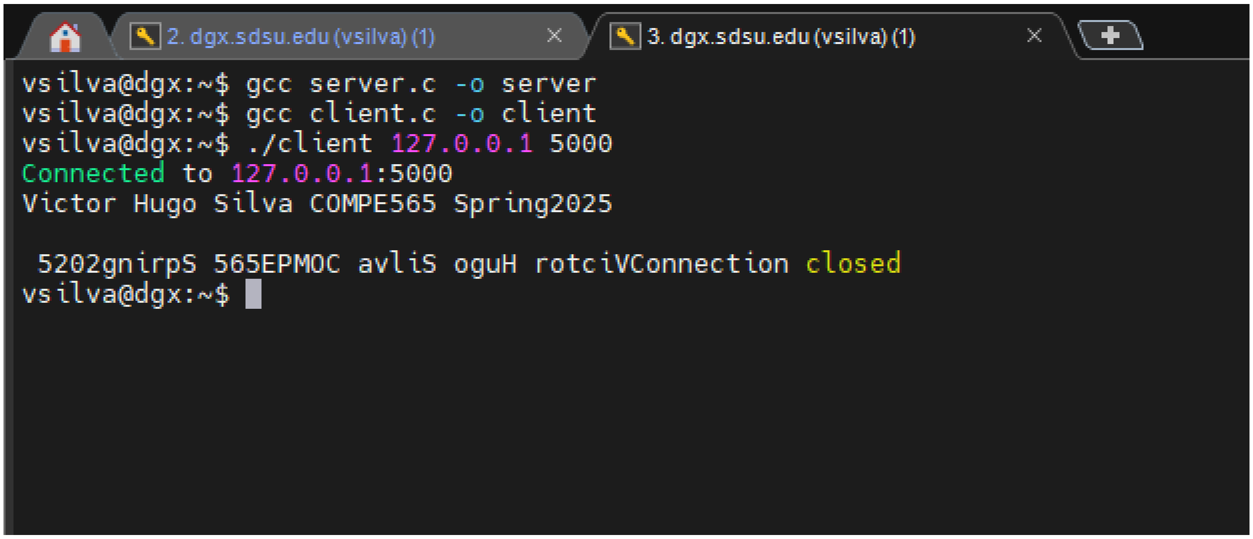
```
    printf("Connection closed\n");

    return 0;
}
```

Program Output

The results below confirm that the server logs each connection and incoming message correctly, and the client receives and prints the accurately reversed data.



```
vsilva@dgx:~$ gcc server.c -o server
vsilva@dgx:~$ gcc client.c -o client
vsilva@dgx:~$ ./server 5000
Server listening on port 5000
Client connected
Victor Hugo Silva COMPE565 Spring2025
Client disconnected
```



```
vsilva@dgx:~$ gcc server.c -o server
vsilva@dgx:~$ gcc client.c -o client
vsilva@dgx:~$ ./client 127.0.0.1 5000
Connected to 127.0.0.1:5000
Victor Hugo Silva COMPE565 Spring2025

 5202gnirpS 565EPMOC avliS oguH rotciVConnection closed
vsilva@dgx:~$
```

Questions

What are sockets, and on which layer do they operate?

Answer: Sockets are endpoints for network communication provided by the operating system. They implement transport‑layer services such as reliable byte streams for TCP or datagrams for UDP.

Differentiate between TCP and UDP? This assignment is based on TCP or UDP? YouTube uses TCP or UDP?

Answer: TCP is connection‑oriented, reliable, and delivers data in order, while UDP is connectionless, unreliable, and may drop or reorder packets. This assignment uses TCP. YouTube video delivery typically uses UDP‑based protocols for lower latency.

What will happen if I use an out-of-range port number in my code? Will I encounter error? If Yes thenwhy and if No then why?

Answer: Using a port outside the valid range (0–65535) causes bind() or connect() to fail and return –1, with errno set to EINVAL, because the OS rejects invalid port values.

What is the maximum number of sockets that a client and a server can have?

Answer: Each process can open as many sockets as its file‑descriptor limit allows (often 1024- configurable via ulimit).

Conclusion

The socket programming assignment successfully demonstrates a basic TCP client–server interaction in C. It confirms that blocking sockets, combined with fread/fwrite and proper error handling, can handle arbitrarily large data in fixed‑size chunks. The reverse‑echo protocol worked reliably on both localhost and across SDSU's Jason/Volta servers.