```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>

#define BUFFERLENGTH 256

// Main function that connects to the server, sends a command, and processes the
server's response.
int main(int argc, char **argv) {

    int client_socket, nbytes;
    struct addrinfo hints;
    struct addrinfo *result, *rp;
    int res;
    char buffer[BUFFERLENGTH];

    // Check if the correct number of arguments is passed.
    if (argc < 4 && argc > 6) {
        fprintf(stderr, "Error: Usage: %s <serverHost> <serverPort> <command> \n",
argv[0]);
        exit(1);
    }

    char *server_host = argv[1];
    char *server_port_str = argv[2];

    // Initialize hints structure for address resolution.
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = 0;
    hints.ai_protocol = 0;

    //gets the address info, returns the list of address structures.
    res = getaddrinfo(server_host, server_port_str, &hints, &result);
    if (res != 0) {
        fprintf(stderr, "Error resolving host: %s\n", gai_strerror(res));
        exit(EXIT_FAILURE);
    }

    // Loop through the address list and try to connect to each until successful.
    for (rp = result; rp != NULL; rp = rp->ai_next) {

        client_socket = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
        // Create the client socket.
        if (client_socket == -1)
            continue;

        // Attempt to connect to the server.
        if (connect(client_socket, rp->ai_addr, rp->ai_addrlen) != -1)
            break;        // Exit loop if connection is successful.
        close(client_socket);
    }

    // If no address connection succeeded, print error and exit.
    if (rp == NULL) {
```

```
        fprintf(stderr, "Could not connect\n");
        exit(EXIT_FAILURE);
    }

    freeaddrinfo(result);     // Free the address information.

    char full_command[BUFFERLENGTH];

    // Build the command string based on the number of arguments.
    if (argc == 4) {

        if (strcmp(argv[3], "R") == 0 || strcmp(argv[3], "L") == 0) {
            snprintf(full_command, sizeof(full_command), "%s", argv[3]);
        } else {
            snprintf(full_command, sizeof(full_command), "%s", argv[3]);
        }
    }
    else if (argc == 6) {
        if (strcmp(argv[3], "A") == 0 || strcmp(argv[3], "C") == 0 ||
strcmp(argv[3], "D") == 0) {
            // Format: A <ip> <port>, same for the rest lol
            // Formats commands such as "A <ip> <port>", "C <ip> <port>", and "D
<ip> <port>".
            snprintf(full_command, sizeof(full_command), "%s %s %s", argv[3],
argv[4], argv[5]);
        } else {
            snprintf(full_command, sizeof(full_command), "%s", argv[3]);
        }
    }

    else {
        fprintf(stderr, "Invalid rule\n");
        close(client_socket);
        exit(1);
    }

    // Sends the command to the server.
    if ((send(client_socket, full_command, strlen(full_command), 0)) < 0) {
        perror("Error sending command to server\n");
        close(client_socket);
        exit(1);
    }

     // Reads and prints the response from the server until there’s no more data.
    while ((nbytes = read(client_socket, buffer, BUFFERLENGTH - 1)) > 0) {
        buffer[nbytes] = '\0';
        printf("%s", buffer);
    }

    // Checks for errors during read.
    if (nbytes < 0) {
        perror("Invalid rule\n");
    }

    // Closes the client socket after communication is complete.
    close(client_socket);
    return 0;
}
```