```
//The code is a server program with an interactive mode that manages firewall
rules, client requests, and command history. It accepts commands to add, remove,
and list rules, and can operate interactively or over a network.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <unistd.h>
#include <stdbool.h>
#include <ctype.h>

#define MAX_RULES 1000
#define MAX_QUERIES 1000
#define BUFFERLENGTH 1024
#define MAX_COMMAND_HISTORY 1000


typedef struct Command {
    char command[BUFFERLENGTH];
    struct Command *next;
} Command;

//struct to store all the requests
typedef struct Requests {
    //char ip[INET_ADDRSTRLEN];
    char *ip;
    int port;
    struct Requests *next;
} Requests;

//struct that stores the firewall rules with their list of matched queries
typedef struct Rule {
    char *ip_range;
    char *port_range;
    Requests *requests;
    size_t num_requests;
    size_t capacity_requests;
    struct Rule *next;
} Rule;

//lists of the rules and all the requests and the command history
Rule *rules = NULL;
Requests *all_requests = NULL;
Command *command_history = NULL;

int command_count = 0;

pthread_mutex_t request_mutex = PTHREAD_MUTEX_INITIALIZER;

// Validates if the input string is a valid IP address.
int check_valid_ip(char *ip) {
    struct sockaddr_in sa;
    return inet_pton(AF_INET, ip, &(sa.sin_addr)) != 0;
}

// Converts an IP address from string format to an integer for easier comparison.
unsigned int ip_to_int(char *ip) {
    unsigned int result;
    struct sockaddr_in sa;
    inet_pton(AF_INET, ip, &(sa.sin_addr));
    result = ntohl(sa.sin_addr.s_addr);
    return result;
}

// Checks if an IP range format (ip1-ip2) is valid, ensuring ip1 < ip2 and both are
valid IPs.
int check_valid_ip_range(char *ip_range) {
    char ip1[INET_ADDRSTRLEN], ip2[INET_ADDRSTRLEN];
    if (sscanf(ip_range, "%[^-]-%s", ip1, ip2) == 2) {
        if (!check_valid_ip(ip1) || !check_valid_ip(ip2)) {
            return 0;
        }
        if (ip_to_int(ip1) >= ip_to_int(ip2)) {
            return 0;
        }
        return 1;
    }
    return 0;
}

// Determines if an IP address falls within the specified IP range [start_ip,
end_ip].
int check_ip_in_range(const char *ip, const char *start_ip, const char *end_ip) {
    struct in_addr ip_addr, start_addr, end_addr;
    inet_pton(AF_INET, ip, &ip_addr);
    inet_pton(AF_INET, start_ip, &start_addr);
    inet_pton(AF_INET, end_ip, &end_addr);

    return ntohl(ip_addr.s_addr) >= ntohl(start_addr.s_addr) &&
           ntohl(ip_addr.s_addr) <= ntohl(end_addr.s_addr);
}

// Validates if a port number is within the allowed range (0-65535).
int check_valid_port(int port) {
    return port >= 0 && port <= 65535;
}

// Checks if a port range format (port1-port2) is valid, ensuring port1 < port2.
int check_valid_port_range(char *port_range) {
    int port1, port2;
    if (sscanf(port_range, "%d-%d", &port1, &port2) == 2) {
        if (port1 < 0 || port1 > 65535 || port2 < 0 || port2 > 65535) {
            return 0;
        }
        if (port1 >= port2) {
            return 0;
        }
        return 1;
    }
    return 0;
}

// Adds a client’s request (IP and port) to the global list of requests.
void add_request_to_list_of_requests(char *ip, int port) {
    Requests *new_query = (Requests *)malloc(sizeof(Requests));
    if (new_query == NULL) {
        perror("Failed to allocate memory for new request");
        return;
    }

    new_query->ip = strdup(ip);
    if (new_query->ip == NULL) {
        free(new_query);
        perror("Failed to allocate memory for IP string");
        return;
    }

    new_query->port = port;
    new_query->next = NULL;

    pthread_mutex_lock(&request_mutex);
    new_query->next = all_requests;
    all_requests = new_query;
    pthread_mutex_unlock(&request_mutex);
}

//add a request to a rule
void add_req_to_rule(Rule *rule, char *ip, int port) {
    if (rule == NULL || ip == NULL) return;

    Requests *new_query = (Requests *)malloc(sizeof(Requests));
    if (new_query == NULL) {
        perror("Failed to allocate memory for request");
        free(new_query);
        return;
    }
    new_query->ip = NULL;

    new_query->ip = strdup(ip);
    if (new_query->ip == NULL) {
        free(new_query);
        perror("Failed to allocate memory for IP string");
        return;
    }

    new_query->port = port;
    new_query->next = rule->requests;
    rule->requests = new_query;
}

// Creates and adds a new firewall rule with IP and port ranges to the rule list.
void add_rule(char *ip_range, char *port_range) {
    Rule *new_rule = (Rule *)malloc(sizeof(Rule));
    if (new_rule == NULL) {
        //pls bro handle malloc failure gracefully i beg u. YESSS
        printf("Memory allocation failed\n");
        return;
    }

    //allocate memory for ip_range and port_range dynamically
    new_rule->ip_range = (char *)malloc(strlen(ip_range) + 1);
    new_rule->port_range = (char *)malloc(strlen(port_range) + 1);

    if (new_rule->ip_range == NULL || new_rule->port_range == NULL) {

        printf("Memory allocation failed for ip_range or port_range\n");
        free(new_rule);
        return;
    }

    //copy the strings
    strcpy(new_rule->ip_range, ip_range);
    strcpy(new_rule->port_range, port_range);

    //initialize the rest of the fields
    new_rule->requests = NULL;
    new_rule->next = rules; // valid rule is added to the list of rules
    rules = new_rule;
}

// Logs a command into the command history list.
void add_comm_to_the_history(char *command) {
    Command *new_command = (Command *)malloc(sizeof(Command));
    strcpy(new_command->command, command);
    new_command->next = command_history;
    command_history = new_command;
    command_count++;
}

// Displays command history in reverse order for interactive mode.
void handle_R_comm_i() {
    pthread_mutex_lock(&request_mutex);

    int command_count = 0;
    Command *current = command_history;
    while (current != NULL) {
        command_count++;
        current = current->next;
    }

    Command **commands = (Command **)malloc(command_count * sizeof(Command *));
    if (commands == NULL) {
        perror("Memory allocation failed");
        pthread_mutex_unlock(&request_mutex);
        return;
    }

    current = command_history;
    for (int i = 0; i < command_count; i++) {
        commands[i] = current;
        current = current->next;
    }

    for (int i = command_count - 1; i >= 0; i--) {
        printf("%s\n", commands[i]->command);
    }

    free(commands);
    pthread_mutex_unlock(&request_mutex);
}

// Sends the command history to a connected client in reverse order over the
network.
void handle_R_comm(int new_socket) {
    pthread_mutex_lock(&request_mutex);
```

```c
    Command *current = command_history;
    int count = 0;
    Command *commands[command_count]; //store commands in array
    while (current != NULL && count < command_count) {
        commands[count++] = current;
        current = current->next;
    }

    char buffer[1024];
    memset(buffer, 0, sizeof(buffer));  //clear the buffer before use
    for (int i = count - 1; i >= 0; i--) { //add to buffer in reverse
        int len = snprintf(buffer + strlen(buffer), sizeof(buffer) -
strlen(buffer), "%s\n", commands[i]->command);
        if (len < 0 || strlen(buffer) >= sizeof(buffer) - 1) {
            break; //break if buffer full
        }
    }
    write(new_socket, buffer, strlen(buffer));
    pthread_mutex_unlock(&request_mutex);
}

// Checks if a given string contains only numeric characters.
int is_int(const char *str) {
    while (*str) {
        if (!isdigit(*str)) return 0;
        str++;
    }
    return 1;
}

// Processes and executes incoming commands based on their type, either in
interactive or network mode.
void process_command(char *command, int new_socket, bool is_interactive) {
    char cmd;
    char ip_range[50], port_range[50];
    char buffer[BUFFERLENGTH];

    add_comm_to_the_history(command);

    sscanf(command, "%c", &cmd);
    switch (cmd) {
        char extra[BUFFERLENGTH];
        case 'R': {
            if (is_interactive) handle_R_comm_i();
            else handle_R_comm(new_socket);
            break;
        }
        case 'A': {
            if (sscanf(command + 2, "%s %s", ip_range, port_range) == 2) {
                if (strchr(ip_range, '-') != NULL) {
                    //ip range
                    if (!check_valid_ip_range(ip_range)) {
                        snprintf(buffer, BUFFERLENGTH, "Invalid rule\n");
                        if (is_interactive) printf("%s", buffer);
                        else write(new_socket, buffer, strlen(buffer));
                        break;
                    }
                } else {
                    //for single IP
                    if (!check_valid_ip(ip_range)) {
                        snprintf(buffer, BUFFERLENGTH, "Invalid rule\n");
                        if (is_interactive) printf("%s", buffer);
                        else write(new_socket, buffer, strlen(buffer));
                        break;  //reject invalid IP address
                    }
                } if (strchr(port_range, '-') != NULL) {
                    if (!check_valid_port_range(port_range)) {
                        snprintf(buffer, BUFFERLENGTH, "Invalid rule\n");
                        if (is_interactive) printf("%s", buffer);
                        else write(new_socket, buffer, strlen(buffer));
                        break;
                    }
                } else {
                    int single_port;
                    if (sscanf(port_range, "%d", &single_port) != 1 || single_port
< 0 || single_port > 65535) {
                        snprintf(buffer, BUFFERLENGTH, "Invalid rule\n");
                        if (is_interactive) printf("%s", buffer);
                        else write(new_socket, buffer, strlen(buffer));
                        break;
                    }
                }

                add_rule(ip_range, port_range); //add rule if IP and is valid
                snprintf(buffer, BUFFERLENGTH, "Rule added\n");

                if (is_interactive) printf("%s", buffer);
                else write(new_socket, buffer, strlen(buffer));

            } else {
                snprintf(buffer, BUFFERLENGTH, "Invalid rule\n");

                if (is_interactive) printf("%s", buffer);
                else write(new_socket, buffer, strlen(buffer));
            }
            break;
        }

        case 'C': {
            char ip[INET_ADDRSTRLEN];
            char start_ip[INET_ADDRSTRLEN], end_ip[INET_ADDRSTRLEN];
            int port, start_port, end_port;

            if (sscanf(command + 2, "%s %d %s", ip, &port, extra) == 2) {
                // Validate the IP and port
                if (!check_valid_ip(ip) || !check_valid_port(port)) {
                    if (is_interactive) printf("Illegal IP address or port
specified\n");
                    else write(new_socket, "Illegal IP address or port specified\
n", strlen("Illegal IP address or port specified\n"));
                    break;
                }
                Rule *rule = rules;
                int found = 0;

                while (rule) {
                    // Parse the rule's IP and port ranges
                    if (sscanf(rule->ip_range, "%15[^-]-%15s", start_ip, end_ip) ==
2 &&
                        sscanf(rule->port_range, "%d-%d", &start_port, &end_port)
== 2) {
                            if (check_ip_in_range(ip, start_ip, end_ip) && port >=
start_port && port <= end_port) {
                            found = 1;
                            break;
                        }
                    }

                    else if (strcmp(rule->ip_range, ip) == 0 && atoi(rule-
>port_range) == port) {
                        found = 1;
                        break;
                    }

                    rule = rule -> next;
                }

                if (found) {
                    snprintf(buffer, BUFFERLENGTH, "Connection accepted\n");

                    if (is_interactive) printf("%s", buffer);
                    else write(new_socket, buffer, strlen(buffer));

                    add_req_to_rule(rule, ip, port);
                    add_request_to_list_of_requests(ip, port); //add to all
requests list

                } else {
                    snprintf(buffer, BUFFERLENGTH, "Connection rejected\n");

                    if (is_interactive) printf("%s", buffer);
                    else write(new_socket, buffer, strlen(buffer));
                }

            } else {
                if (is_interactive) printf("Illegal IP address or port specified\
n");
                else write(new_socket, "Illegal IP address or port specified\n",
strlen("Illegal IP address or port specified\n"));
            }

            break;
        }

        case 'D': {
            if (sscanf(command + 2, "%s %s %s", ip_range, port_range, extra) == 2)
{
                if (!is_int(port_range)) {
                    if (is_interactive) printf("Invalid rule\n");
                    else write(new_socket, "Invalid rule\n", strlen("Invalid rule\
n"));
                    break;
                }

                Rule *prev = NULL, *curr = rules;
                while (curr) {
                    if (strcmp(curr->ip_range, ip_range) == 0 && strcmp(curr-
>port_range, port_range) == 0) {
                        if (prev) {
                            prev->next = curr->next;
                        } else {
                            rules = curr->next;
                        }
                        free(curr->ip_range);
                        free(curr->port_range);
                        free(curr);
                        if (is_interactive) printf("Rule deleted\n");
                        else write (new_socket, "Rule deleted\n", strlen("Rule
deleted\n"));
                        return;
                    }
                    prev = curr;
                    curr = curr->next;
                }

                if (is_interactive) printf("Rule not found\n");
                else write(new_socket, "Rule not found\n", strlen("Rule not found\
n"));

            } else {
                if (is_interactive) printf("Rule invalid\n");
                else write(new_socket, "Rule invalid\n", strlen("Rule invalid\n"));
            }

            break;
        }
        case 'L': {

            Rule *rule = rules;
            while (rule) {
                snprintf(buffer, BUFFERLENGTH, "Rule: %s %s\n", rule->ip_range,
rule->port_range);

                if (is_interactive) printf("%s", buffer);
                else write(new_socket, buffer, strlen(buffer));

                Requests *query = rule->requests;
                while (query) {
                    snprintf(buffer, BUFFERLENGTH, "Query: %s %d\n", query->ip,
query->port);

                    if (is_interactive) printf("%s", buffer);
                    else write(new_socket, buffer, strlen(buffer));

                    query = query->next;
                }

                rule = rule->next;
            }

            break;
        }

        default:
            if (is_interactive) printf("Illegal request\n");
            else write(new_socket, "Illegal request\n", strlen("Illegal request\
n"));
            break;
    }
}

// Frees memory allocated for a list of requests to prevent memory leaks.
void free_reqs(Requests *request) {
    Requests *current = request;
    Requests *next;

    while (current != NULL) {
        next = current->next;

        if (current->ip != NULL) {
            free(current->ip);
        }

        free(current);
        current = next;
    }
}

// Frees memory allocated for a rule and its associated requests.
void free_rules(Rule *rule) {
    if (rule == NULL) return;
    free_reqs(rule->requests);
    free(rule);
    exit(0);
}

// Frees memory used by the command history list.
void free_comm_history() {
    Command *current_command = command_history;
    while (current_command) {
        Command *next_command = current_command->next;
        free(current_command);
        current_command = next_command;
    }
}

// Frees memory used by the global list of all requests.
void free_all_reqs() {
    Requests *current_request = all_requests;
    while (current_request) {
        Requests *next_request = current_request->next;
        free_reqs(current_request);
        current_request = next_request;
    }
}

// Runs the server in interactive mode, continuously reading and processing
commands from standard input.
void interactive_mode() {
    char command[BUFFERLENGTH];
    while (1) {
        if (fgets(command, BUFFERLENGTH, stdin) == NULL) {
            break;
        }
        command[strcspn(command, "\n")] = 0;  //to remove trailing newline
        process_command(command, -1, true);
    }
}

// Initializes the server, either in interactive or network mode, setting up a
socket and handling incoming connections.
int main(int argc, char **argv) {

    setvbuf(stdout, NULL, _IONBF, 0);

    if (argc > 1 && strcmp(argv[1], "-i") == 0) {
        interactive_mode();

        return 0;
    }

    socklen_t client_len;
    int server_socket, new_socket, portno;
    char buffer[BUFFERLENGTH];
    struct sockaddr_in server_addr, client_addr;
    int n;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <port>\n", argv[0]);
        exit(1);
    }

    //create socket
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket < 0) {
        perror("Error opening socket");
    }

    bzero ((char *) &server_addr, sizeof(server_addr));
    portno = atoi(argv[1]);
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons (portno);

    //bind socket to port
    if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) <
0) {
        perror("Binding failed");
    }

    //listen for incoming connections
    listen(server_socket, 3);
    client_len = sizeof(client_addr);

    while (1) {
        new_socket = accept(server_socket, (struct sockaddr *)&client_addr,
&client_len);
        if (new_socket < 0) {
            perror("Accept failed");
        }
        bzero(buffer,BUFFERLENGTH);
        n = read (new_socket, buffer, BUFFERLENGTH - 1);
        if (n < 0) {
            perror ("Error reading from socket");
        } else if (n == 0) {
            printf("Client disconnected\n");
            close(new_socket);
            continue;
        }
        switch (buffer[0])
        {
        case 'R':
        {
            process_command(buffer, new_socket, false);
            break;
        }
        case 'A': {
            process_command(buffer, new_socket, false);
            break;
        }
        case 'C': {
            process_command(buffer, new_socket, false);
            break;
        }
        case 'D': {
            process_command(buffer, new_socket, false);
            break;
        }
        case 'L': {
            process_command(buffer, new_socket, false);
            break;
        }
        default:
            write(new_socket, "Illegal request\n", strlen("Illegal request\n"));
        }

        close(new_socket);
    }

    free_rules(rules);
    free_reqs(all_requests);
    free_comm_history(command_history);
    free_all_reqs();

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