VSCODE_PRINT_SCRIPT_TAGS

Selected files

2 printable files

```
Assignments\Assignment3\stnc.c
Assignments\Assignment3\makefile
```

Assignments\Assignment3\stnc.c

```c
#include<stdio.h>
#include<stdlib.h>
#include <stdint.h>
#include<string.h>
#include <sys/un.h>
#include<sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include<netinet/tcp.h>
#include<netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <poll.h>
#include <netdb.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
#include <unistd.h>

#define B_SIZE 6400
#define B_SIZE_UDP 64000
# define B_SIZE_UDP_IPV6 3000
#define FIFO_NAME "OS_EX3_pipe"
#define SOCK_PATH "/tmp/stnc_sock_path.sock"

int ipv6_to_ipv4(char *ipv6_str, char *ipv4_str) {
    struct in6_addr ipv6_addr;
    struct sockaddr_in ipv4_addr;
    int ret;

    // Convert the IPv6 address string to a binary representation
    ret = inet_pton(AF_INET6, ipv6_str, &ipv6_addr);
    if (ret != 1) {
        return -1;
    }

    // Convert the IPv6 address to an IPv4-mapped IPv6 address
    memset(&ipv4_addr, 0, sizeof(ipv4_addr));
    ipv4_addr.sin_family = AF_INET;
    ipv4_addr.sin_port = 0;
    ipv4_addr.sin_addr.s_addr =
            htonl(0xFFFF0000) | ((ipv6_addr.s6_addr[12] << 8) & 0xFF00) |
(ipv6_addr.s6_addr[13] & 0xFF);

```

```c
    // Convert the IPv4 address to a string
    inet_ntop(AF_INET, &ipv4_addr.sin_addr, ipv4_str, INET_ADDRSTRLEN);

    return 0;
}

void port_for_info(char *port, char *port_out) {
    int new_port = atoi(port);
    new_port++;
    if (new_port == 65536) {
        new_port -= 2;
    }
    sprintf(port_out, "%d", new_port);
}


long checksum(char *str, int limit) {
    long sum = 0;
    for (int i = 0; str[i] != '\0' && i < limit; i++) {
        sum += (uint8_t) str[i];
    }
    return sum;
}

long checksum_file(FILE *file, long *bytes_counter) {
    uint8_t byte;
    long sum = 0;
    long count = 0;
    while (fread(&byte, sizeof(byte), 1, file) == 1) {
        sum += byte;
        count++;
    }
    *bytes_counter = count;
    return sum;
}


int portHandler(int port) {
    if ((port <= 1024) || (port >= 65535)) {
        printf("PORT should be numerical between 1024 and 65535 included\n");
        exit(EXIT_FAILURE);
    }
    return 0;
}

int IPtype(char *ip) {
    for (int i = 0; ip[i] != '\0'; i++) {
        if (ip[i] == ':') {
            return 6;
        }
    }
    return 4;
}


int IPv4Handler(char *ip) {
    char *token = strtok(ip, ".");
    int dot_count = 0;
    while (token != NULL) {
        int num = atoi(token);
        if (num < 0 || num > 255) {
            printf("Invalid IP address\n");
            return 0;
        }
        token = strtok(NULL, ".");
        dot_count++;
    }

    if (dot_count != 4) {
        printf("Invalid IP address\n");
        return 0;
    }

    return 1;
}

int IPv6Handler(char *ip) {
    struct sockaddr_in6 sa6;
    int result = inet_pton(AF_INET6, ip, &(sa6.sin6_addr));
    return result == 1;
}


int tcp_client_conn(char *ip, char *port) {
    int PORT = atoi(port);
    char IP[100];
    strcpy(IP, ip);
    portHandler(PORT);

    int ip_type = IPtype(ip);

    if (ip_type == 4) {

        if (IPv4Handler(IP)) {
            //initializing a TCP socket.
            int sock = socket(AF_INET, SOCK_STREAM, 0);
            if (sock == -1) {
                printf("Could not create socket.\n");
                exit(EXIT_FAILURE);
            }

            struct sockaddr_in receiver_adderess;
            //setting to zero the struct senderAddress
            memset(&receiver_adderess, 0, sizeof(receiver_adderess));
            receiver_adderess.sin_family = AF_INET;
            receiver_adderess.sin_port = htons(PORT);
            int checkP = inet_pton(AF_INET, (const char *) IP,
&receiver_adderess.sin_addr);

            if (checkP < 0) {
                printf("inet_pton() FAILED.\n");
                exit(EXIT_FAILURE);
            }

            //connecting to the Receiver on the socket
            sleep(1);
            int connectCheck = connect(sock, (struct sockaddr *) &receiver_adderess,
sizeof(receiver_adderess));

            if (connectCheck == -1) {
```

```c
                printf("connect() FAILED.\n");
                exit(EXIT_FAILURE);
            }
            return sock;
        }
    } else if (ip_type == 6) {
        if (IPv6Handler(ip)) {//need to create IPv6Handler method
            int sock = socket(AF_INET6, SOCK_STREAM, 0);
            if (sock == -1) {
                perror("socket error\n");
                exit(EXIT_FAILURE);
            }
            int on = 1;
            if (setsockopt(sock, IPPROTO_IPV6, IPV6_V6ONLY, &on, sizeof(on)) == -1) {
                perror("setsockopt error\n");
                exit(EXIT_FAILURE);
            }
            // struct addrinfo hints, *res; //tell alon there is an error in hints
            // memset(&hints, 0, sizeof(hints)); //error on sizeof
            // hints.ai_family = AF_INET6; //error on hints
            // hints.ai_socktype = SOCK_STREAM;//error on hints
            // if (getaddrinfo(ip, port, &hints, &res) != 0) {
            //     perror("getaddrinfo error");
            //     exit(EXIT_FAILURE);
            // }
            // if (connect(sock, res->ai_addr, res->ai_addrlen) == -1) {
            //     perror("connect error");
            //     exit(EXIT_FAILURE);
            // }
            return sock;
        }
    }
    return -1;
}

int tcp_server_conn(char *port) {
    int PORT = atoi(port);
    char IP[] = "0.0.0.0";
    portHandler(PORT);

    //initializing a TCP socket.
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in senderAddress;
    //setting to zero the struct senderAddress
    memset(&senderAddress, 0, sizeof(senderAddress));
    senderAddress.sin_family = AF_INET;
    senderAddress.sin_port = htons(PORT);

    //opening the socket.
    int Bcheck = bind(sock, (struct sockaddr *) &senderAddress, sizeof(senderAddress));
    if (Bcheck == -1) {
        return 1;
    }
    //start listening on the socket (one client at the time)
    int Lcheck = listen(sock, 4);
    if (Lcheck == -1) {
        printf("Error in listen().\n");
        return 1;
    }

    //accepting the client (the Sender)
    unsigned int senderAddressLen = sizeof(senderAddress);
    int senderSock = accept(sock, (struct sockaddr *) &senderAddress, &senderAddressLen);
    if (senderSock == -1) {
        printf("accept() failed.\n");
        close(sock);
        return 1;
    }
    close(sock);

    return senderSock;

}


void usage() {
    printf("Usage options:\n");
    printf("client side usage: ./stnc -c IP PORT\n");
    printf("server side usage: ./stnc -s PORT\n");
}


int client_A(char *port, char *ip) {

    int sock = tcp_client_conn(ip, port);
    int fd = -1;
    struct pollfd fds[2];
    fds[0].fd = 0; // stdin
    fds[0].events = POLLIN; // tell me when I can read from it
    fds[1].fd = sock;
    fds[1].events = POLLIN;
    while (1) {
        int err = poll(fds, 2, -1);
        if (err < 0) {
            printf("poll failed\n");
            return 1;
        }
        if (fds[0].revents && POLLIN) {
            // read from keyboard and send to the server
            char buffer[10000] = {'\0'};
            if (fgets(buffer, sizeof(buffer), stdin) != NULL) {
                // remove newline character from the end of the line
                buffer[strcspn(buffer, "\n")] = '\0';
            }
            int size = strlen(buffer);
            if (send(sock, buffer, size, 0) < 0) {
                perror("could not send the data\n");
                return 1;
            }
        }
        if (fds[1].revents && POLLIN) {
            // read from sock and print out
            char buffer2[10000] = {'\0'};
            if (recv(sock, buffer2, 10000, 0) < 0) {
                perror("could not receive data\n");
                return 1;
            }
            printf("%s\n", buffer2);
        }
    }
    return 0;
}

int server_A(char *port) {
    int PORT = atoi(port);
    char IP[] = "0.0.0.0";
    portHandler(PORT);

    int sock = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in senderAddress;
    //setting to zero the struct senderAddress
    memset(&senderAddress, 0, sizeof(senderAddress));
    senderAddress.sin_family = AF_INET;
    senderAddress.sin_port = htons(PORT);

    //opening the socket.
    int Bcheck = bind(sock, (struct sockaddr *) &senderAddress, sizeof(senderAddress));
    if (Bcheck == -1) {
        return 1;
    }
    //start listening on the socket (one client at the time)
    int Lcheck = listen(sock, 1);
    if (Lcheck == -1) {
        printf("Error in listen().\n");
        return 1;
    }

    //accepting the client (the Sender)
    unsigned int senderAddressLen = sizeof(senderAddress);
    int senderSock = accept(sock, (struct sockaddr *) &senderAddress, &senderAddressLen);
    if (senderSock == -1) {
        printf("accept() failed.\n");
        close(sock);
        return 1;
    }
    int fd = -1;
    struct pollfd fds[2];
    fds[0].fd = 0; // stdin
    fds[0].events = POLLIN; // tell me when I can read from it
    fds[1].fd = senderSock;
    fds[1].events = POLLIN;
    while (1) {
        int err = poll(fds, 2, -1);
        if (err < 0) {
            perror("poll failed\n");
            return 1;
        }
        if (fds[0].revents & POLLIN) {
            // read from keyboard and send to the server
            char buffer[10000] = {'\0'};

            if (fgets(buffer, sizeof(buffer), stdin) != NULL) {
                // remove newline character from the end of the line
                buffer[strcspn(buffer, "\n")] = '\0';
            }
            int size = strlen(buffer);
            if (send(senderSock, buffer, size, 0) < 0) {
                perror("could not send the data\n");
                close(senderSock);
                return 1;
```

```c
            }
        }
        if (fds[1].revents & POLLIN) {
            // read from sock and print out
            char buffer2[10000] = {'\0'};
            if (recv(senderSock, buffer2, 10000, 0) < 0) {
                perror("could not receive data\n");
                close(senderSock);
                return 1;
            }
            printf("%s\n", buffer2);
        }
    }

    return 0;
}

//sending on the info socket to the server <type> , <param>, checksum, bytes to send.
void type_param(int sock, char *type, char *param, long checks, long bytes) {
    char checks_str[16] = {'\0'};
    char bytes_str[16] = {'\0'};
    sprintf(checks_str, "%ld", checks);
    sprintf(bytes_str, "%ld", bytes);
    char message[100] = {'\0'};
    strcat(message, type);
    strcat(message, ",");
    strcat(message, param);
    strcat(message, ",");
    strcat(message, checks_str);
    strcat(message, ",");
    strcat(message, bytes_str);

    send(sock, message, strlen(message), 0);
}

int client_TCP_B(char *ip, char *port, int info_sock, FILE *file) {
    int data_sock = tcp_client_conn(ip, port);

    char buffer[B_SIZE];
    size_t bytes_read;
    char *start = "start";
    char *end = "end";
    send(info_sock, start, strlen(start), 0);
    fseek(file, 0L, SEEK_SET);
    while (1) {
        bytes_read = fread(buffer, 1, B_SIZE, file);

        if (bytes_read == 0) {
            // End of file
            break;
        }
        //buffer[bytes_read] = '\0'; // add null terminator
        if (send(data_sock, buffer, bytes_read, 0) == -1) {
            perror("send");
            exit(EXIT_FAILURE);
        }
    }

    send(info_sock, end, strlen(end), 0);

    close(data_sock);
    close(info_sock);
    return 0;
}

int server_TCP_B(char *port, int info_sock, long bytes_target, long checksum_target, int
q) {
    struct timeval start, end, diff;
    int data_sock = tcp_server_conn(port);

    struct pollfd fds[2];
    fds[0].fd = info_sock;
    fds[0].events = POLLIN; // tell me when I can read from it
    fds[1].fd = data_sock;
    fds[1].events = POLLIN;
    long bytes_recived = 0;
    long checksum_sum = 0;
    int done = 0;
    int started = 0;
    char buffer[B_SIZE];
    char buffer_str[B_SIZE + 1];
    while (1) {
        memset(buffer, 0, sizeof(buffer));
        int err = poll(fds, 2, -1);
        if (err < 0) {
            perror("poll");
            return 1;
        }
        if (fds[0].revents && POLLIN) {
            //read from the socket
            if (!started)
                recv(info_sock, buffer, strlen("start") + 1, 0);
            else
                recv(info_sock, buffer, strlen("end") + 1, 0);
            if (strcmp(buffer, "start") == 0) {
                gettimeofday(&start, NULL);
                started = 1;
            } else if (strcmp(buffer, "end") == 0) {
                gettimeofday(&end, NULL);
                done = 1;
            }
        }
        else if (fds[1].revents && POLLIN) {
            // recive the data and count byts
            bytes_recived += recv(data_sock, buffer, sizeof(buffer), 0);
            strcpy(buffer_str, buffer);
            buffer_str[B_SIZE] = '\0';
            checksum_sum += checksum(buffer_str, B_SIZE + 1);
        }

        if (done == 1) {
            //set the socket to non-blocking code
            int flags = fcntl(data_sock, F_GETFL, 0);
            fcntl(data_sock, F_SETFL, flags | O_NONBLOCK);

            // receive data for one second
            time_t start_time = time(NULL);
            while (time(NULL) - start_time <= 1) {
                long bytes_temp = 0;
                bytes_temp = recv(data_sock, buffer, sizeof(buffer), 0);
```

```c
                if (bytes_temp != 0) {
                    bytes_recived += bytes_temp;
                    strcpy(buffer_str, buffer);
                    buffer_str[B_SIZE] = '\0';
                    checksum_sum += checksum(buffer_str, B_SIZE + 1);
                    gettimeofday(&end, NULL);
                    start_time = time(NULL);
                }
            }
            break;
        }
    }
    close(data_sock);
    timersub(&end, &start, &diff);
    if (!q) {
        printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
    }
// compare checksum and bytes
    if (bytes_recived != bytes_target) {
        if (!q)
            printf("error: did not received full data!\n");
        else
            printf("failure\n");
        return 1;
    }
    if (checksum_target != checksum_sum) {
        if (!q) {
            printf("error: checksum failed\n");
            printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
        } else {
            printf("failure\n");
        }

        return 1;
    }

//print results
    long microsec = diff.tv_usec;
    long milisec = microsec / 1000;
    milisec += diff.tv_sec * 1000;
    printf("ipv4_tcp,%ld\n", milisec);
    return 0;
}


int client_TCP_IPV6_B(char *ip, char *port, int info_sock, FILE *file){
    int data_sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (data_sock < 0) {
        perror("Error creating socket\n");
        return 1;
    }
    int int_port = atoi(port);
    struct sockaddr_in6 addr;
    memset(&addr, 0, sizeof(addr));
    addr.sin6_family = AF_INET6;
    addr.sin6_port = htons(int_port);


//    if (inet_pton(AF_INET6, ip, &addr.sin6_addr) <= 0) {
//        perror("inet_pton error");
//        return 1;
//    }


    if (connect(data_sock, (struct sockaddr *)&addr, sizeof(addr)) == -1) {
        perror("connect error");
        return 1;
    }



    char buffer[B_SIZE];
    size_t bytes_read;
    char *start = "start";
    char *end = "end";
    send(info_sock, start, strlen(start), 0);
    fseek(file, 0L, SEEK_SET);
    while (1) {
        bytes_read = fread(buffer, 1, B_SIZE, file);

        if (bytes_read == 0) {
            // End of file
            break;
        }
        if(bytes_read < B_SIZE){
            printf("%zu\n",bytes_read);
        }
        //buffer[bytes_read] = '\0'; // add null terminator
        if (send(data_sock, buffer, bytes_read, 0) == -1) {
            perror("send");
            exit(EXIT_FAILURE);
        }
    }

    send(info_sock, end, strlen(end), 0);

    close(data_sock);
    close(info_sock);
    return 0;
}

int server_TCP_IPV6_B(char *port, int info_sock, long bytes_target, long checksum_target, int q){
    struct timeval start, end, diff;
    int sockfd = socket(AF_INET6, SOCK_STREAM, 0);
    if (sockfd == -1) {
        perror("socket");
        return 1;
    }
    int int_port = atoi(port);
    struct sockaddr_in6 servaddr, cliaddr;
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin6_family = AF_INET6;
    servaddr.sin6_addr = in6addr_any;
    servaddr.sin6_port = htons(int_port);

    if (bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == -1) {
        perror("bind error");
        return 1;
    }

    if (listen(sockfd, 2) == -1) {
        perror("listen error");
        return 1;
    }

    socklen_t clilen = sizeof(cliaddr);

    // Accept the next incoming connection and create a new connected socket
    int data_sock = accept(sockfd, (struct sockaddr *)&cliaddr, &clilen);
    if (data_sock == -1) {
        perror("accept error");
        return 1;
    }




    struct pollfd fds[2];
    fds[0].fd = info_sock;
    fds[0].events = POLLIN; // tell me when I can read from it
    fds[1].fd = data_sock;
    fds[1].events = POLLIN;
    long bytes_recived = 0;
    long checksum_sum = 0;
    int done = 0;
    int started = 0;
    char buffer[B_SIZE];
    char buffer_str[B_SIZE + 1];
    while (1) {
        memset(buffer, 0, sizeof(buffer));
        int err = poll(fds, 2, -1);
        if (err < 0) {
            perror("poll");
            return 1;
        }
        if (fds[0].revents && POLLIN) {
            //read from the socket
            if (!started)
                recv(info_sock, buffer, strlen("start") + 1, 0);
            else
                recv(info_sock, buffer, strlen("end") + 1, 0);
            if (strcmp(buffer, "start") == 0) {
                gettimeofday(&start, NULL);
                started = 1;
            } else if (strcmp(buffer, "end") == 0) {
                gettimeofday(&end, NULL);
                done = 1;
            }
        }
        else if (fds[1].revents && POLLIN) {
            // recive the data and count byts
            bytes_recived += recv(data_sock, buffer, sizeof(buffer), 0);
            strcpy(buffer_str, buffer);
            buffer_str[B_SIZE] = '\0';
            checksum_sum += checksum(buffer_str, B_SIZE + 1);
        }

        if (done == 1) {
            //set the socket to non-blocking code
            int flags = fcntl(data_sock, F_GETFL, 0);
            fcntl(data_sock, F_SETFL, flags | O_NONBLOCK);

            // receive data for one second
            time_t start_time = time(NULL);
            while (time(NULL) - start_time <= 1) {
                long bytes_temp = 0;
                bytes_temp = recv(data_sock, buffer, sizeof(buffer), 0);
                if (bytes_temp != 0) {
                    bytes_recived += bytes_temp;
                    strcpy(buffer_str, buffer);
                    buffer_str[B_SIZE] = '\0';
                    checksum_sum += checksum(buffer_str, B_SIZE + 1);
                    gettimeofday(&end, NULL);
                    start_time = time(NULL);
                }
            }
            break;
        }
    }
    close(data_sock);
    close(info_sock);
    close(sockfd);

    timersub(&end, &start, &diff);
    if (!q) {
        printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
    }
// compare checksum and bytes
    if (bytes_recived != bytes_target) {
        if (!q)
            printf("error: did not received full data!\n");
        else
            printf("failure\n");
        return 1;
    }
    if (checksum_target != checksum_sum) {
        if (!q) {
            printf("error: checksum failed\n");
            printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
        } else {
            printf("failure\n");
        }

        return 1;
    }

//print results
    long microsec = diff.tv_usec;
    long milisec = microsec / 1000;
    milisec += diff.tv_sec * 1000;
    printf("ipv6_tcp,%ld\n", milisec);
    return 0;
}

int client_UDP_B(char *ip, char *port, int info_sock, FILE *file) {

    // open udp sock
    struct sockaddr_in server_addr;
    int data_sock = socket(AF_INET, SOCK_DGRAM, 0);
```

```c
    if (data_sock < 0) {
        perror("Error creating socket\n");
        return 1;
    }
    int int_port = atoi(port);
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(int_port);
    inet_pton(AF_INET, ip, &server_addr.sin_addr);

    char buffer[B_SIZE_UDP];
    size_t bytes_read;
    char *start = "start";
    char *end = "end";
    send(info_sock, start, strlen(start), 0);
    fseek(file, 0L, SEEK_SET);
    while (1) {
        bytes_read = fread(buffer, 1, B_SIZE_UDP, file);

        if (bytes_read == 0) {
            // End of file
            break;
        }
//        buffer[bytes_read] = '\0'; // add null terminator
        if (sendto(data_sock, buffer, bytes_read, 0, (struct sockaddr *) &server_addr,
sizeof(server_addr)) < 0) {
            perror("Send failed\n");
            return 1;
        }
    }
    printf("sending end\n");
    send(info_sock, end, strlen(end), 0);

    close(data_sock);
    close(info_sock);
    return 0;
}

int server_UDP_B(char *port, int info_sock, long bytes_target, long checksum_target, int
q) {
    struct timeval start, end, diff;
    //open udp sock
    struct sockaddr_in servaddr, cliaddr;


    // Creating socket file descriptor
    int data_sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (data_sock < 0) {
        perror("socket creation failed\n");
        return 1;
    }

    memset(&servaddr, 0, sizeof(servaddr));
    memset(&cliaddr, 0, sizeof(cliaddr));
    unsigned int len;
    len = sizeof(cliaddr);

    // Filling server information
    int port_int = atoi(port);
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(port_int);

    // Bind the socket with the server address
    if (bind(data_sock, (const struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
        perror("bind failed\n");
        return 1;
    }


    struct pollfd fds[2];
    fds[0].fd = info_sock;
    fds[0].events = POLLIN; // tell me when I can read from it
    fds[1].fd = data_sock;
    fds[1].events = POLLIN;
    long bytes_recived = 0;
    long checksum_sum = 0;
    int done = 0;
    int started = 0;
    char buffer[B_SIZE_UDP];
    char buffer_str[B_SIZE_UDP + 1];
    while (1) {
        memset(buffer, 0, sizeof(buffer));
        int err = poll(fds, 2, -1);
        if (err < 0) {
            perror("poll failed\n");
            return 1;
        }
        if (fds[0].revents && POLLIN) {
            //read from the socket
            if (!started)
                recv(info_sock, buffer, strlen("start") + 1, 0);
            else
                recv(info_sock, buffer, strlen("end") + 1, 0);
            if (strcmp(buffer, "start") == 0) {
                gettimeofday(&start, NULL);
                started = 1;
            } else if (strcmp(buffer, "end") == 0) {
                gettimeofday(&end, NULL);
                done = 1;
            }
        }
        else if (fds[1].revents && POLLIN) {

            bytes_recived = recvfrom(data_sock, (char *) buffer, B_SIZE_UDP, 0, (struct
sockaddr *) &cliaddr, &len);
            strcpy(buffer_str, buffer);
            buffer_str[B_SIZE_UDP] = '\0';
            checksum_sum += checksum(buffer_str, B_SIZE_UDP + 1);
        }
        if (done == 1) {

            //set the socket to non-blocking code
            int flags = fcntl(data_sock, F_GETFL, 0);
            fcntl(data_sock, F_SETFL, flags | O_NONBLOCK);

            // receive data for one second
            time_t start_time = time(NULL);
            while (time(NULL) - start_time <= 5) {
                long bytes_temp = 0;
                bytes_temp = recvfrom(data_sock, buffer, B_SIZE_UDP, 0, (struct sockaddr
*) &cliaddr, &len);
                if (bytes_temp > 0) {
                    bytes_recived += bytes_temp;
                    strcpy(buffer_str, buffer);
                    buffer_str[B_SIZE_UDP] = '\0';
                    checksum_sum += checksum(buffer_str, B_SIZE_UDP + 1);
                    gettimeofday(&end, NULL);
                    start_time = time(NULL);
                }
            }
            break;
        }
    }
    close(data_sock);
    timersub(&end, &start, &diff);
    if (!q) {
        printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
    }
    // compare checksum and bytes
    if (bytes_recived != bytes_target) {
        if (!q)
            printf("error: did not received full data!\n");
        else
            printf("failure\n");
        return 1;
    }
    if (checksum_target != checksum_sum) {
        if (!q) {
            printf("error: checksum failed\n");
            printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
        } else {
            printf("failure\n");
        }
        return 1;
    }
    //print results
    long microsec = diff.tv_usec;
    long milisec = microsec / 1000;
    milisec += diff.tv_sec * 1000;
    printf("ipv4_udp,%ld\n", milisec);
    return 0;
}


int client_UDP_IPV6_B(char *ip, char *port, int info_sock, FILE *file) {
    // open udp sock
    int data_sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (data_sock < 0) {
        perror("Error creating socket\n");
        return 1;
    }
    int int_port = atoi(port);
    struct sockaddr_in6 servaddr;
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin6_family = AF_INET6;
    servaddr.sin6_port = htons(int_port);
    if (inet_pton(AF_INET6, ip, &servaddr.sin6_addr) <= 0) {
        perror("inet_pton failed");
        return 1;
    }
    char buffer[B_SIZE_UDP_IPV6];
    size_t bytes_read;
    char *start = "start";
    char *end = "end";
    send(info_sock, start, strlen(start), 0);
    fseek(file, 0L, SEEK_SET);
    while (1) {
        bytes_read = fread(buffer, 1, B_SIZE_UDP_IPV6, file);

        if (bytes_read == 0) {
            // End of file
            break;
        }
//        buffer[bytes_read] = '\0'; // add null terminator
        if (sendto(data_sock, buffer, bytes_read, 0, (struct sockaddr *) &servaddr,
sizeof(servaddr)) < 0) {
            perror("Send failed");
            return 1;
        }
    }
    send(info_sock, end, strlen(end), 0);

    close(data_sock);
    close(info_sock);
    return 0;
}

int server_UDP_IPV6_B(char *port, int info_sock, long bytes_target, long checksum_target,
int q) {
    struct timeval start, end, diff;
    //open udp sock
    struct sockaddr_in6 servaddr, cliaddr;

    // Creating socket file descriptor
    int data_sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (data_sock < 0) {
        perror("socket creation failed\n");
        return 1;
    }

    memset(&servaddr, 0, sizeof(servaddr));
    memset(&cliaddr, 0, sizeof(cliaddr));
    unsigned int len;
    len = sizeof(cliaddr);

    // Filling server information
    // Bind the socket to a port
    int int_port = atoi(port);
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin6_family = AF_INET6;
    servaddr.sin6_port = htons(int_port);
    servaddr.sin6_addr = in6addr_any;
    if (bind(data_sock, (const struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
        perror("bind failed\n");
        return 1;
    }

    struct pollfd fds[2];
    fds[0].fd = info_sock;
    fds[0].events = POLLIN; // tell me when I can read from it
    fds[1].fd = data_sock;
    fds[1].events = POLLIN;
    long bytes_recived = 0;
    long checksum_sum = 0;
    int done = 0;
    int started = 0;
    char buffer[B_SIZE_UDP_IPV6];
    char buffer_str[B_SIZE_UDP_IPV6 + 1];
    while (1) {
        memset(buffer, 0, sizeof(buffer));
        int err = poll(fds, 2, -1);
        if (err < 0) {
            perror("poll failed\n");
            return 1;
        }
        if (fds[0].revents && POLLIN) {
            //read from the socket
            if (!started)
                recv(info_sock, buffer, strlen("start") + 1, 0);
            else
                recv(info_sock, buffer, strlen("end") + 1, 0);
            if (strcmp(buffer, "start") == 0) {
                gettimeofday(&start, NULL);
                started = 1;
            } else if (strcmp(buffer, "end") == 0) {
                gettimeofday(&end, NULL);
                done = 1;
            }
        }
        if (fds[1].revents && POLLIN ) {

            bytes_recived += recvfrom(data_sock, (char *) buffer, B_SIZE_UDP_IPV6, 0,
(struct sockaddr *) &cliaddr,
                                      &len);
            strcpy(buffer_str, buffer);
            buffer_str[B_SIZE_UDP_IPV6] = '\0';
            checksum_sum += checksum(buffer_str, B_SIZE_UDP_IPV6 + 1);

        }
        if (done == 1) {
            //set the socket to non-blocking code
            int flags = fcntl(data_sock, F_GETFL, 0);
            fcntl(data_sock, F_SETFL, flags | O_NONBLOCK);

            // receive data for one second
            time_t start_time = time(NULL);
            while (time(NULL) - start_time <= 1) {

                long bytes_temp = 0;
                bytes_temp = recvfrom(data_sock, buffer, B_SIZE_UDP_IPV6, 0, (struct
sockaddr *) &cliaddr, &len);
                if (bytes_temp  >= 0) {
                    bytes_recived += bytes_temp;
                    strcpy(buffer_str, buffer);
                    buffer_str[B_SIZE_UDP_IPV6] = '\0';
                    checksum_sum += checksum(buffer_str, B_SIZE_UDP_IPV6 + 1);
                    gettimeofday(&end, NULL);
                    start_time = time(NULL);
                }
            }
            break;
        }
    }
    close(data_sock);
    timersub(&end, &start, &diff);
    if (!q) {
        printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
    }
    // compare checksum and bytes
    if (bytes_recived != bytes_target) {
        printf("failure\n");
        return 1;
    }
    if (checksum_target != checksum_sum) {
        if(!q) {
            printf("error: checksum failed\n");
            printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
        } else{
            printf("failure\n");
        }

        return 1;
    }
    //print results
    long microsec = diff.tv_usec;
    long milisec = microsec / 1000;
    milisec += diff.tv_sec * 1000;
    printf("ipv6_udp,%ld\n", milisec);
    return 0;
}


int client_UDS_DGRAM(int info_sock, FILE *file, long bytes_target){
    int data_sock;
    socklen_t len;
    char buffer[B_SIZE];
    char *start = "start";
    char *end = "end";
    size_t bytes_read;
    long total_bytes = 0;

    struct sockaddr_un remote = {
        .sun_family = AF_UNIX,
        // .sin_path = SOCK_PATH //cant assign to an array
    };

    if((data_sock = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1){
        perror("socket");
        exit(EXIT_FAILURE);
    }

    strcpy(remote.sun_path, SOCK_PATH);
    len = strlen(remote.sun_path) + sizeof(remote.sun_family);

    if(send(info_sock, start, strlen(start), 0) <= 0){ //start timer
        perror("send start");
        close(data_sock);
        close(info_sock);
        exit(EXIT_FAILURE);
```

```c
    }
    fseek(file, 0L, SEEK_SET);
    while (total_bytes < bytes_target) {
        bytes_read = fread(buffer, 1, B_SIZE, file);
        total_bytes += bytes_read;

        //buffer[bytes_read] = '\0'; // add null terminator
        if (sendto(data_sock, buffer, bytes_read, 0, (struct sockaddr *) &remote, len) ==
-1) {
            perror("sendto");
            close(data_sock);
            close(info_sock);
            exit(EXIT_FAILURE);
        }
    }
    if(send(info_sock, end, strlen(end), 0) == -1){ //end timer
        perror("send end");
        close(data_sock);
        close(info_sock);
        exit(EXIT_FAILURE);
    }
    close(data_sock);
    close(info_sock);
    return 0;

}

int server_UDS_DGRAM(int info_sock, long bytes_target, long checksum_target, int q){
    int sock, bytes = 0,  done = 0, started = 0, i = 0;
    socklen_t local_len, remote_len;
    long bytes_recived = 0, checksum_sum = 0;
    char buffer[B_SIZE], buffer_str[B_SIZE + 1] = "";

    struct timeval start, end, diff;
    struct sockaddr_un remote, local = {
        .sun_family = AF_UNIX,
        // .sin_path = SOCK_PATH //cant assign to an array
    };

    strcpy(local.sun_path, SOCK_PATH);
    unlink(local.sun_path);
    local_len = strlen(local.sun_path) + sizeof(local.sun_family);
    remote_len = sizeof(remote);

    if((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1){
        perror("socket");
        exit(EXIT_FAILURE);
    }

    if(bind(sock, (struct sockaddr *) &local, local_len) == -1){
        perror("bind");
        close(sock);
        exit(EXIT_FAILURE);
    }

    struct pollfd fds[2];
    fds[0].fd = info_sock;
    fds[0].events = POLLIN; // tell me when I can read from it
    fds[1].fd = sock;
    fds[1].events = POLLIN;

    while(1){
        memset(buffer, 0, sizeof(buffer));

        if(poll(fds, 2, -1) < 0){
            perror("poll");
            close(sock);
            exit(EXIT_FAILURE);
        }

        if (fds[0].revents && POLLIN) {
            //read from the socket
            if (!started){
                recv(info_sock, buffer, strlen("start") + 1, 0);
            }else{
                recv(info_sock, buffer, strlen("end") + 1, 0);
            }if (strcmp(buffer, "start") == 0) {
                gettimeofday(&start, NULL);
                started = 1;
            } else if (strcmp(buffer, "end") == 0) {
                gettimeofday(&end, NULL);
                done = 1;
            }
        }

        if (fds[1].revents && POLLIN) {
            // recive the data and count byts
            bytes = recvfrom(sock, buffer, sizeof(buffer), 0, (struct sockaddr *)
&remote, &remote_len);
            if(bytes < 0){
                perror("recvfrom");
                close(sock);
                exit(EXIT_FAILURE);
            }

            bytes_recived += bytes;
            strcat(buffer_str, buffer);
            buffer_str[B_SIZE] = '\0';
            checksum_sum += checksum(buffer_str, B_SIZE + 1);
        }

        if(done){
            break;
        }
    }
    close(sock);
    timersub(&end, &start, &diff);
    if (!q) {
        printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
    }
    // compare checksum and bytes
    if (bytes_recived != bytes_target) {
        if (!q)
            printf("error: did not received full data!\n");
        else
            printf("failure\n");
        return 1;
    }
    if (checksum_target != checksum_sum) {
        if (!q) {
            printf("error: checksum failed\n");
            printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
        } else {
            printf("failure\n");
        }

        return 1;
    }

    //print results
    long microsec = diff.tv_usec;
    long milisec = microsec / 1000;
    milisec += diff.tv_sec * 1000;
    printf("uds_dgram,%ld\n", milisec);
    return 0;

}


int client_UDS_STREAM(int info_sock, FILE *file, long bytes_target){
    int data_sock, len;
    char buffer[B_SIZE];
    char *start = "start";
    char *end = "end";
    size_t bytes_read;
    long total_bytes = 0;

    struct sockaddr_un remote = {
        .sun_family = AF_UNIX,
        // .sin_path = SOCK_PATH //cant assign to an array
    };

    if((data_sock = socket(AF_UNIX, SOCK_STREAM, 0)) == -1){
        perror("socket");
        exit(EXIT_FAILURE);
    }

    strcpy(remote.sun_path, SOCK_PATH);
    len = strlen(remote.sun_path) + sizeof(remote.sun_family);
    if(connect(data_sock, (struct sockaddr *) &remote, len) == -1){
        perror("connect");
        exit(EXIT_FAILURE);
    }

    send(info_sock, start, strlen(start), 0); //start timer
    fseek(file, 0L, SEEK_SET);
    while (total_bytes < bytes_target) {
        bytes_read = fread(buffer, 1, B_SIZE, file);
        total_bytes += bytes_read;

        //buffer[bytes_read] = '\0'; // add null terminator
        if (send(data_sock, buffer, bytes_read, 0) == -1) {
            perror("send");
            exit(EXIT_FAILURE);
        }
    }
    send(info_sock, end, strlen(end), 0); //end timer

    close(data_sock);
    close(info_sock);
```

```c
    return 0;

}

int server_UDS_STREAM(int info_sock, long bytes_target, long checksum_target, int q){
    int sock1, sock2, len, done = 0, started = 0;
    long bytes_recived = 0, checksum_sum = 0;
    char buffer[B_SIZE], buffer_str[B_SIZE + 1] = "";

    struct timeval start, end, diff;
    struct sockaddr_un remote, local = {
        .sun_family = AF_UNIX,
        // .sin_path = SOCK_PATH //cant assign to an array
    };

    if((sock1 = socket(AF_UNIX, SOCK_STREAM, 0)) == -1){
        perror("socket");
        exit(EXIT_FAILURE);
    }

    strcpy(local.sun_path, SOCK_PATH);
    unlink(local.sun_path);
    len = strlen(local.sun_path) + sizeof(local.sun_family);

    if(bind(sock1, (struct sockaddr *) &local, len) == -1){
        perror("bind");
        exit(EXIT_FAILURE);
    }

    if(listen(sock1, 1) == -1){
        perror("listen");
        exit(EXIT_FAILURE);
    }

    if((sock2 = accept(sock1, (struct sockaddr *) &remote, &len)) == -1){
        perror("accept");
        exit(EXIT_FAILURE);
    }

    close(sock1);

    struct pollfd fds[2];
    fds[0].fd = info_sock;
    fds[0].events = POLLIN; // tell me when I can read from it
    fds[1].fd = sock2;
    fds[1].events = POLLIN;

    while(1){
        memset(buffer, 0, sizeof(buffer));

        if(poll(fds, 2, -1) == -1){
            perror("poll");
            exit(EXIT_FAILURE);
        }

        if (fds[0].revents && POLLIN) {
            //read from the socket
            if (!started)
                recv(info_sock, buffer, strlen("start") + 1, 0);
            else
```

```
                recv(info_sock, buffer, strlen("end") + 1, 0);
            if (strcmp(buffer, "start") == 0) {
                gettimeofday(&start, NULL);
                started = 1;
            } else if (strcmp(buffer, "end") == 0) {
                gettimeofday(&end, NULL);
                done = 1;
            }
        }

        else if (fds[1].revents && POLLIN) {
            // recive the data and count byts
            bytes_recived += recv(sock2, buffer, sizeof(buffer), 0);
            strcat(buffer_str, buffer);
            buffer_str[B_SIZE] = '\0';
            checksum_sum += checksum(buffer_str, B_SIZE + 1);
        }

        if(done){
            break;
        }
    }
    close(sock2);
    timersub(&end, &start, &diff);
    if (!q) {
        printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
    }
    // compare checksum and bytes
    if (bytes_recived != bytes_target) {
        if (!q)
            printf("error: did not received full data!\n");
        else
            printf("failure\n");
        return 1;
    }
    if (checksum_target != checksum_sum) {
        if (!q) {
            printf("error: checksum failed\n");
            printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
        } else {
            printf("failure\n");
        }

        return 1;
    }

    //print results
    long microsec = diff.tv_usec;
    long milisec = microsec / 1000;
    milisec += diff.tv_sec * 1000;
    printf("uds_stream,%ld\n", milisec);
    return 0;

}


int client_mmap(int info_sock , FILE *file, long bytes_target){
    int fd;
    char *data;
    long total_bytes = 0;
```

```c

    fd = fileno(file);
    ftruncate(fd, B_SIZE);

    char buffer[B_SIZE];
    size_t bytes_read;
    char *start = "start";
    char *end = "end";
    if(send(info_sock, start, strlen(start), 0) == -1){
        perror("failed sending start");
        exit(EXIT_FAILURE);
    }

    data = mmap(NULL, B_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if(data == MAP_FAILED){
        perror("failed mapping data");
        exit(EXIT_FAILURE);
    }

    if(send(info_sock, end, strlen(end), 0) == -1){
        printf("failed!!!\n");
        perror("failed sending end");
        exit(EXIT_FAILURE);
    }

    close(fd);
    close(info_sock);
}

int server_mmap(int info_sock, long bytes_target, long checksum_target, int q){
    int fd, started = 0, done = 0, i = 0;
    long bytes_recived = 0, checksum_sum = 0;
    char buffer[B_SIZE], buffer_str[B_SIZE + 1], *data;

    struct timeval start, end, diff;
    struct pollfd fds[2];

    fds[0].fd = info_sock;
    fds[0].events = POLLIN;
    fds[1].fd = fd;
    fds[1].events = POLLIN;

    while(1){
        memset(buffer, 0, sizeof(buffer));
        if(poll(fds, 2, -1) == -1){
            perror("poll failed");
            exit(EXIT_FAILURE);
        }

        if (fds[0].revents && POLLIN) {
            //read from the socket
            if (!started){
                if(recv(info_sock, buffer, strlen("start") + 1, 0) == -1){
                    perror("failed receiving start");
                    exit(EXIT_FAILURE);
                }
            }else{
                if(recv(info_sock, buffer, strlen("end") + 1, 0) == -1){
                    perror("failed receiving end");
                    exit(EXIT_FAILURE);
                }
            }
            if (strcmp(buffer, "start") == 0) {
                gettimeofday(&start, NULL);
                started = 1;
            } else if (strcmp(buffer, "end") == 0) {
                gettimeofday(&end, NULL);
                done = 1;
            }
        }

        else if(fds[1].revents && POLLIN){
            data = mmap(NULL, B_SIZE, PROT_READ, MAP_SHARED, fd, 0);
            if(data == MAP_FAILED){
                perror("failed sending start");
                close(fd);
                exit(EXIT_FAILURE);
            }
            strcpy(buffer_str, data);
            buffer_str[B_SIZE] = '\0';
            checksum_sum += checksum(buffer_str, B_SIZE + 1);
        }
        if(done){
            break;
        }
    }
    close(fd);
    timersub(&end, &start, &diff);
    if (!q) {
        printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
    }
    // compare checksum and bytes
    if (bytes_recived != bytes_target) {
        if (!q)
            printf("error: did not received full data!\n");
        else
            printf("failure\n");
        return 1;
    }
    if (checksum_target != checksum_sum) {
        if (!q) {
            printf("error: checksum failed\n");
            printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
        } else {
            printf("failure\n");
        }

        return 1;
    }

    //print results
    long microsec = diff.tv_usec;
    long milisec = microsec / 1000;
    milisec += diff.tv_sec * 1000;
    printf("mmap,%ld\n", milisec);
    return 0;
}


int client_named_pipe(int info_sock ,char *fifo_name, FILE *file, long bytes_target){
    int fd;
    long total_bytes = 0;

    mkfifo(fifo_name, 0666);

    fd = open(fifo_name, O_WRONLY);
    if(fd == -1){
        perror("failed to open the pipe\n");
        exit(EXIT_FAILURE);
    }

    char buffer[B_SIZE];
    size_t bytes_read;
    char *start = "start";
    char *end = "end";
    if(send(info_sock, start, strlen(start), 0) == -1){
        perror("failed sending start\n");
        exit(EXIT_FAILURE);
    }

    fseek(file, 0L, SEEK_SET);
    while (total_bytes < bytes_target) {
        bytes_read = fread(buffer, 1 , B_SIZE, file);
        total_bytes += bytes_read;
        write(fd, buffer, bytes_read);
    }

    if(send(info_sock, end, strlen(end), 0) == -1){
        perror("failed sending end\n");
        exit(EXIT_FAILURE);
    }

    close(fd);
    close(info_sock);
}

int server_named_pipe(int info_sock, char *fifo_name, long bytes_target, long
checksum_target, int q){
    int fd, started = 0, done = 0, i = 0;
    long bytes_recived = 0, checksum_sum = 0;
    char buffer[B_SIZE], buffer_str[B_SIZE + 1];

    mkfifo(fifo_name, 0666);

    fd = open(fifo_name,O_RDONLY);
    if(fd == -1){
        perror("failed to open the pipe\n");
        exit(EXIT_FAILURE);
    }

    struct timeval start, end, diff;
    struct pollfd fds[2];

    fds[0].fd = info_sock;
    fds[0].events = POLLIN;
    fds[1].fd = fd;
    fds[1].events = POLLIN;

    while(1){
        memset(buffer, 0, sizeof(buffer));
        if(poll(fds, 2, -1) == -1){
            perror("poll failed\n");
            exit(EXIT_FAILURE);
        }

        if (fds[0].revents && POLLIN) {
            //read from the socket
            if (!started){
                if(recv(info_sock, buffer, strlen("start") + 1, 0) == -1){
                    perror("failed receiving start\n");
                    exit(EXIT_FAILURE);
                }
            }else{
                if(recv(info_sock, buffer, strlen("end") + 1, 0) == -1){
                    perror("failed receiving end\n");
                    exit(EXIT_FAILURE);
                }
            }
            if (strcmp(buffer, "start") == 0) {
                gettimeofday(&start, NULL);
                started = 1;
            } else if (strcmp(buffer, "end") == 0) {
                gettimeofday(&end, NULL);
                done = 1;
            }
        }

        else if(fds[1].revents && POLLIN){
            while(1){
                int bytes = read(fd, buffer, sizeof(buffer));

                if(bytes > 0){
                    bytes_recived += bytes;
                }
                else if(bytes == 0){
                    break;
                }
                else{
                    perror("error while readinf the file\n");
                    exit(EXIT_FAILURE);
                }
            }
            strcpy(buffer_str, buffer);
            buffer_str[B_SIZE] = '\0';
            checksum_sum += checksum(buffer_str, B_SIZE + 1);
        }
        if(done){
            break;
        }
    }
    close(fd);
    timersub(&end, &start, &diff);
    if (!q) {
        printf("expected: %ld ,got: %ld\n", bytes_target, bytes_recived);
    }
    // compare checksum and bytes
    if (bytes_recived != bytes_target) {
        if (!q)
            printf("error: did not received full data!\n");
        else
```

```c
            printf("failure\n");
        return 1;
    }
    if (checksum_target != checksum_sum) {
        if (!q) {
            printf("error: checksum failed\n");
            printf("expected: %ld ,got: %ld\n", checksum_target, checksum_sum);
        } else {
            printf("failure\n");
        }

        return 1;
    }

    //print results
    long microsec = diff.tv_usec;
    long milisec = microsec / 1000;
    milisec += diff.tv_sec * 1000;
    printf("pipe,%ld\n", milisec);
    return 0;
}


int server_B(char *port, int q) {
    char info_port[6];
    port_for_info(port, info_port);
    int info_sock = tcp_server_conn(info_port);

    char message[100] = {'\0'};

    recv(info_sock, message, sizeof(message), 0);

    char *type = strtok(message, ",");
    char *param = strtok(NULL, ",");
    char *checksum_target = strtok(NULL, ",");
    char *bytes_target = strtok(NULL, ",");

    long bytes_target_long = atol(bytes_target);
    long checksum_target_long = atol(checksum_target);

    int ret = 0;
    if (!q) {
        printf("type: %s\n", type);
        printf("param: %s\n", param);
    }
    if (strcmp(type, "ipv4") == 0) {
        if (strcmp(param, "tcp") == 0) {
            ret = server_TCP_B(port, info_sock, bytes_target_long, checksum_target_long,
q);

        } else if (strcmp(param, "udp") == 0) {
            ret = server_UDP_B(port, info_sock, bytes_target_long, checksum_target_long,
q);
        }

    } else if (strcmp(type, "ipv6") == 0) {
        if (strcmp(param, "tcp") == 0) {
            server_TCP_IPV6_B(port,info_sock,bytes_target_long,checksum_target_long,q);
        } else if (strcmp(param, "udp") == 0) {
            ret = server_UDP_IPV6_B(port, info_sock, bytes_target_long,
checksum_target_long, q);
```

```
        }

    } else if (strcmp(type, "uds") == 0) {
        if (strcmp(param, "dgram") == 0) {
            ret = server_UDS_DGRAM(info_sock, bytes_target_long, checksum_target_long,
q);
        } else if (strcmp(param, "stream") == 0) {
            ret = server_UDS_STREAM(info_sock, bytes_target_long, checksum_target_long,
q);
        }
    } else if (strcmp(type, "mmap") == 0) {
        ret = server_mmap(info_sock, bytes_target_long, checksum_target_long, q);

    } else if (strcmp(type, "pipe") == 0) {
        ret = server_named_pipe(info_sock, FIFO_NAME, bytes_target_long,
checksum_target_long, q);

    } else {
        perror("wrong parameters\n");
        return 1;
    }
    close(info_sock);
    return ret;
}


int main(int argc, char *argv[]) {
    int is_tcp = 0;
    int is_udp = 0;
    int is_ip = 0;
    int is_port = 0;
    int is_ipv4 = 0;
    int is_ipv6 = 0;
    int is_uds = 0;
    int is_dgram = 0;
    int is_stream = 0;
    int is_mmap = 0;
    int is_pipe = 0;
    int is_c = 0;
    int is_p = 0;
    int is_q = 0;
    int is_server = 0;
    char file_name[50] = {'\0'};
    char ip[40] = {'\0'};
    char port[10] = {'\0'};
    if (argc < 3) {
        usage();
        exit(EXIT_FAILURE);
    }
    for (int i = 1; i < argc; ++i) {
        if (strcmp(argv[i], "-c") == 0) {
            is_c = 1;
        } else if (strcmp(argv[i], "-s") == 0) {
            is_server = 1;
        } else if (strcmp(argv[i], "-p") == 0) {
            is_p = 1;
        } else if (strcmp(argv[i], "-q") == 0) {
            is_q = 1;
        } else if (strcmp(argv[i], "pipe") == 0) {
            is_pipe = 1;
            if (i + 1 >= argc) {
                usage();
                exit(EXIT_FAILURE);
            }
            strcpy(file_name, argv[i + 1]);
            i++;
        } else if (strcmp(argv[i], "mmap") == 0) {
            is_mmap = 1;
            if (i + 1 >= argc) {
                usage();
                exit(EXIT_FAILURE);
            }
            strcpy(file_name, argv[i + 1]);
            i++;
        } else if (strcmp(argv[i], "uds") == 0) {
            is_uds = 1;
        } else if (strcmp(argv[i], "ipv6") == 0) {
            is_ipv6 = 1;
        } else if (strcmp(argv[i], "ipv4") == 0) {
            is_ipv4 = 1;
        } else if (strcmp(argv[i], "tcp") == 0) {
            is_tcp = 1;
        } else if (strcmp(argv[i], "udp") == 0) {
            is_udp = 1;
        } else if (strstr(argv[i], ".") != NULL) {
            is_ip = 1;
            strcpy(ip, argv[i]);
        } else if (strstr(argv[i], ":") != NULL) {
            is_ip = 1;
            strcpy(ip, argv[i]);
        } else if (strcmp(argv[i], "dgram") == 0) {
            is_dgram = 1;
        } else if (strcmp(argv[i], "stream") == 0) {
            is_stream = 1;
        } else {
            is_port = 1;
            strcpy(port, argv[i]);
        }
    }
    if (!is_p) { //part A
        if (is_c) { //client A
            if (argc != 4) {
                usage();
                exit(EXIT_FAILURE);
            }
            if (!is_port || !is_ip) {
                usage();
                exit(EXIT_FAILURE);
            }
            if (client_A(port, ip) != 0)
                exit(EXIT_FAILURE);

        } else if (is_server) { // server A
            if (argc != 3) {
                usage();
                exit(EXIT_FAILURE);
            }
            if (!is_port) {
                usage();
                exit(EXIT_FAILURE);
            }
            if (server_A(port) != 0)
                exit(EXIT_FAILURE);
        } else {
            usage();
            exit(EXIT_FAILURE);
        }
    } else { // part B
        if (is_c) { // client side
            FILE *file;
            file = fopen("100MB.bin", "ab+");
            if (file == NULL) {
                perror("File open failed\n");
                exit(EXIT_FAILURE);
            }
            long bytes_count = 0;
            long checksum = checksum_file(file, &bytes_count);

            char new_port[6];
            port_for_info(port, new_port);
            char info_ip[40] = {'\0'};
            if (is_ipv6)
                ipv6_to_ipv4(ip, info_ip);
            else
                strcpy(info_ip, ip);
            int info_sock = tcp_client_conn(info_ip, new_port);


            if (is_ipv4 && is_tcp) { // ipv4 - tcp
                //socket to notify the receiver to start timing
                type_param(info_sock, "ipv4", "tcp", checksum, bytes_count);
                client_TCP_B(ip, port, info_sock, file);

            } else if (is_ipv4 && is_udp) { // ipv4 - udp
                //socket to notify the receiver to start timing
                type_param(info_sock, "ipv4", "udp", checksum, bytes_count);
                client_UDP_B(ip, port, info_sock, file);
            } else if (is_ipv6 && is_tcp) { // ipv6- tcp
                //socket to notify the receiver to start timing
                type_param(info_sock, "ipv6", "tcp", checksum, bytes_count);
                client_TCP_IPV6_B(ip,port,info_sock,file);
            } else if (is_ipv6 && is_udp) { // ipv6 - udp
                //socket to notify the receiver to start timing
                type_param(info_sock, "ipv6", "udp", checksum, bytes_count);
                client_UDP_IPV6_B(ip, port, info_sock, file);

            } else if (is_uds && is_dgram) { // uds dgram
                //socket to notify the receiver to start timing
                type_param(info_sock, "uds", "dgram", checksum, bytes_count);
                client_UDS_DGRAM(info_sock, file, bytes_count);

            } else if (is_uds && is_stream) {// uds stream
                //socket to notify the receiver to start timing
                type_param(info_sock, "uds", "stream", checksum, bytes_count);
                client_UDS_STREAM(info_sock, file, bytes_count);

            } else if (is_mmap) { // mmap
                //socket to notify the receiver to start timing
                type_param(info_sock, "mmap", "filename", checksum, bytes_count);
                client_mmap(info_sock, file, bytes_count);

```

```c
            } else if (is_pipe) { // pipe
                //socket to notify the receiver to start timing
                type_param(info_sock, "pipe", "filename", checksum, bytes_count);
                client_named_pipe(info_sock, FIFO_NAME, file, bytes_count);

            } else {
                fclose(file);
                usage();
                exit(EXIT_FAILURE);
            }
            fclose(file);
        } else if (is_server) {
            server_B(port, is_q); //continue as long as server_B returns 0;
            return 0;
        } else {
            usage();
            exit(EXIT_FAILURE);
        }
    }
    return 0;
}
```

Assignments\Assignment3\makefile

```makefile
all: stnc

clean:
	rm -f *.o OS_EX3_pipe stnc
.PHONY: all clean

stnc:stnc.c
	gcc stnc.c -o stnc


```