RELAZIONE CHAT CLIENT/SERVER MULTITHREAD BASATA SU TOR

Daniele Intra – Matricola 01507A

Requisiti:

- Sistema operativo Linux
  - Ambiente utilizzato: Debian GNU/Linux 12 (bookworm) x86_64
  - Kernel: 6.1.0-12-amd64
- IDE di sviluppo: Visual Studio Code, v-1.82.0
- Librerie utilizzate:

#include *<stdio.h>*
Questa libreria fornisce funzioni per l'input e l'output standard, come `printf` e `scanf`. È utilizzata per la gestione di operazioni di input/output.
#include *<stdlib.h>*
Questa libreria contiene funzioni per la gestione della memoria dinamica, come `malloc` e `free`, e altre funzioni utili come `exit`.
#include *<string.h>*
Questa libreria fornisce funzioni per la gestione delle stringhe, come `strcpy` e `strlen`.
#include *<unistd.h>*
Questa libreria contiene funzioni e costanti relative alle chiamate di sistema Unix, tra cui `read`, `write`, e `close`. È utilizzata per la gestione dei file descriptor e delle operazioni di I/O a basso livello.
#include *<arpa/inet.h>*
Questa libreria fornisce funzioni e strutture dati per la gestione degli indirizzi IP e delle operazioni di rete, come la conversione tra indirizzi IP e rappresentazioni testuali.
#include *<sys/socket.h>*
Questa libreria è utilizzata per la gestione dei socket, inclusi socket di rete. Contiene funzioni e definizioni per la creazione, la configurazione e l'uso dei socket.
#include *<pthread.h>*
Questa libreria è utilizzata per la programmazione multithreading in C. Fornisce funzioni e strutture dati per la creazione e la gestione di **thread**.
#include *<signal.h>*
Questa libreria è utilizzata per la gestione dei segnali, che sono eventi asincroni generati dal sistema operativo. È utilizzata per registrare gestori di segnali personalizzati.

#include *<curl/curl.h>*
Questa libreria fornisce l'interfaccia per l'uso di libcurl, una libreria ampiamente utilizzata per l'accesso a risorse su Internet, inclusi download di file e richieste HTTP. L'ho utilizzata nel codice del client per poter implementare la funzione che controlla

l'indirizzo ip da cui parte la richiesta e quindi verificare che sia passato davvero dal proxy socks5 (tor)
#define _GNU_SOURCE
Questa è una direttiva di precompilazione che abilita alcune estensioni GNU specifiche nel tuo programma. Ho provato ad inserirla per risolvere un errore di tipo indefinito quando creavo la **struct per** signal (gestione dell'uscita dal server mediante segnale di interrupt ctrl+c) ma si è rilevata inutile (problema spiegato successivamente)

Descrizione del progetto

Questo progetto è costituito da uno o più client e un server di chat. Il client consente agli utenti di connettersi al server (passando per Tor), inviare messaggi a tutti gli altri client connessi e ricevere messaggi dai client. Ecco una panoramica delle principali funzionalità del client:

1. **Verifica dell'IP Pubblico:** Il client utilizza la libreria *libcurl* per ottenere il suo indirizzo IP pubblico e lo mostra durante la connessione → capisco se tor funziona
2. **Connessione al Server**: Il client si connette a un server specifico, identificato dall'indirizzo IP e dalla porta.
3. **Inserimento del Nome**: L'utente inserisce un nome utente che verrà utilizzato per identificare il mittente dei messaggi inviati al server.
4. **Comunicazione con il Server**: Il client può inviare messaggi di testo al server, che verranno quindi inoltrati a tutti gli altri client connessi.
5. **Disconnessione**: L'utente può digitare "**exit**" per disconnettersi dal server. Un messaggio di notifica viene inviato agli altri client per informarli della disconnessione.
6. **Ricezione dei Messaggi**: Il client utilizza un ***thread separato*** per la ricezione continua dei messaggi dai altri client. I messaggi ricevuti vengono visualizzati sulla console.

NB: quando voglio eseguire il client per connettermi, do questo comando:
torsocks ./client
torsocks permette di far passare via rete Tor qualsiasi argomento che gli viene passato successivamente, in questo caso l'eseguibile compilato a partire dal nostro codice client. (*per capire effettivamente se stiamo andando su rete Tor ho inserito una funzione check_ip che fa una richiesta ad un sito che restituisce l'ip e determina se è un ip tor oppure l'ip pubblico del modem di casa/università*)
La soluzione migliore (così in locale non ha molto senso) sarebbe avere l'eseguibile del server su una macchina remota, così da sfruttare a pieno la potenzialità della rete Tor. → vedi parte finale documento

Codice commentato del Client.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <pthread.h>
#include <curl/curl.h>
//ricordo che il file viene eseguito come -> torsocks ./client (già su rete tor)
127.0.0.1:9051 -> installato sull'host linux
#define SERVER_IP "127.0.0.1"   // Indirizzo IP del server
#define SERVER_PORT 8888        // Porta del server
#define BUFFER_SIZE 1024        // Dimensione del buffer per i messaggi

int client_socket;              // Socket del client per la connessione al server
pthread_t receive_thread;       // Thread per la ricezione dei messaggi dal server

// Funzione per ricevere i messaggi dal server (thread separato)
void *receive_messages(void *arg) {
    char buffer[BUFFER_SIZE];   // Buffer per memorizzare i messaggi ricevuti dal server
    ssize_t num_bytes;

    while (1) { //ciclo while per ricevere i dati del server tramite il socket del client
        memset(buffer, 0, sizeof(buffer));
        num_bytes = recv(client_socket, buffer, sizeof(buffer) - 1, 0);
        if (num_bytes <= 0) { //quando non ricevo più nulla esco
            break;
        } else {
            printf("%s\n", buffer); //se invece ricevo qualcosa stampo cosa è arrivato
        }
    }

    printf("Connessione al server terminata.\n"); //quando esco dal ciclo while
    pthread_exit(NULL); //connessione server terminata -> chiudo thread
    return 0; //termino l'esecuzione del client
}

// Funzione per verificare l'indirizzo IP pubblico del client (son su Tor o no???)
void check_ip() {
    CURL *curl;
    CURLcode res; //l'indirizzo nella console viene stampato come JSON
    char *url = "https://api.ipify.org/?format=json"; // Servizio per ottenere l'IP pubblico
```

```
    // Inizializza libcurl
    curl = curl_easy_init();
    if (!curl) {
        fprintf(stderr, "Errore nell'inizializzazione di libcurl\n");
        exit(EXIT_FAILURE);
    }

    // Configura l'URL di destinazione
    curl_easy_setopt(curl, CURLOPT_URL, url);

    // Esegui la richiesta HTTP
    res = curl_easy_perform(curl);
    if (res != CURLE_OK) {
        fprintf(stderr, "Errore nella richiesta HTTP: %s\n", curl_easy_strerror(res));
        curl_easy_cleanup(curl);
        exit(EXIT_FAILURE);
    }

    // Chiudi la connessione HTTP
    curl_easy_cleanup(curl);
}

int main() {
    struct sockaddr_in server_addr; // Struttura per memorizzare l'indirizzo del server
    char buffer[BUFFER_SIZE];      // Buffer per memorizzare i messaggi inviati e ricevuti
    char client_name[BUFFER_SIZE];  // Nome del client

    // Verifica l'indirizzo IP pubblico del client, chiamo funzione check_ip
    check_ip();

    // Creazione del socket del client
    if ((client_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Errore nella creazione del socket");
        exit(EXIT_FAILURE);
    }

    // Impostazione dell'indirizzo del server
    server_addr.sin_family = AF_INET;            // Famiglia di indirizzi (IPv4)
    server_addr.sin_port = htons(SERVER_PORT);    // Porta del server in formato network
byte order

    // Conversione dell'indirizzo IP del server da formato testuale a binario
    if (inet_pton(AF_INET, SERVER_IP, &(server_addr.sin_addr)) <= 0) {
        perror("Errore nell'indirizzo del server");
        exit(EXIT_FAILURE);
```

```
    }

    // Connessione al server
    if (connect(client_socket, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0) {
        perror("Errore nella connessione al server");
        exit(EXIT_FAILURE);
    }

    printf("\nConnessione al server %s:%d avvenuta con successo.\n", SERVER_IP, SERVER_PORT); //qua stamperà server 127.0.0.1:8888

    // Inserimento del nome del client
    printf("Inserisci il tuo nome: ");
    fgets(client_name, sizeof(client_name), stdin);
    client_name[strcspn(client_name, "\n")] = '\0'; // Rimuove il carattere newline dalla stringa

    // Invio del nome del client al server
    if (send(client_socket, client_name, strlen(client_name), 0) < 0) {
        perror("Errore nell'invio del nome del client");
        close(client_socket);
        exit(EXIT_FAILURE);
    }

    // Creazione del thread per la ricezione dei messaggi
   //come funzione al thread gli passo quella di prima -> receive_messages
    if (pthread_create(&receive_thread, NULL, receive_messages, NULL) != 0) {
        perror("Errore nella creazione del thread");
        close(client_socket);
        exit(EXIT_FAILURE);
    }

    // Ciclo di input dei messaggi
    while (1) {
        fgets(buffer, sizeof(buffer), stdin);
        buffer[strcspn(buffer, "\n")] = '\0'; // Rimuove il carattere newline dalla stringa

        if (strcmp(buffer, "exit") == 0) { //quando client scrive exit, termino
            printf("Disconnessione dal server.\n");
      //mando un messaggio al server e agli altri client che ho abbandonato
            if (send(client_socket, "Il client ha abbandonato", 30, 0) < 0) {
                perror("Errore nell'invio del messaggio");
                break; // L'utente ha digitato "exit", usciamo dal ciclo
            }
```

```
            return 0; // Usciamo dal programma
        }

        // Invio del messaggio al server (altrimenti invio al server quello che mi ha scritto)
        if (send(client_socket, buffer, strlen(buffer), 0) < 0) {
            perror("Errore nell'invio del messaggio");
            break;
        }
    }

    // Join del thread di ricezione dei messaggi
    pthread_join(receive_thread, NULL);
Usando pthread_join, il thread principale "si mette in pausa" fino a quando il
receive_thread non è completo
pthread_join(receive_thread, NULL); è posizionato alla fine del programma per assicurarsi
che il thread di ricezione abbia finito di ricevere tutti i messaggi prima di terminare il
programma.

    // Chiusura del socket del client
    close(client_socket);

    return 0;
}
```

Codice commentato del Server.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <pthread.h>
#include <signal.h>

#define MAX_CLIENTS 10 //definisco il numero massimo di client che si possono connettere
#define BUFFER_SIZE 1024 //dimensione del buffer per lo scambio di messaggi

// lista per memorizzare le informazioni sui client connessi
struct ClientInfo {
    int socket;
    char name[BUFFER_SIZE]; //nome client
    struct ClientInfo* next;
};

int server_socket; // Socket del server
int num_clients = 0; // Contatore dei client connessi
int flag = 0; // Flag per gestire la chiusura del server con signal
struct ClientInfo* clients = NULL; // Lista dei client connessi
pthread_mutex_t client_mutex = PTHREAD_MUTEX_INITIALIZER; // Mutex per sincronizzare l'accesso alla lista dei client

// Gestisce la chiusura del server quando viene ricevuto il segnale SIGINT (Ctrl+C)
void handle_shutdown(int sig);

// Funzione eseguita in ogni thread client per la gestione delle comunicazioni
void* handle_client(void* arg);

int main() {
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len;
    pthread_t client_threads[MAX_CLIENTS];

    // Dichiarazione e inizializzazione del gestore del segnale per SIGINT (Ctrl+C)
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = handle_shutdown;

    // Configurazione del gestore del segnale SIGINT
```

```c
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("Errore nell'impostazione del gestore di segnali");
        exit(EXIT_FAILURE);
    }

    // Creazione del socket del server
    if ((server_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Errore nella creazione del socket del server");
        exit(EXIT_FAILURE);
    }

    // Impostazione dell'indirizzo del server
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(8888);

    // Binding del socket del server
    if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("Errore nel binding del socket del server");
        exit(EXIT_FAILURE);
    }

    // Inizio dell'ascolto delle connessioni in arrivo
    if (listen(server_socket, MAX_CLIENTS) < 0) {
        perror("Errore nell'ascolto delle connessioni in arrivo");
        exit(EXIT_FAILURE);
    }

    printf("Server avviato. In attesa di connessioni...\n"); //qua va tutto bene

    while (!flag) { //finchè non è arrivato nessun segnale ctrl+c
        client_addr_len = sizeof(client_addr);
        int client_socket;

        // Accettazione di una nuova connessione
        if ((client_socket = accept(server_socket, (struct sockaddr*)&client_addr,
&client_addr_len)) < 0) {
            perror("Errore nell'accettazione di una nuova connessione");
            exit(EXIT_FAILURE);
        }

        // Verifica se ci sono slot liberi per nuovi client
        if (num_clients >= MAX_CLIENTS) {
            printf("Numero massimo di client raggiunto. Nuova connessione rifiutata.\n");
            close(client_socket);
```

```
            continue;
        }

        // Mutex per sincronizzare l'accesso alla lista dei client
        pthread_mutex_lock(&client_mutex); //facio lock sulla lista

        // Creazione di una nuova struttura ClientInfo per il client connesso
        if (clients == NULL) { //nel caso sia il primo client
            clients = (struct ClientInfo*)malloc(sizeof(struct ClientInfo));
            clients->socket = client_socket;
            memset(clients->name, 0, sizeof(clients->name));
            clients->next = NULL;
        } else { //nel caso ci siano già client
            struct ClientInfo* current = clients;
            while (current->next != NULL) { //quindi scorro la lista fino alla fine
                current = current->next;
            }
            current->next = (struct ClientInfo*)malloc(sizeof(struct ClientInfo));
            current->next->socket = client_socket;
            memset(current->next->name, 0, sizeof(current->next->name));
            current->next->next = NULL;
        }
        num_clients++; //incremento il numero di client (ne ho appena aggiunto uno)

        // Fine della sezione critica
        pthread_mutex_unlock(&client_mutex); //unlocco

        printf("Nuovo client connesso. Numero di client connessi: %d\n", num_clients);

        // Creazione di un thread per gestire il client
        pthread_t client_thread;
        if (pthread_create(&client_thread, NULL, handle_client, (void*)&client_socket) !=
0) {
            perror("Errore nella creazione del thread per il client");
            exit(EXIT_FAILURE);
        }
    }

    // Chiusura del socket del server
    close(server_socket);

    // Liberazione delle risorse allocate per i client
    struct ClientInfo* current = clients;
    while (current != NULL) {
        struct ClientInfo* next = current->next;
```

```
        close(current->socket);
        free(current);
        current = next;
    }

    return 0;
}

// Funzione di gestione del segnale di chiusura (Ctrl+C)
void handle_shutdown(int sig) {
    flag = 1;
    printf("\nServer terminato. Impossibile accettare nuove connessioni.\n");
}

// Funzione eseguita in ogni thread client
void* handle_client(void* arg) {
    int client_socket = *(int*)arg;
    char buffer[BUFFER_SIZE];
    ssize_t num_bytes;

    // Ricezione del nome del client
    memset(buffer, 0, sizeof(buffer));
    num_bytes = recv(client_socket, buffer, sizeof(buffer) - 1, 0);
    if (num_bytes <= 0) {
        close(client_socket);
        pthread_exit(NULL);
    }
    buffer[num_bytes] = '\0';

    // Mutex per sincronizzare l'accesso alla lista dei client
    pthread_mutex_lock(&client_mutex);
    struct ClientInfo* current = clients;
    while (current != NULL) {
        if (current->socket == client_socket) {
            strncpy(current->name, buffer, sizeof(current->name) - 1);
            break;
        }
        current = current->next;
    }
    pthread_mutex_unlock(&client_mutex);

    // Invio di un messaggio di benvenuto al client
    snprintf(buffer, sizeof(buffer), "Benvenuto, client %s!", current->name);
    if (send(client_socket, buffer, strlen(buffer), 0) < 0) {
        perror("Errore nell'invio del messaggio di benvenuto al client");
```

```
    pthread_exit(NULL);
}

// Loop per ricevere e inoltrare i messaggi tra i client
while (1) {
    memset(buffer, 0, sizeof(buffer));
    num_bytes = recv(client_socket, buffer, sizeof(buffer) - 1, 0);
    if (num_bytes <= 0) {
        break;
    } else {
        printf("Messaggio ricevuto dal client %s: %s\n", current->name, buffer);

        // Mutex per sincronizzare l'accesso alla lista dei client
        pthread_mutex_lock(&client_mutex);
        struct ClientInfo* sender = clients;
        while (sender != NULL) {
            if (sender->socket != 0 && sender->socket != client_socket) {
                char message[4096];
                snprintf(message, sizeof(message), "%s: %s", current->name, buffer);
                //stampo chi ha inviato il messaggio e il messaggio
                if (send(sender->socket, message, strlen(message), 0) < 0) {
                    perror("Errore nell'invio del messaggio al client");
                    break; //e poi con il while lo mando a tutti gli altri client che ho attivi
                }
            }
            sender = sender->next; //passo al prossimo client
        }
        pthread_mutex_unlock(&client_mutex);
    }
}

// Rimozione del client dalla lista e chiusura del socket
pthread_mutex_lock(&client_mutex);
struct ClientInfo* prev = NULL;
current = clients;
while (current != NULL) {
    if (current->socket == client_socket) {
        if (prev != NULL) {
            prev->next = current->next;
        } else {
            clients = current->next;
        }
        free(current);
        break;
    }
```

```
        prev = current;
        current = current->next;
    }
    num_clients--;
    pthread_mutex_unlock(&client_mutex);

    close(client_socket);
    pthread_exit(NULL);
}
```

Implementazione server remoto

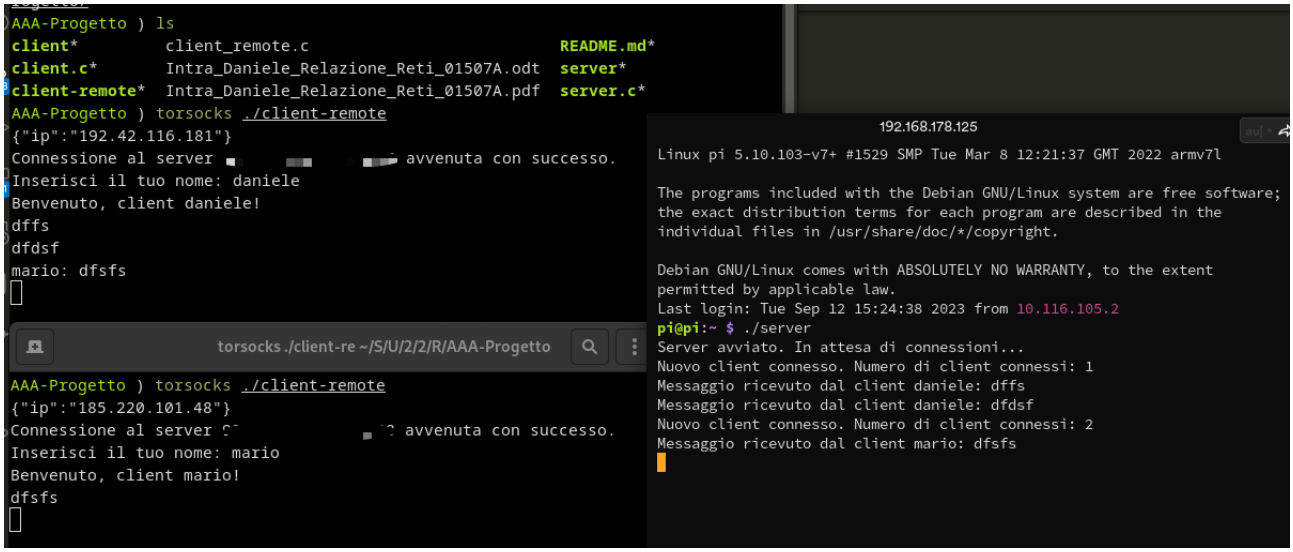
Il progetto in locale ha puramente uno scopo didattico in quanto non viene sfruttato bene il potenziale della comunicazione mediante rete Tor,
Ho deciso allora di procedere con un'implementazione "online"

1. la mia rete di casa è configurata in port forwarding per far passare le richieste al mio raspberry su cui risiede il programma del server.
2. Ho cambiato l'indirizzo nel codice del client mettendo quello pubblico di casa mia.
3. Ora il client con torsocks "bussa" alla 127.0.0.1:9050 per poi eseguire la richiesta nel codice al'ip pubblico di casa mia che risponderà sempre su rete tor e darà la possibilità al client di scambiare messaggi.

Il codice è cambiato nel seguente modo:

```
#define SERVER_IP "mio-ip-pubblico"
#define SERVER_PORT 8888
#define BUFFER_SIZE 1024
```

N.B. per comodità quando devo mostrare il funzionamento del progetto in un luogo diverso da casa mia, mi collego in vpn alla mia rete locale e svolgo tutto da lì, anche per evitare problemi con le richieste Tor (in università a quanto pare vengono bloccate)



Sviluppi futuri

1. **Miglioramento della Grafica**: Il client è attualmente basato su testo e ha un'interfaccia utente molto basilare. Si potrebbe creare un'interfaccia utente più amichevole e interattiva grazie all'utilizzo di librerie C (anche se non è il massimo)
2. **Implementazione del Protocollo SSL**: Per migliorare la sicurezza delle comunicazioni tra client e server, si potrebbe implementare il protocollo SSL/TLS per crittografare i dati trasmessi sulla rete. Questo proteggerà i messaggi dai tentativi di intercettazione.
3. **Sicurezza degli Accessi**: Implementare un sistema di autenticazione e autorizzazione dei client per garantire che solo gli utenti autorizzati possano accedere alla chat.
4. **Storico delle Conversazioni**: Consentire agli utenti di visualizzare il registro delle conversazioni precedenti.

Prove eseguite con altre tecnologie

Per la parte di Tor, dopo aver provato diversi metodi, ho scelto quello più semplice ed efficace: torsocks.
Prima di arrivare a questo tool ho provato direttamente ad eseguire (dal codice C client) una connessione tramite proxy al 127.0.0.1:9051 (proxy socks5 di tor).
Purtroppo richiedeva l'uso di un server remoto su cui reindirizzare la richiesta (avevo provato anche con un sito onion da me creato).
Un'altra idea è stata quella di utilizzare ProxyChain (catena di proxy). La principale differenza tra ProxyChains e Torsocks è che ProxyChains è più flessibile e adatto a una varietà di casi d'uso, mentre Torsocks è specificamente progettato per l'uso con la rete Tor e fornisce un elevato grado di anonimato, ma con meno flessibilità nelle configurazioni proxy.

Conclusioni

E' stato sviluppato un sistema di chat client-server in C usando multithreading e basandosi sulla rete Tor. Il server è in grado di gestire più client contemporaneamente, consentendo loro di comunicare tra loro tramite il server stesso. La chat supporta messaggi di testo e presenta una funzionalità di base per l'invio di messaggi di benvenuto ai nuovi client. Il client, d'altro canto, si connette al server, invia e riceve messaggi.

Ringrazio il prof. Ardagna e il dott. Berto per essere stati sempre disponibili a risolvere tutti i dubbi o problemi che si sono presentati durante la progettazione e realizzazione di tutto il codice.