UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE,
SCIENZE FISICHE E SCIENZE DELLA TERRA

Corso di Laurea Triennale in Informatica

Sistema Client-Server multithread con autenticazioni per lo streaming video multi-utente

Antonio
Santagati
Matricola 531795
Luigi
Villari
Matricola 532465

ANNO ACCADEMICO, 2023/2024

Indice

Relazione Progetto

Antonio Santagati e Luigi Villari

June 2024

1 Stato dell'arte

1.1 Cos'è uno Streaming video

Lo streaming video è una tecnologia che consente la trasmissione continua di contenuti video attraverso una rete, come Internet, senza dover scaricare interamente il file prima di poterlo visualizzare. Questa tecnologia permette agli utenti di guardare video in tempo reale, man mano che i dati vengono trasferiti dal server al dispositivo dell'utente. I tre pricipali step per lo streaming video sono:

- **Preparazione del contenuto**: I principali processi di preparazione del contenuto sono la segmentazione cioè la suddivisione del video in piccoli segmenti, spesso di pochi secondi, che facilitano la trasmissione e il buffering permettendo una riproduzione piu fluida.
- **Trasmissione del video** : La trasmissione del video avviene tramite protocolli come : HTTP (HyperText Transfer Protocol) o RTSP (Real-Time Streaming Protocol) oppure tramite l'implementazione di un server di streaming che è configurato per inviare i segmenti del video agli utenti su richiesta.
- **Ricezione e Riproduzione del Video**: La ricezione e la riproduzione del video avviene in diversi step:
  - Richiesta del Contenuto: Il dispositivo dell'utente invia una richiesta al server di streaming per il video desiderato.
  - Buffering: Il dispositivo inizia a ricevere i segmenti del video e li memorizza temporaneamente in un buffer. Questo aiuta a garantire che il video possa essere riprodotto senza interruzioni anche se c'è una temporanea instabilità nella connessione.
  - Riproduzione: Il video decodificato viene visualizzato in tempo reale sul dispositivo dell'utente. Man mano che il video viene riprodotto, nuovi segmenti continuano ad essere scaricati e decodificati.

1.2 Cos'è un'architettura client-server

L'architettura client-server è un modello di rete informatica in cui il carico di lavoro viene diviso tra i fornitori di risorse o servizi (server) e i richiedenti di queste risorse o servizi (client). Questo modello è ampiamente utilizzato in Internet e in molte applicazioni di rete per la sua efficienza, scalabilità e facilità di gestione. Le due componenti principali sono:

- **Client**: Il client è un dispositivo o un programma che richiede servizi o risorse da un server. E' in grado di inviare richieste al server, elaborare le risposte e presentare i dati all'utente.
- **Server**: Il server è un dispositivo o un programma che fornisce servizi o risorse ai client. Esso riceve e gestisce le richieste dei client, esegue le operazioni richieste e invia le risposte appropriate.

L'architettura client-server funziona attraverso una serie di step chiave: connessione, richiesta, elaborazione, risposta e visualizzazione. Un client stabilisce una connessione con il server, invia una richiesta, il server elabora la richiesta e invia una risposta che il client presenta all'utente. I vantaggi includono la centralizzazione delle risorse, la scalabilità, la manutenibilità e la sicurezza. Tuttavia, ci sono anche svantaggi come i costi di implementazione e manutenzione, la dipendenza dal server e la possibile congestione del server sotto un alto volume di richieste. In conclusione,l'architettura client-server è un modello fondamentale nella progettazione dei sistemi di rete, che consente una gestione centralizzata delle risorse e una distribuzione efficiente dei servizi ai client.

1.3 Cos'è il Multithreading

Un thread è la più piccola unità di elaborazione che può essere eseguita da un sistema operativo. Un thread è composto da un insieme di istruzioni che il processore può eseguire e può essere considerato come un "sotto-processo" all'interno di un processo più grande. Ogni processo può contenere uno o più thread, che condividono lo stesso spazio di indirizzamento, le risorse e i dati del processo. Il multithreading è una tecnica di programmazione che permette l'esecuzione simultanea di più thread all'interno di un singolo processo. Utilizzando il multithreading, un'applicazione può eseguire più operazioni in parallelo, migliorando le prestazioni e la reattività del sistema.

1.4 Cos'è VLC

VLC Media Player, comunemente noto come VLC, è un lettore multimediale open-source e cross-platform sviluppato dal progetto VideoLAN. È noto per la sua capacità di riprodurre un'ampia varietà di formati audio e video senza la necessità di installare codec aggiuntivi. VLC è estremamente versatile e supporta numerosi protocolli di streaming, rendendolo uno degli strumenti più popolari per la riproduzione di contenuti multimediali.

2 Descrizione Problema

Lo sviluppo di un sistema client-server multithread con autenticazione per lo streaming multi-utente rappresenta una sfida significativa nell'ambito delle applicazioni distribuite. Il progetto prevede la realizzazione di un'infrastruttura in grado di gestire contemporaneamente più utenti e offrendo uno streaming video in tempo reale.

2.1 Obiettivi progetto

Gli obiettivi principali del progetto sono:

- **Gestione Multi-Utente**: Il sistema deve essere capace di gestire simultaneamente più utenti, permettendo a ciascuno di accedere ai contenuti video senza interferenze.
- **Autenticazione Sicura**: Implementare meccanismi di autenticazione per assicurare che solo utenti autorizzati possano accedere ai contenuti video.
- **Streaming Video in Tempo Reale**: Fornire un servizio di streaming video fluido e in tempo reale, minimizzando latenza e buffering.
- **Architettura Multithread**: Utilizzare un'architettura multithread per migliorare le prestazioni del server, gestendo più richieste contemporaneamente senza degrado del servizio.
- **Utilizzo della Libreria VLC**: Sfruttare le funzionalità della libreria VLC per gestire lo streaming video.
- **Comunicazioni tra Socket**: Utilizzare socket per garantire una comunicazione affidabile tra client e server.

2.2 Sfide e problemi

- **Scalabilità**: Garantire che il sistema possa scalare efficacemente con l'aumento del numero di utenti richiede una gestione ottimale delle risorse del server e l'adozione di tecniche di load balancing.
- **Autenticazione**: Implementare un sistema di autenticazione sicuro e user-friendly è cruciale, inclusa la gestione delle credenziali degli utenti, la protezione dei dati sensibili e la prevenzione di accessi non autorizzati.
- **Sincronizzazione**: Assicurare che lo stato del server rimanga consistente anche quando più thread accedono e modificano dati simultaneamente richiede l'uso di tecniche di sincronizzazione per evitare race conditions e altri problemi di concorrenza.

2.3 Descrizione tecnica

- **Architettura Client-Server**: Il sistema sarà basato su un'architettura client-server, dove i client inviano richieste di accesso e streaming video al server, che risponde con i contenuti richiesti.
- **Multithreading**: Il server sarà implementato con un'architettura multithread, permettendo la gestione parallela delle richieste degli utenti. Ogni thread gestirà una connessione separata, migliorando l'efficienza e la reattività del sistema.
- **Libreria VLC**: La libreria VLC sarà utilizzata per gestire lo streaming video. VLC offre funzionalità avanzate di streaming e può essere integrato facilmente in applicazioni personalizzate per fornire una riproduzione video di alta qualità.
- **Comunicazioni tra Socket**:
  - **Connessione e Trasferimento**: Utilizzare socket per stabilire connessioni tra client e server. Il server ascolta su una porta specifica per le richieste in arrivo dai client. Una volta stabilita la connessione, i dati (inclusi i flussi video) vengono trasferiti attraverso il socket.
  - **Affidabilità e Controllo di Errore**: Le comunicazioni tra socket basate su TCP garantiscono l'affidabilità del trasferimento dei dati. TCP gestisce il controllo degli errori, l'ordinamento dei pacchetti e la ritrasmissione dei pacchetti persi, assicurando che i dati arrivino integri e nell'ordine corretto.
  - **Gestione della Concorrenza**: Ogni connessione socket viene gestita da un thread separato nel server multithread. Questo approccio permette di servire più client simultaneamente, migliorando la scalabilità e la reattività del sistema.

3 Implementazione

3.1 Server

Il Server è stato progettato per l'autenticazione dell'utente e il successivo streaming video. Il server utilizza socket per stabilire connessioni di rete con i client e il multithreading per gestire più connessioni contemporaneamente, garantendo un'elaborazione efficiente delle richieste. Le funzioni implementate per lo sviluppo del server sono:

- **verify**: Questa funzione si occupa di autenticare l'utente.
- **handle_client**: Si occupa di trasmettere al client il video in blocchi di dati.
- **manage_log**: gestisce il file di log e verifica se un utente è già connesso.

- **main**:Funzione principale del server. Si occupa di configurare il socket e metterlo in ascolto di connessioni in arrivo.

Le librerie importate all'interno del server sono:

- **stdio.h**: Questa è la libreria standard di input e output in C.
- **stdlib.h**: È la libreria standard di utilità generale che contiene funzioni per la gestione della memoria dinamica, controllo dei processi, conversioni e altro.
- **string.h**: Fornisce funzioni per la manipolazione delle stringhe, come la copia, la concatenazione, la comparazione e la ricerca di caratteri e sottostringhe.
- **sys/socket.h**: Contiene le definizioni dei socket e le relative funzioni per la comunicazione tra processi, sia locali che remoti, utilizzando i socket di rete.
- **arpa/inet.h**: Fornisce funzioni per la conversione di indirizzi IP da e verso rappresentazioni in formato stringa, così come altre utilità per la programmazione di rete.
- **unistd.h**: Include le definizioni delle chiamate di sistema standard come lettura, scrittura, chiusura di file, gestione dei processi, e altro, fornendo un'interfaccia per interagire con il kernel del sistema operativo.
- **pthread.h**: Fornisce funzioni per la gestione dei thread POSIX, permettendo la creazione, la sincronizzazione e la gestione di thread per la programmazione concorrente.

3.1.1 Verify

Nel contesto del programma server, la funzione `verify` svolge un ruolo importante nell'autenticazione degli utenti. Quando un client si connette al server, deve fornire un nome utente e una password. La funzione `verify` è incaricata di controllare queste credenziali confrontandole con un elenco di credenziali valide memorizzate in un file di testo. Se le credenziali corrispondono, l'utente viene autenticato e può procedere a interagire ulteriormente con il server; in caso contrario, l'accesso viene negato. La funzione `verify` prende tre argomenti:

- `username`: una stringa contenente il nome utente fornito dal client.
- `password`: una stringa contenente la password fornita dal client.
- `client_descriptor`: un intero che rappresenta il descrittore di file del socket del client, utilizzato per inviare messaggi di risposta al client.

La funzione legge il file riga per riga utilizzando `fgets`. Ogni riga contiene un nome utente e una password separati da uno spazio. Viene utilizzato `sscanf` per estrarre questi valori. Se il nome utente e la password estratti corrispondono a quelli forniti dal client (`username` e `password`), viene stampato un messaggio di autenticazione riuscita e inviato un messaggio al client in caso contrario viene inviato al client un messaggio di autenticazione fallita.

```
int verify(const char *username, const char *password, int
    client_descriptor) {
    char file_username[BUFFER_SIZE];
    char file_password[BUFFER_SIZE];
    char line[BUFFER_SIZE];
    FILE *check_user = fopen("check_user.txt", "r");

    if (check_user == NULL) {
        perror("Error opening file");
        return EXIT_FAILURE;
    }

    while (fgets(line, sizeof(line), check_user)) {
        if (sscanf(line, "%s %s", file_username,
            file_password) == 2) {
            if (strcmp(file_username, username) == 0 &&
                strcmp(file_password, password) == 0) {
                printf("Authenticated!\n");
                send(client_descriptor, "Authenticated\n",
                    strlen("Authenticated\n"), 0);
                fclose(check_user);
                return 0;
            }
        }
    }

    fclose(check_user);
    return EXIT_FAILURE;
}
```

3.1.2 Manage_log

La funzione `manage_log` gestisce il file di log per verificare se un utente specifico è già connesso. I parametri passati alla funzione sono: `username`, `password` e una flag (`add`) che viene utilizzata per aggiungere o rimuovere l'utente specificato, in base al valore della flag (1 per aggiungere, 0 per rimuovere). All'interno della funzione è presente una sezione critica, poiché bisogna gestire la concorrenza tra i thread che tentano di leggere o scrivere sul file di log. Questo viene fatto attraverso l'utilizzo dei **mutex** (un semaforo binario che può avere un valore tra 0 e 1), presente nella libreria `pthread`. La funzione esegue il lock del mutex e apre il file in modalità `r+` (lettura e scrittura). Successivamente, entra in un

ciclo dove vengono lette tutte le righe presenti nel file di log. Se `username` e `password`, passati come parametri, sono presenti nel file, viene inizializzata una variabile `user_exists` a 1, altrimenti a 0. Una volta eseguiti questi passaggi, seguono una serie di istruzioni condizionali (`if`) per verificare la flag `add` e la variabile `user_exists`. In base ai valori di queste variabili, si effettuerà la scrittura o la rimozione dell'utente specificato all'interno del file di log. Infine, viene eseguito l'unlock del mutex e la chiusura del file.

```
int manage_log(const char *username, const char *password,
    int add ){
    pthread_mutex_lock(&m);
    FILE *log_file = fopen("log.txt","r+");
    if (log_file == NULL){
        log_file = fopen("log.txt","w+");
        if(log_file == NULL){
            printf("error, could not open log file\n");
            pthread_mutex_unlock(&m);
            return -1;
        }
    }
    char line[BUFFER_SIZE];
    char file_username[BUFFER_SIZE];
    char file_password[BUFFER_SIZE];
    int user_exists = 0;
    while (fgets(line, sizeof(line), log_file)) {
        sscanf(line, "%s %s", file_username, file_password);
        if (strcmp(file_username, username) == 0 &&
            strcmp(file_password, password) == 0) {
            user_exists = 1;
            break;
        }
    }
     if (add && user_exists) {
        pthread_mutex_unlock(&m);
        return EXIT_FAILURE;  // User already logged in
    } else if (add && !user_exists) {
        fprintf(log_file, "%s %s\n", username, password);
    } else if (!add && user_exists) {
        FILE *temp_file = fopen("temp_log_file.txt", "w");
        fseek(log_file, 0, SEEK_SET); //posizionamento
            puntatore file
        while (fgets(line, sizeof(line), log_file)) {
            sscanf(line, "%s %s", file_username,
                file_password);
            if (!(strcmp(file_username, username) == 0 &&
                strcmp(file_password, password) == 0)) {
                fprintf(temp_file, "%s", line);
            }
        }
        fclose(log_file);
```

```
        fclose(temp_file);
        remove("log.txt");
        rename("temp_log_file.txt", "log.txt");
        pthread_mutex_unlock(&m);
        return 0;
    }
     fclose(log_file);
    pthread_mutex_unlock(&m);
    return 0;

}
```

3.1.3 Handle_client

La funzione `handle_client` gestisce la comunicazione con un singolo client all'interno del server. Quando un client si connette, questa funzione viene eseguita in un thread separato per gestire le richieste di quel client in modo asincrono rispetto agli altri. In linea di massima, la funzione inizia inviando al client la richiesta di username e password. Una volta ricevuta la risposta da parte del client viene richiamata la funzione `verify()` per verificare che l'utente abbia il permesso di accesso, se l'utente è verificato si procede con la scrittura di quest'ultimo sul file di log attraverso la funzione `manage_log` precedentemente spiegata.

```
printf("request username...\n");
    send(client_socket,"Username: ",strlen("Username: "),0);
    read(client_socket,buffer_verify,BUFFER_SIZE);
    char username[BUFFER_SIZE];
    strcpy(username,buffer_verify);
    memset(buffer_verify,0,BUFFER_SIZE);
    printf("Username recieved..\n");

    printf("request passowrd...\n");
    send(client_socket, "Password: ", strlen("Password: "),
        0);
    read(client_socket, buffer_verify, BUFFER_SIZE);
    char password[BUFFER_SIZE];
    strcpy(password, buffer_verify);
    memset(buffer_verify, 0, BUFFER_SIZE);
    printf("password recieved..\nAuthentication...\n");

    if(verify(username,password,client_socket)!=0){
        send(client_socket, "Authentication Failed\n",
            strlen("Authentication Failed\n"), 0);
        return NULL;
    }

    strcpy(client_info->username, username);
    strcpy(client_info->password, password);
```

```
    if (manage_log(client_info->username,
        client_info->password, 1) != 0) {
        send(client_socket, "User already logged\n",
            strlen("User already logged\n"), 0);
        close(client_socket);
        free(client_info);
        return NULL;
    }
```

Viene definita una struttura `client_info_t` che contiene il descrittore di socket del client, il nome del video scelto e i buffer per l'autenticazione e l'invio del video. All'interno della funzione, questa struttura viene utilizzata per accedere ai dati del client.

```
    client_info_t *client_info = (client_info_t *)arg;
    int client_socket = client_info->client_socket;
    char buffer[BUFFER_SIZE];
    char video[BUFFER_SIZE];
    char buffer_verify[BUFFER_SIZE];
```

Nel seguente codice viene creata una lista contenente tutti i titoli dei video disponibili sul server.Viene inviata al client dopo che quest'ultimo ha effettuato l'autenticazione. Appena il client riceve la lista ne sceglie uno,digitando il titolo sul prompt e inviandolo al server.Infine il server legge il titolo del video inviato dal client e si prepara per inviare i blocchi di byte.

```
    // Invio della lista dei video al client
    char *video_names[] = {"valenzia", "sium", "genas",
        "galaxy","movie"};
    char video_list[BUFFER_SIZE] = "";
    for (int i = 0; i < sizeof(video_names) / sizeof(char
        *); i++) {
        strcat(video_list, video_names[i]);
        strcat(video_list, "\n");
    }
    send(client_socket, video_list, strlen(video_list), 0);

    // Ricezione della scelta del video dal client
    memset(buffer, 0, BUFFER_SIZE);  // Pulizia del buffer
        prima della lettura
    if (read(client_socket, buffer, BUFFER_SIZE) <= 0) {
        perror("Error reading video choice from client");
        close(client_socket);
        free(client_info);
        manage_log(client_info->username,
            client_info->password, 0);
        return NULL;
    }
```

```
    snprintf(video, BUFFER_SIZE, "%s.mp4", buffer); //
        Formattazione sicura del nome del file
    printf("Selected video: %s\n", video);
    strcpy(client_info->video_name, video);
```

In questa parte di codice viene aperto il video_file, in caso di errore viene restituito un errore e chiusa la socket. Nel caso in cui il file video venisse aperto correttamente viene eseguito il ciclo while fino all'EOF, all'interno del ciclo vengono letti i byte attraverso la funzione **fread** e successivamente inoltrati al client attraverso la funzione **send**. Inoltre è presente un istruzione condizionale che verifica se sono presenti messaggi inviati dal client, questo viene fatto attraverso la funzione `recv` e la flag `MSG_DONTWAIT` per rendere la chiamata non bloccante. Una volta termianto l'invio del video,si effettua la chiusura del file e della socket.

```
// Apertura e invio del file video
    FILE *video_file = fopen(client_info->video_name, "rb");
    if (video_file == NULL) {
        perror("Error opening video file");
        close(client_socket);
        free(client_info);
        manage_log(client_info->username,
            client_info->password, 0);
        return NULL;
    }

   while (!feof(video_file)) {
    size_t bytes_read = fread(buffer, 1, sizeof(buffer),
        video_file);

    // Controlla se ci sono dati disponibili dal client
        senza bloccare
    char buffer_verify[BUFFER_SIZE];
    int recv_result = recv(client_socket, buffer_verify,
        sizeof(buffer_verify) - 1, MSG_DONTWAIT);
    if (recv_result > 0) {
        buffer_verify[recv_result] = '\0';
        printf("Received from client: %s\n", buffer_verify);
        if (strcmp(buffer_verify, "q") == 0) {
            printf("Client requested to close
                connection\n");
            fclose(video_file);
            close(client_socket);
            manage_log(client_info->username,
                client_info->password, 0);
            free(client_info);
            return NULL;
        }
    }else if (recv_result < 0 && errno != EWOULDBLOCK &&
        errno != EAGAIN) {
        perror("recv error");
```

```
            break;
        }

        if (bytes_read > 0) {
            send(client_socket, buffer, bytes_read, 0);
        }
    }
    // Pulizia finale nel caso di disconnessione o errore
        fclose(video_file);
        close(client_socket);
        manage_log(client_info->username,
            client_info->password, 0);
        free(client_info);
        return NULL;
    }
```

3.1.4 Main Server

La funzione main è la funzione principale del programma server, responsabile per l'avvio e il funzionamento del server stesso. Essa gestisce la creazione del socket del server, l'associazione dell'indirizzo e della porta, l'ascolto delle connessioni in entrata e la gestione delle richieste dei client. Nella prima parte è presente l'inizializzazione del server. Viene creato un socket utilizzando la funzione **socket()**, specificando il dominio degli indirizzi (AF_INET) e il tipo di socket (SOCK_STREAM) per la comunicazione TCP/IP. Se la creazione del socket fallisce, viene restituito un messaggio di errore e il programma termina.

```
server_socket = socket(AF_INET, SOCK_STREAM, 0);
if (server_socket == -1) {
    perror("Error creating socket");
    return EXIT_FAILURE;
}
```

Successivamente, vengono inizializzati l'indirizzo e la porta del server utilizzando la struttura `sockaddr_in`, e il socket viene associato a tale indirizzo utilizzando la funzione `bind()`. Se l'associazione fallisce, viene restituito un messaggio di errore e il programma termina.

```
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT);
server_addr.sin_addr.s_addr = inet_addr(IP_ADDRESS);

if (bind(server_socket, (struct sockaddr *)&server_addr,
    sizeof(server_addr)) < 0) {
    perror("Error binding socket");
    close(server_socket);
    return EXIT_FAILURE;
}
```

Il server si mette in ascolto di connessioni in entrata utilizzando la funzione `listen()`. Se l'ascolto fallisce, viene restituito un messaggio di errore e il programma termina.

```
if (listen(server_socket, N_THREADS) < 0) {
    perror("Error listening on socket");
    close(server_socket);
    return EXIT_FAILURE;
}
```

Infine, all'interno di un ciclo while infinito, il server accetta connessioni in entrata utilizzando la funzione `accept()`. Quando una connessione viene accettata, viene creato un thread per gestire la richiesta del client utilizzando la funzione `handle_client()`.

```
while (1) {
        client_socket = accept(server_socket, (struct
            sockaddr *)&client_addr, &client_addr_len);
        if (client_socket < 0) {
            perror("Error accepting client");
            continue;
        }
        client_info_t *client_info =
            malloc(sizeof(client_info_t));
        client_info->client_socket = client_socket;
        pthread_t thread;
        pthread_create(&thread, NULL, handle_client, (void
            *)client_info);
        pthread_detach(thread);
    }
```

Questo ciclo while continua ad ascoltare e gestire le connessioni dei client finché il server non viene chiuso manualmente o si verifica un errore irreversibile. Alla fine, vengono chiusi il socket del server e il programma termina.

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

3.2 Client

Il client è stato implementato per permettere all'utente di autenticarsi e segliere un video per la riproduzione. Inoltre permette all'utente di stoppare/riprendere l'esecuzione del video scelto o di interrompere la riproduzione. Il client si divide in due principali sezioni:

- **init socket()** : Funzione che si occupa di gestire la connessione, l'autenticazione e la scelta del video da parte dell'utente.
- **funzioni vlc**: che si occupano di istanziare vlc e permettere la riproduzione del video

Le librerie specifiche del client sono:

- **netinet/in.h**:Utilizzata in programmi che necessitano di comunicazioni di rete. Definisce strutture e costanti per l'internet protocol come sockaddr_in
- **vlc/vlc.h**:È la libreria di VLC media player per lo sviluppo di applicazioni che manipolano media.

3.2.1 Init_socket()

La funzione init_socket() è progettata per stabilire una connessione di rete tramite socket, autenticare un utente e infine permettere la scelta di un video da parte dell'utente.

Creazione del socket:
socket(AF_INET, SOCK_STREAM, 0) crea un nuovo socket TCP IP . **AF_INET** indica che si utilizza IPv4, mentre **SOCK_STREAM** specifica un tipo di socket orientato alla connessione (TCP), adatto per flussi di dati affidabili.

```
int init_socket() {
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    char buffer[BUFFER_SIZE] = {0};
    char authentication[BUFFER_SIZE];
    char username[BUFFER_SIZE] = {0};
    char password[BUFFER_SIZE] = {0};
    char video_choice[BUFFER_SIZE];
    if (sock < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }
```

Controllo della creazione del Socket:
Se la creazione del socket fallisce (cioè, sock è minore di 0), il programma stampa un errore

```
(perror(Socket creation failed)) e termina
   (exit(EXIT\_FAILURE)).
  if (sock < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }
```

Configurazione dell'Indirizzo del Server:
La struttura **sockaddr_in server_addres** viene inizializzata a zero e configurata con i dettagli del server: il tipo di famiglia di indirizzi **AF_INET**, la porta del server che utilizza **htons** per assicurarsi che la porta sia nel formato di rete appropriato e l'indirizzo IP del server **inet_addr**.

```
struct sockaddr_in server_address;
    memset(&server_address, 0, sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(SERVER_PORT);
    server_address.sin_addr.s_addr = inet_addr(SERVER_IP);

    if (connect(sock, (struct sockaddr *)&server_address,
        sizeof(server_address)) < 0) {
        perror("Connect failed");
        close(sock);
        exit(EXIT_FAILURE);
    }
```

Connessione al server:
Viene tentata una connessione al server specificato. In caso di fallimento, viene stampato un messaggio di errore, il socket viene chiuso, e il programma termina.

```
    if (connect(sock, (struct sockaddr *)&server_address,
        sizeof(server_address)) < 0) {
        perror("Connect failed");
        close(sock);
        exit(EXIT_FAILURE);
    }
```

Conclusione della Funzione:
Viene richiesto all'utente di inserire il nome utente che poi viene mandato al server, poi viene effettuato lo stesso procedimento per la password. Infine il server invia una risposta di autenticazione che viene letta e confrontata per accertare che l'autenticazione sia avvenuta con successo. Se l'autenticazione è riuscita, la funzione restituisce il valore del descriptor del socket sock, e l'utente sarà in grado di scegliere un video da visualizzare. La sceltà sarà inviata al server. Se l'autenticazione non riesce, il programma termina.

```
    read(sock, buffer, BUFFER_SIZE);
    printf("%s", buffer);
    fgets(username, BUFFER_SIZE, stdin);
    username[strcspn(username, "\n")] = '\0';
    send(sock, username, strlen(username), 0);

    memset(buffer, 0, BUFFER_SIZE);
    read(sock, buffer, BUFFER_SIZE);
    printf("%s", buffer);
    fgets(password, BUFFER_SIZE, stdin);
    password[strcspn(password, "\n")] = '\0';
    send(sock, password, strlen(password), 0);

    memset(buffer, 0, BUFFER_SIZE);
    read(sock, buffer, BUFFER_SIZE);
    strcpy(authentication, buffer);
    printf("%s", buffer);
    memset(buffer, 0, BUFFER_SIZE);
    if(strcmp (authentication, "Authenticated\n")==0){
            printf("Lista dei video disponibili:\n");
            read(sock, buffer, BUFFER_SIZE);
            printf("%s", buffer);

            printf("Choose a video to be displayed...\n");
            fgets(video_choice, BUFFER_SIZE, stdin);
            video_choice[strcspn(video_choice, "\n")] = 0;
            send(sock, video_choice, strlen(video_choice),
                0);

    //if(strcmp (authentication, "Authenticated\n")==0){
            return sock;
    }

    else{
        return -1;
    }
}
```

3.2.2 Funzioni VLC

Le funzioni vlc che troviamo all'interno del client sono:

- **open_media**:Inizializza un flusso di dati da un socket. Usata per configurare le risorse necessarie per iniziare a ricevere dati.
- **read_media**:Legge i dati dal socket configurato e li trasferisce in un buffer, tipicamente per processi di streaming o trasferimento dati.
- **seek_media e close_media**: Queste due funzioni sono state implementate perchè erano fondamentali come parametri all'interno delle funzioni di **callback** della libreria vlc. In esse è sufficiente la loro versione di default non sono state aggiornate o modificate ulteriormente.

```
static int open_media(void *opaque, void **datap, uint64_t
    *sizep) {
    int sock = *(int *)opaque;
    *datap = opaque;
    *sizep = 0;
    return 0;
}

static ssize_t read_media(void *opaque, unsigned char *buf,
    size_t len) {
    int sock = *(int *)opaque;
    ssize_t bytes_read = read(sock, buf, len);
    if (bytes_read < 0) {
        perror("Read failed");
        return 0;
    }
    return bytes_read;
}

static int seek_media(void *opaque, uint64_t offset) {
    return -1;
}

static void close_media(void *opaque) {
}
```

3.2.3 Main Client

Inizializzazione socket e di libVLC : Inizializza un socket.Se la creazione del socket fallisce , il programma termina restituendo 1. Crea un'istanza di libVLC con l'argomento **no xlib**, il quale indica di non utilizzare Xlib per l'output video, utile per sistemi senza un display grafico o per ridurre le dipendenze.

```
int sock = init_socket();
  if (sock < 0){
      return 1;
  }

  const char *vlc_args[] = {
          "--no-xlib"
      };
```

Controllo istanza vlc: Se **libvlc_new** non riesce a creare l'istanza, il programma stampa un errore e termina la connessione del socket e il programma stesso con uno stato di fallimento.
Creazione media player e media callback:

```
libvlc_instance_t *vlc_instance = libvlc_new(0, vlc_args);
  if (!vlc_instance) {
      fprintf(stderr, "libvlc initialization failed\n");
      close(sock);
      return EXIT_FAILURE;
  }

  libvlc_media_player_t *mp =
      libvlc_media_player_new(vlc_instance);
  if (!mp) {
      fprintf(stderr, "libvlc media player creation
          failed\n");
      libvlc_release(vlc_instance);
      close(sock);
      return EXIT_FAILURE;
  }

  libvlc_media_t *m = libvlc_media_new_callbacks(
      vlc_instance, open_media, read_media, seek_media,
          close_media, &sock
  );
  if (!m) {
      fprintf(stderr, "libvlc media creation failed\n");
      libvlc_media_player_release(mp);
      libvlc_release(vlc_instance);
      close(sock);
      return EXIT_FAILURE;
  }
```

Configurazione e Avvio del Media Player:

```
libvlc_media_player_set_media(mp, m);
libvlc_media_release(m);
libvlc_media_player_play(mp);
printf("Streaming video...\n");
```

Controllo interattivo e rilascio risorse: Il programma stampa istruzioni e rimane in attesa di input dall'utente, consentendo di mettere in pausa, in play o terminare il programma. Nel caso di terminazione della riproduzione video, attraverso il `send` viene notificato al server che il client vuole chiudere la connessione.

```
    printf("Streaming video...\nPress 'p' to pause/play,
        's' to stop, 'q' to quit.\n");

    char command;
    while ((command = getchar())) {
        switch (command) {
            case '\n':
                libvlc_media_player_pause(mp);
                break;
            case 'q':
                send(sock, &command, 1, 0);
                libvlc_media_player_stop(mp);
                libvlc_media_player_release(mp);
                libvlc_release(vlc_instance);
                close(sock);
                return 0;
            default:
                continue;
        }
    }
```

4 Risultati Sperimentali

Durante la fase di sviluppo del nostro sistema client-server multithread per lo streaming video con autenticazione, abbiamo condotto una serie di test rigorosi per valutare le prestazioni, la stabilità e la sicurezza della piattaforma. I dati sperimentali sono stati raccolti in diverse configurazioni di rete e carichi di lavoro, permettendo un'analisi dettagliata della risposta del sistema sotto vari scenari di uso.

Test di Scalabilità: Abbiamo simulato un numero crescente di richieste contemporanee di streaming da parte dei client per testare l'efficacia della nostra architettura multithread. I risultati hanno mostrato che il sistema è in grado di gestire efficacemente molteplici connessioni simultanee senza significativi degradi di performance, grazie alla gestione ottimizzata dei thread e all'allocazione dinamica delle risorse. Sono stati condotti test di carico massimo permettendoci

di stabilire quale fosse il numero massimo di video riproducibili da una singola postazione. Nel nostro caso specifico il test effettuato sui computer del laboratorio ci ha evidenziato un normale funzionamento fino al raggiungimento di 50 client simultanei senza riscontrare crash del dispositivo.
Test di Stabilità e Affidabilità: Abbiamo sottoposto il sistema a periodi prolungati di operatività per identificare eventuali perdite di memoria o comportamenti instabili. Il monitoraggio continuo per ore ha dimostrato una notevole stabilità, senza perdite di memoria o crash del sistema, anche sotto sforzo.
Analisi di Latenza e Throughput: La latenza è stata misurata dall'istante in cui il client invia la richiesta fino al momento in cui inizia la riproduzione del video. I risultati hanno indicato che la latenza media si mantiene sotto il secondo in condizioni normali di rete. Il throughput, misurato in termini di dati trasmessi per secondo dal server ai client, ha evidenziato la capacità del sistema di trasmettere dati ad alta velocità, compatibile con la trasmissione di video in alta definizione.

5 Conclusione e Sviluppi futuri

In conclusione, il progetto di sviluppo di un sistema client-server multithread per lo streaming video con autenticazione ha dimostrato come le moderne tecnologie di comunicazione e sicurezza possano essere integrate per offrire un servizio di streaming video affidabile e sicuro. Attraverso l'implementazione di un sistema multithread, abbiamo ottimizzato la gestione delle risorse del server, permettendo così di gestire contemporaneamente più richieste di client in modo efficace e senza perdite significative di prestazioni. L'integrazione di un sistema di autenticazione ha ulteriormente rafforzato la sicurezza del nostro sistema, assicurando che solo gli utenti autorizzati possano accedere ai contenuti video. Questo aspetto è particolarmente importante nell'era digitale attuale, dove la protezione dei contenuti e la privacy degli utenti sono di massima priorità. L'architettura client-server progettata ha mostrato una notevole stabilità e scalabilità durante la fase di testing, confermando la validità delle scelte progettuali e l'efficacia delle tecnologie adottate. Le sfide incontrate durante lo sviluppo, in particolare relative alla sincronizzazione dei thread e alla gestione delle sessioni utente sicure, sono state affrontate con soluzioni innovative che hanno arricchito la nostra competenza e preparazione professionale. In definitiva, il successo di questo progetto non solo dimostra la fattibilità tecnica e la sicurezza di un sistema di streaming video multithread, ma pone anche le basi per futuri sviluppi che potrebbero estendere ulteriormente le funzionalità e l'efficienza del sistema come ad esempio, l'aumento del numero di video presenti nel server, la divisione di quest'ultimi in categorie, l'implementazione di un interfaccia utente intuitiva e infine la possibilità di mettere in pausa il video per riprenderlo in un secondo momento. Siamo fiduciosi che le competenze acquisite e le tecnologie implementate ci permetteranno di affrontare con successo le sfide future nel campo dell'ingegneria del software e delle comunicazioni digitali.

Repository GitHub: Streming_video_VLC