Università degli Studi di Messina

Dipartimento MIFT

LABORATORIO DI RETI E SISTEMI DISTRIBUITI
PROGETTO DI FINE CORSO:
Streaming audio multi-utente con Python/VLC

Saccà Maurizio Matricola: 504056
Shatani Adel Matricola: 528496

MESSINA,
A.A. 2023/2024

Indice

Capitolo 1

Stato dell'Arte

1.1: Introduzione Generale al Contesto

Il progetto si propone di sviluppare un sistema di streaming audio multiutente utilizzando un'architettura client-server multithread in Python.
Lo streaming audio è una tecnologia che permette la trasmissione continua di contenuti audio tramite una rete, migliorando l'esperienza dell'utente in quanto l'audio non deve essere scaricato interamente prima di essere riprodotto. Utilizzando questa tecnologia, è possibile garantire una riproduzione costante e di alta qualità per ogni utente, indipendentemente dal file audio in uso.

1.2: Architettura Client-Server Multithread

Un'architettura client-server è un modello di rete dove i client (dispositivi o applicazioni) richiedono servizi o risorse da un server centrale. Il server risponde a queste richieste, fornendo dati o servizi come richiesto. Questo modello è fondamentale per lo streaming audio, poiché consente al server di gestire e distribuire il contenuto audio a più client contemporaneamente.

1.2.1 Funzionamento dell'Architettura Client-Server

Nel contesto del progetto, il server è responsabile della gestione degli stream audio e della distribuzione degli stessi ai client che ne fanno richiesta. Il server accetta connessioni dai client, che stabiliscono una sessione di comunicazione per inviare e ricevere dati. Ecco una panoramica di come funziona:

1. **Inizializzazione del Server**: Il server è configurato per ascoltare le richieste su una specifica porta di rete.

2. **Connessione dei Client**: I client si connettono al server inviando una richiesta di connessione.

3. **Gestione delle Connessioni**: Una volta stabilita la connessione, il server crea un nuovo thread per ogni client, permettendo la gestione parallela di più connessioni.

4. **Scambio di Dati**: Il server e i client possono ora scambiarsi dati. Nel caso dello streaming audio, il server invia i dati audio al client che li riproduce in tempo reale.

5. **Chiusura della Connessione**: Al termine dello streaming o su richiesta del client, la connessione viene chiusa e il thread associato viene terminato.

1.2.2 Multithreading in Python

La gestione di connessioni multiple è facilitata dal modulo `threading` di Python, che consente di eseguire operazioni in parallelo, creando thread separati per ogni connessione client. Questo approccio permette al server di gestire simultaneamente più richieste, migliorando l'efficienza del sistema e riducendo i tempi di risposta.

In alternativa al threading tradizionale, Python offre il modulo `asyncio`, che consente la gestione asincrona delle operazioni di I/O, utile per applicazioni con molte connessioni che non richiedono il parallellismo offerto dai thread.

1.3: Protocolli di Trasporto: TCP vs UDP

Per lo streaming audio, la scelta del protocollo di trasporto è cruciale. I due protocolli principali sono TCP (Transmission Control Protocol) e UDP (User Datagram Protocol):

1.3.1 Transmission Control Protocol (TCP)

TCP è un protocollo orientato alla connessione che garantisce la consegna affidabile dei pacchetti di dati tra il client e il server. Ogni pacchetto inviato deve essere confermato dal destinatario, e i pacchetti persi vengono ritrasmessi. Questo garantisce che i dati siano ricevuti correttamente e nell'ordine giusto.

- **Affidabilità**: TCP garantisce che i dati inviati arriveranno al destinatario in modo completo e nell'ordine corretto.
- **Controllo di Congestione**: TCP include meccanismi per adattare la velocità di trasmissione in base alla congestione della rete.
- **Gestione degli Errori**: TCP verifica l'integrità dei dati ricevuti e richiede ritrasmissioni in caso di errori.

Queste caratteristiche rendono TCP ideale per applicazioni dove la precisione dei dati è più importante della velocità, come nel caso dello streaming audio dove la perdita di dati potrebbe degradare significativamente l'esperienza dell'utente.

1.3.2 User Datagram Protocol (UDP)

UDP, al contrario di TCP, è un protocollo senza connessione che non garantisce la consegna affidabile dei pacchetti. I pacchetti sono inviati senza controllo di errore o conferma, il che può portare a perdite o disordine dei dati.

- **Velocità**: UDP è più veloce di TCP poiché non richiede la conferma dei pacchetti inviati.
- **Semplicità**: UDP è più semplice e meno sovraccarico rispetto a TCP.

Tuttavia, la mancanza di affidabilità e controllo rende UDP meno adatto per applicazioni dove l'integrità dei dati è critica, come nello streaming audio di alta qualità.

1.4: VLC per lo Streaming Audio

VLC è un famoso lettore multimediale che supporta una vasta gamma di formati audio e video. Nel contesto del progetto, VLC viene utilizzato per lo streaming audio perché supporta vari formati e protocolli, consentendo una trasmissione audio affidabile e di alta qualità.

Per l'integrazione con Python, viene utilizzata la libreria `python-vlc`, che permette di controllare le funzionalità di VLC direttamente da Python, facilitando l'implementazione di un sistema di streaming audio. Questa libreria offre un'interfaccia semplice per gestire la riproduzione e la trasmissione di contenuti audio nel progetto.

1.5: Autenticazione nello Streaming Audio

Per garantire che solo gli utenti autorizzati possano accedere al sistema di streaming, il progetto implementa un sistema di autenticazione di base. Questo sistema consiste nella verifica delle credenziali utente (nome utente e password) per controllare l'accesso al sistema.

- **Autenticazione Base**: Verifica delle credenziali fornite dall'utente per garantire l'accesso.
- **Gestione delle Sessioni**: Utilizzo di sessioni per mantenere l'autenticazione durante lo streaming.

Questo metodo assicura che solo gli utenti autorizzati possano connettersi al server e accedere agli stream audio.

1.6: Gestione di Streaming Audio Multiutente

La gestione di più stream audio simultanei presenta diverse sfide tecniche:

- **Sincronizzazione degli Stream**: Assicurare che gli utenti possano ricevere flussi audio in sincronia, evitando ritardi significativi.
- **Ottimizzazione delle Risorse**: Gestione efficiente delle risorse del server per supportare connessioni multiple senza degradazione della qualità dello streaming.

Questi aspetti sono cruciali per fornire un'esperienza di streaming fluida e di alta qualità agli utenti.

Capitolo 2

Descrizione del Problema

2.1: Introduzione

Lo sviluppo di un sistema di streaming audio multiutente con autenticazione presenta diverse sfide tecniche e operative. Questo capitolo discute le principali problematiche affrontate durante il progetto e definisce gli obiettivi chiave.

2.2: Problematiche con lo Streaming Audio

Lo streaming audio deve affrontare numerose problematiche per garantire una trasmissione fluida e di alta qualità. Alcuni dei problemi principali includono:

- **Latenza**: Il ritardo nella trasmissione dei dati può influenzare l'esperienza dell'utente. È essenziale minimizzare la latenza per garantire un flusso audio continuo.
- **Qualità del Servizio**: Mantenere un'elevata qualità del servizio è fondamentale per prevenire interruzioni o degradazione del flusso audio.
- **Gestione della Larghezza di Banda**: La disponibilità e l'uso efficiente della larghezza di banda sono critici per evitare buffer vuoti e interruzioni.

2.3: Creazione della Connessione e Problemi di Sicurezza

La creazione e la gestione delle connessioni tra client e server sono complesse, specialmente in un ambiente multiutente:

- **Stabilire Connessioni Affidabili**: Garantire che le connessioni tra client e server siano affidabili e stabili nel tempo.
- **Sicurezza e Autenticazione**: Proteggere le connessioni da accessi non autorizzati e garantire che solo gli utenti legittimi possano accedere al sistema.
- **Gestione delle Credenziali**: Memorizzare e verificare in modo sicuro le credenziali degli utenti.

Per la gestione delle credenziali e la prevenzione dell'accesso simultaneo ai dati critici, verrà esplorato l'uso dei mutex nel capitolo successivo.

2.4: Protocolli di Trasporto: TCP vs UDP

La scelta del protocollo di trasporto ha un impatto significativo sul funzionamento del sistema di streaming audio:

2.4.1 Scelta di TCP

Abbiamo scelto di utilizzare TCP rispetto a UDP per i seguenti motivi:

- **Affidabilità**: TCP garantisce la consegna dei pacchetti nell'ordine corretto, evitando perdite di dati che possono degradare l'esperienza audio.
- **Controllo del Flusso e di Congestione**: TCP adatta la velocità di trasmissione in base alla congestione della rete, prevenendo sovraccarichi e garantendo una trasmissione fluida.
- **Integrità dei Dati**: TCP verifica l'integrità dei dati trasmessi, riducendo gli errori e garantendo che i dati audio siano trasmessi senza corruzioni.

Sebbene UDP possa offrire velocità superiori e minori latenze, la sua mancanza di affidabilità e controllo lo rende meno adatto per applicazioni di streaming audio dove la qualità dei dati è critica.

2.5: Obiettivi del Progetto

Gli obiettivi principali del progetto sono:

1. **Implementare un Sistema di Streaming Audio Multiutente**: Consentire a più utenti di accedere simultaneamente a flussi audio di alta qualità.

2. **Garantire l'Autenticazione degli Utenti**: Implementare un sistema di autenticazione sicuro per proteggere l'accesso ai flussi audio.

3. **Ottimizzare l'Utilizzo delle Risorse del Server**: Gestire efficientemente le risorse per supportare connessioni multiple senza compromettere la qualità dello streaming.

4. **Minimizzare la Latenza e Garantire la Sincronizzazione**: Assicurare che i flussi audio siano riprodotti con una latenza minima e che siano sincronizzati tra i diversi utenti.

Questi obiettivi guidano lo sviluppo del progetto e definiscono i criteri di successo per il sistema di streaming audio proposto.

Capitolo 3

Implementazione

Questo capitolo descrive in dettaglio l'implementazione del sistema di streaming audio multiutente. Verranno esaminati i file sorgente principali e le loro funzioni critiche, con particolare attenzione alla gestione dell'autenticazione, dei file audio, e degli utenti.

3.1: Librerie Utilizzate

Il sistema di streaming audio multiutente utilizza varie librerie Python per implementare le sue funzionalità principali:

- `socket`: Permette la comunicazione tra server e client utilizzando il protocollo TCP/IP.
- `threading`: Consente la gestione simultanea di più connessioni client tramite l'uso di thread.
- `os`: Fornisce funzioni per l'interazione con il sistema operativo, come la gestione dei file.
- `subprocess`: Utilizzata nel client per avviare e gestire il lettore VLC, riproducendo lo stream audio ricevuto.
- `time`: Gestisce i ritardi temporali per sincronizzare la comunicazione tra server e client.
- `authuser` e `fileHandler`: Moduli creati da noiper l'autenticazione degli utenti e la gestione dei file.

3.2: File: `authuser.py`

Il file `authuser.py` gestisce l'autenticazione degli utenti nel sistema. Esso contiene le funzioni principali per aggiungere nuovi utenti e autenticare gli utenti esistenti utilizzando un file di testo (`users.txt`) per la memorizzazione delle credenziali.

3.2.1 Funzione: `add_user`

La funzione `add_user` aggiunge un nuovo utente al sistema. Essa verifica che il nome utente non esista già nel file `users.txt` e, se non presente, aggiunge il nuovo utente con la sua password.

```
def add_user(username, password):
    if not isinstance(username, str):
        return False

    with open('users.txt', 'a+') as file:
        file.seek(0)
        for line in file:
            parts = line.strip().split(':')
            if len(parts) != 2:
                continue
            stored_username, _ = parts
            if stored_username == username:
                return False
        file.write(f"{username}:{password}\n")
        return True
```

Restituisce `True` se l'utente è stato aggiunto con successo, altrimenti `False`.

3.2.2 Funzione: `authenticate_user`

La funzione `authenticate_user` verifica le credenziali di un utente cercando una corrispondenza esatta tra il nome utente e la password nel file `users.txt`.

```
def authenticate_user(username, password):
    with open('users.txt', 'r') as file:
        for line in file:
```

```
            stored_username, stored_password = line.strip().split(':')
            if stored_username == username and stored_password == password:
                return True
    return False
```

Restituisce `True` se trova una corrispondenza esatta, altrimenti `False`.

3.3: File: `fileHandler.py`

Il file `fileHandler.py` gestisce le operazioni relative ai file nel sistema, in particolare l'elenco dei file disponibili per lo streaming.

3.3.1 Funzione: `listFiles`

La funzione `listFiles` elenca tutti i file presenti in una directory presa come parametro d'ingresso e restituisce gli elementi presenti in questa directory.

```
import os

def listFiles(dir):
    files = []
    contents = os.listdir(dir)

    for _ in contents:
        files.append(_)

    return files
```

3.4: Directory: `files.py`

La directory `files.py` contiene i file che gli utenti possono riprodurre in streaming.

3.5: File: `users.txt`

Il file `users.txt` contiene le informazioni sugli utenti e le loro credenziali. Ogni linea del file segue il formato `username:password`.

```
# Esempio di contenuto di users.txt
Adel:1234
mauri:fuffi
```

Le funzioni add_user() e authenticate_user fanno riferimento a questo file.

3.6: File: server.py

Il file `server.py` implementa il lato server del sistema di streaming audio. Questo server gestisce le connessioni dei client, autentica gli utenti e avvia lo streaming dei file audio disponibili.

3.6.1 Configurazione del Server

Il server viene configurato per ascoltare le connessioni su un indirizzo IP e una porta specifici. Utilizza il modulo `socket` per la comunicazione di rete e `threading` per gestire più connessioni contemporaneamente.

```
import socket
from threading import Thread, Lock
import time
import authuser
import fileHandler
import os

BUFFERSIZE = 4096
MUTEX = Lock()

SERVER_HOST = '0.0.0.0'
SERVER_PORT = 9999

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind((SERVER_HOST, SERVER_PORT))
server.listen()
```

```
print(f"Server in ascolto all'indirizzo: {SERVER_HOST}:{SERVER_PORT}")
```

- **Socket di Rete**: Il server crea un socket TCP (`socket.SOCK_STREAM`), lo configura per riutilizzare gli indirizzi (`socket.SO_REUSEADDR`) e lo associa all'indirizzo IP e alla porta dichiarati precedentemente.

3.6.2 Gestione delle Connessioni dei Client

La funzione `handleClient` gestisce le interazioni con il client una volta che si collega, autenticandolo e indirizzandolo al menù.

```
def handleClient(clientSock, clientAddr):
    print(f"[INFO] Connessione accettata da: {clientAddr}")
    try:
        send_message(clientSock, "Inserire modalità:\nRegistrazione\nAccedi\nEsci")
        time.sleep(0.1)
        send_message(clientSock, "requiredInput")
        mode = clientSock.recv(BUFFERSIZE).strip().decode('utf-8')

        if mode.lower() not in ['registrazione', 'accedi', 'esci']:
            send_message(clientSock, "Operazione non valida. Verrai disconnesso.")
            clientSock.close()
            return

        if mode.lower() == 'esci':
            send_message(clientSock, "Arrivederci!")
            clientSock.close()
            return

        if mode:
            send_message(clientSock, "Username:")
            time.sleep(0.1)
            send_message(clientSock, "requiredInput")
            username = clientSock.recv(BUFFERSIZE).strip().decode('utf-8')

            send_message(clientSock, "Password:")
            time.sleep(0.1)
```

```
            send_message(clientSock, "requiredInput")
            password = clientSock.recv(BUFFERSIZE).strip().decode('utf-8')

        with MUTEX:
            if mode.lower() == 'registrazione':
                if not authuser.add_user(username, password):
                    send_message(clientSock, "Errore: registrazione fallita, Verrai
                    clientSock.close()
                    return
                send_message(clientSock, "Registrazione effettuata con successo!")
            elif mode.lower() == 'accedi':
                if not authuser.authenticate_user(username, password):
                    send_message(clientSock, "Errore: autenticazione fallita, Verra
                    clientSock.close()
                    return
                send_message(clientSock, "Accesso effettuato con successo!")

        menu(username, clientSock, clientAddr)

    except Exception as e:
        print(f"[ERROR] Errore durante la gestione del client {clientAddr}: {e}")
    finally:
        print(f"[INFO] Connessione chiusa con: {clientAddr}")
```

- **Ricezione della modalità**: Il server chiede al client di specificare la modalità (`Registrazione`, `Accedi`, `Esci`) e utilizza le funzioni di `authuser` per gestire la registrazione e l'autenticazione degli utenti.

- **Mutex**: Utilizza un lock (`MUTEX`) per garantire che le operazioni su file siano thread-safe. L'utilizzo con `with` garantisce che il lock venga sbloccato in **automatico** all'uscita dal codice

- **Gestione degli Errori**: Gestisce eventuali errori e chiude la connessione in modo sicuro.

3.6.3 Menù di Selezione

La funzione `menu` permette agli utenti autenticati di scegliere tra lo streaming audio e la disconnessione.

```
def menu(username, clientSock, clientAddr):
    send_message(clientSock, f"Benvenuto {username}, inserisci l'operazione che vuo
    time.sleep(0.1)
    send_message(clientSock, "requiredInput")
    while True:
        try:
            mode = clientSock.recv(BUFFERSIZE).strip().decode('utf-8')

            if mode == '1':
                streaming(username, clientSock, clientAddr)
            elif mode == '2':
                send_message(clientSock, "Connessione chiusa.")
                break
            else:
                send_message(clientSock, "Operazione non valida. Riprova.")
        except Exception as e:
            print(f"[ERROR] Errore durante il menu per il client {clientAddr}: {e}"
            break
    clientSock.close()
```

- **Scelta dell'Operazione**: Il client può scegliere tra avviare lo streaming audio o disconnettersi.
- **Gestione dello Streaming**: Se l'utente sceglie di avviare lo streaming, la funzione `streaming` viene chiamata.

3.6.4 Streaming dei File Audio

La funzione `streaming` è il cuore del progetto, gestisce la visualizzazione, selezione e trasmissione dei file audio disponibili allo streaming.

```
def streaming(username, clientSock, clientAddr):
    files = fileHandler.listFiles('./files')
    files_list_str = '\n'.join(f"ID: {i} \tAudio: {file_name}" for i, file_name in
```

```
send_message(clientSock, f"Scegli l'ID dell'audio da riprodurre\n{files_list_st
time.sleep(0.1)
send_message(clientSock, "requiredInput")
chosenAudioId = clientSock.recv(BUFFERSIZE).strip().decode('utf-8')
send_message(clientSock, f"Riproduco il file {chosenAudioId}")

try:
    chosen_audio_id = int(chosenAudioId)
    if 0 <= chosen_audio_id < len(files):
        chosen_audio = files[chosen_audio_id]
        file_path = os.path.abspath(f'./files/{chosen_audio}')

        print(f"[INFO] L'utente {username} ha scelto di riprodurre: {chosen_aud

        with open(file_path, 'rb') as f:
            while True:
                data = f.read(BUFFERSIZE)
                if not data:
                    break
                clientSock.sendall(data)
                time.sleep(0.1)

        send_message(clientSock, "Streaming terminato.")
    else:
        send_message(clientSock, "ID non valido. Connessione chiusa.")
except ValueError:
    send_message(clientSock, "Input non valido. Connessione chiusa.")
except Exception as e:
    print(f"[ERROR] Errore durante lo streaming per il client {clientAddr}: {e}
    send_message(clientSock, "Errore durante lo streaming. Connessione chiusa."

clientSock.close()
```

- **Elenco dei File**: Utilizza la funzione `listFiles` di `fileHandler` per ottenere la lista dei file disponibili.
- **Trasmissione dei Dati**: Legge il file audio in blocchi e li invia al client.

- **Gestione degli Errori**: Gestisce input non validi e errori durante lo streaming.

3.6.5 Main - Accettazione delle Connessioni

Il server accetta le connessioni dei client e avvia un thread separato per gestire ciascuna connessione.

```
while True:
    client_socket, client_address = server.accept()
    client_thread = Thread(target=handleClient, args=(client_socket, client_address
    client_thread.start()
```

Con il `while True` il server si pone in ascolto per nuove connessioni *sempre*, subito dopo aver creato un thread dedicato al client che si vuole connettere.

3.7: File: `client.py`

Il file `client.py` gestisce il lato client del sistema di streaming audio, permettendo agli utenti di connettersi al server, autenticarsi e riprodurre lo stream audio selezionato.

3.7.1 Funzione: `play_stream`

La funzione `play_stream` si occupa della riproduzione dello stream audio ricevuto dal server. Utilizza `subprocess` per avviare VLC in modalità di lettura da pipe, permettendo di inviare i dati audio ricevuti direttamente al lettore.

```
import socket
import subprocess

def play_stream(sock):
    process = subprocess.Popen(
        ['vlc', '--file-caching=3000', '-'],
        stdin=subprocess.PIPE,
```

```
        stdout=subprocess.DEVNULL,
        stderr=subprocess.DEVNULL
    )

    try:
        while True:
            data = sock.recv(BUFFERSIZE)
            if not data:
                break
            process.stdin.write(data)
    except Exception as e:
        print(f"[ERROR] Errore durante lo streaming: {e}")
    finally:
        process.stdin.close()
        process.wait()
```

`play_stream` riceve i dati audio dal server in piccoli blocchi (`BUFFERSIZE`) e li invia al processo VLC tramite la pipe standard di ingresso (`stdin`). La funzione gestisce anche eventuali errori durante lo streaming e chiude il processo VLC al termine della trasmissione.

3.7.2 Funzione: `connection`

La funzione `connection` gestisce la connessione al server e l'interazione con esso. Si occupa di inviare le credenziali, ricevere risposte e richiedere l'input dall'utente, utilizzando un ciclo per mantenere attiva la comunicazione fino alla disconnessione.

```
def connection():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((SERVER_HOST, SERVER_PORT))

        while True:
            response = sock.recv(BUFFERSIZE).decode(errors='ignore').strip()

            if "Verrai disconnesso." in response or "Arrivederci!" in response or "
                print(f"[SERVER] {response}")
                print("Disconnessione dal server.")
```

```
            break

        elif 'Riproduco il file' in response:
            print(f"[SERVER] {response}")
            play_stream(sock)
            break

        elif 'requiredInput' in response:
            message = input("Tu: ")
            sock.sendall(message.encode())
        else:
            print(f"[SERVER] {response}")
```

All'interno della funzione, il client riceve e gestisce le risposte del server. In base al contenuto della risposta, il client può:

- Terminare la connessione.
- Avviare lo streaming audio chiamando `play_stream`.
- Richiedere l'input dall'utente e inviarlo al server.
- Visualizzare messaggi informativi ricevuti dal server.

Capitolo 4

Risultati Sperimentali

4.1: Introduzione

In questa sezione verranno presentati i risultati ottenuti dai test di carico eseguiti sul sistema di streaming audio multiutente.
I test hanno l'obiettivo di valutare le prestazioni del sistema in termini di utilizzo delle risorse sotto **diverse condizioni di carico**.

4.2: Metodologia dei Test

Per analizzare le prestazioni del sistema, sono stati effettuati diversi test simulando un numero variabile di utenti connessi simultaneamente.
Ogni test è stato eseguito con un determinato numero di client che richiedono lo streaming audio, misurando l'utilizzo della CPU e della memoria RAM.

I test sono stati condotti utilizzando lo strumento di monitoraggio `htop`.
I risultati sono stati raccolti e analizzati per ogni configurazione di carico.

4.3: Risultati dei Test

Di seguito verranno presentati i risultati ottenuti dai test di carico. Le immagini e i grafici rappresentano le metriche chiave misurate durante i test.

4.3.1 Utilizzo della CPU e della Memoria

Figura 4.1: Utilizzo della CPU con 5 client connessi

Figura 4.2: Utilizzo della CPU con 10 client connessi

Figura 4.3: Utilizzo della CPU con 60 client connessi

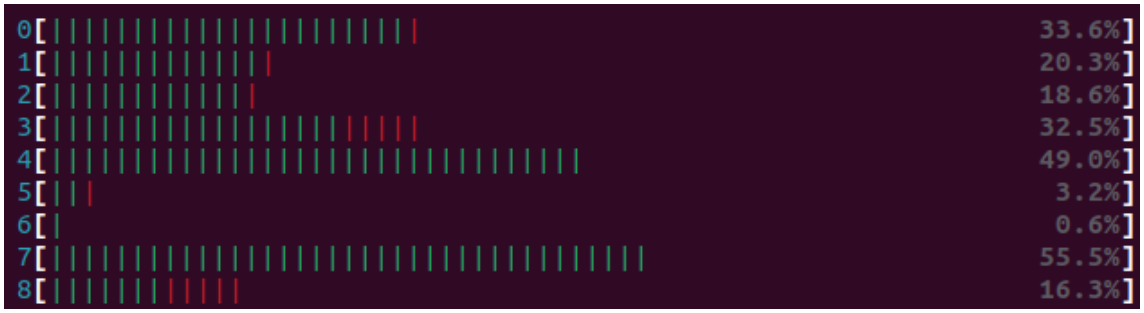

Figura 4.4: Utilizzo della CPU con 240 client connessi

4.3.2 Qualità del Servizio

La qualità del servizio è rimasta invariata, sotto un punto di vista auditivo,nonostante la elevata quantità di client connessi simultaneamente.
Tuttavia, a grande quantità di richieste ha aumentato, anche se non eccessivamente, la latenza nel gestire le richieste.

4.4: Discussione dei Risultati

La visualizzazione del carico fornita da htop ci mostra come:

- Per 5 client, il carico distribuito sui core del computer non supera il 15%.
- Per 10 client, la percentuale, sebbene di poco, aumenta del 5%
- Con 60 client si nota un incremento non indifferente, passando infatti dal 20% circa al 50%
- A 240 client, il server ha un carico distribuito del 170%

È importante notare che *htop* fornisce un'analisi indiscriminata, senza tenere conto di quali processi stiano aveno impatto sulla CPU; per cui, seppur poco in quanto erano attivi solo i processi stantard di Ubuntu, è presente un po' di **rumore** in questi risultati.
Nonostante il rumore, si può notare come i core siano sottoposti a uno stress maggiore quando la quantità dei client aumenta. È comunque stato possibile arrivare fino a **250 client** circa senza avere interruzioni o degradazioni sulla qualità dello streaming.

Capitolo 5

Conclusioni e Sviluppi Futuri

5.1: Conclusioni

I risultati dei test mostrano che il sistema di streaming audio multiutente sviluppato è in grado di gestire un numero significativo di connessioni simultanee mantenendo una latenza accettabile e un utilizzo delle risorse del server sotto controllo. Tuttavia, all'aumentare del numero di utenti, è stato osservato un incremento della latenza e un maggiore consumo di risorse, evidenziando la necessità di ottimizzazioni ulteriori per garantire una qualità del servizio costante.

In conclusione, il progetto ha raggiunto gli obiettivi prefissati di implementare un sistema di streaming audio multiutente con autenticazione, ottimizzando l'utilizzo delle risorse del server e minimizzando la latenza.
I test di carico hanno dimostrato la robustezza del sistema, pur suggerendo che ulteriori miglioramenti potrebbero essere implementati per gestire scenari di utilizzo ancora più intensivi.

Future espansioni del progetto potrebbero includere:

- Implementazione della crittografia per l'autenticazione degli utenti;
- Adozione di tecnologie di bilanciamento del carico;
- Distribuzione dei server;
- Sviluppo di un'interfaccia utente più intuitiva e user-friendly;
- Aggiunta di funzionalità per il controllo dello streaming audio;