Sistema Multitheading per la prenotazione di ticket

La Fauci Nicolo'
Matricola 529202

Relazione Progetto
Laboratorio di Reti
e Sistemi Distribuiti

UNIVERSITÀ DEGLI STUDI DI MESSINA
DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE,
SCIENZE FISICHE E SCIENZE DELLA TERRA
Corso di Laurea Triennale in Informatica

Indice

1 Stato dell'arte

1.1 Cos'è un sistema di ticketing

Il sistema di ticketing e prenotazione per eventi è una componente cruciale per molte organizzazioni, che spaziano dai teatri alle compagnie aeree. La gestione efficiente di questi sistemi è fondamentale per garantire una buona esperienza utente e massimizzare l'efficienza operativa. L'uso di un'architettura client/server multithread, abbinata a un database NoSQL come MongoDB, offre una soluzione scalabile e reattiva per gestire tali sistemi. Molti sistemi di ticketing e prenotazione esistono già, alcuni dei quali utilizzano tecnologie simili:

- **Ticketmaster:** Utilizza un'architettura client/server scalabile per gestire milioni di richieste al giorno.
- **Eventbrite:** Offre una piattaforma di gestione eventi che utilizza un'infrastruttura distribuita per garantire alta disponibilità e scalabilità.
- **Airline Reservation Systems:** Sistemi come Amadeus e Sabre utilizzano architetture client/server e database distribuiti per gestire milioni di prenotazioni in tempo reale.

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. È 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

Il multithreading è una tecnica di programmazione che permette l'esecuzione simultanea di più thread all'interno di un singolo processo. Un thread è la più piccola unità di elaborazione che può essere gestita in modo indipendente da un sistema operativo. Un processo può contenere uno o più thread che condividono le stesse risorse, come la memoria, ma possono essere eseguiti indipendentemente l'uno dall'altro.

- **Vantaggi:** Uno dei principali vantaggi del multithreading è il miglioramento dell'utilizzo della CPU, un altro beneficio importante è la maggiore reattività delle applicazioni.Inoltre, il multithreading permette una più facile condivisione delle risorse. Infine, il multithreading semplifica la programmazione di applicazioni che richiedono esecuzione simultanea di più compiti, come i server web che devono gestire più richieste di client contemporaneamente.

- **Svantaggi:** Nonostante i suoi vantaggi, il multithreading presenta anche diverse sfide. La complessità della programmazione è uno dei principali svantaggi, un altro problema comune è il deadlock, una situazione in cui due o più thread si bloccano aspettando risorse occupate da altri thread. Inoltre, il cambio di contesto tra thread può introdurre un overhead, riducendo i benefici della concorrenza in alcuni casi.

- **Utilizzo:** Il multithreading trova applicazione in molti ambiti delle moderne applicazioni informatiche. Nei sistemi operativi, il multithreading è utilizzato per gestire operazioni simultanee come input/output. Nelle applicazioni di rete, il multithreading è essenziale per gestire più connessioni client contemporaneamente.Infine, le interfacce utente grafiche utilizzano il multithreading per mantenere l'interfaccia reattiva mentre eseguono compiti in background.

1.4 MongoDB

MongoDB è un database NoSQL che memorizza i dati in documenti BSON (Binary JSON), rendendolo altamente flessibile e scalabile. È particolarmente adatto per applicazioni che richiedono una gestione efficiente di grandi volumi di dati non strutturati.

- **Vantaggi di MongoDb:** MongoDB supporta schemi dinamici. A differenza dei tradizionali database relazionali, dove ogni modifica alla struttura dei dati richiede un intervento complesso, MongoDB consente di aggiungere, rimuovere o modificare campi in modo agile. Uno dei punti di forza di MongoDB è la sua facilità di scalabilità orizzontale. Scalare orizzontalmente significa aggiungere nuovi nodi al cluster per gestire un carico di lavoro crescente, inoltre, MongoDB offre una replica automatica dei dati, che assicura alta disponibilità e tolleranza ai guasti. Infine, MongoDB è ottimizzato per gestire operazioni di lettura e scrittura ad alta velocità. La capacità di MongoDB di gestire grandi volumi di operazioni simultanee senza compromettere le prestazioni è vitale per un sistema di prenotazione di successo.

- **Integrazione di MongoDb in sistemi multithread:** Integrare MongoDB in un sistema multithread client/server richiede attenzione a specifiche considerazioni. Le connessioni concorrenti, per gestire un pool di connessioni a MongoDB per evitare overhead di connessione. Assicurare che le operazioni di lettura e scrittura siano consistenti e gestire correttamente le transazioni ed, infine, utilizzare librerie come threading in Python per gestire i thread in modo efficiente.

La combinazione di un'architettura client/server multithread con MongoDB fornisce una soluzione robusta e scalabile per gestire sistemi di ticketing e prenotazione per eventi. Questa configurazione permette di rispondere efficacemente a un grande numero di richieste simultanee, garantendo al contempo la flessibilità necessaria per adattarsi a cambiamenti futuri e gestire grandi volumi di dati.

2 Descrizione del Problema

Sviluppare un sistema client-server multithread con autenticazione per la realizzazione di un sistema di ticketing e prenotazione di eventi rappresenta una sfida significativa nell'ambito delle applicazioni distribuite. Il progetto prevede la realizzazione di un'infrastruttura in grado di gestire contemporaneamente più utenti, offrendo la possibilità di fare prenotazioni in tempo reale.

2.1 Obiettivi del progetto

I principali obiettivi del progetto sono:

- **Architettura client-server:**
  - **Server:** Gestire le richieste dei client, interagire con il database MongoDB per le operazioni di lettura/scrittura e gestire la concorrenza tramite il multithreading.
  - **Client:** Interfacciarsi con il server per effettuare operazioni di visualizzazione, prenotazione e cancellazione di ticket.
- **Concorrenza e Multithreading:** Utilizzare il multithreading per permettere al server di gestire simultaneamente più richieste da parte di diversi client. Assicurarsi che le operazioni concorrenti sul database siano gestite correttamente per evitare problemi di consistenza dei dati.
- **Gestione del Db di Mongo:** Progettare e implementare il modello di dati per la gestione degli eventi e dei ticket. Implementare le operazioni CRUD (Create, Read, Update, Delete) sul database per la gestione degli eventi e delle prenotazioni.
- **Interfaccia Client:** Realizzare un'interfaccia utente semplice per il client, che può essere una CLI (Command Line Interface) o una GUI (Graphical User Interface), per consentire agli utenti di interagire con il sistema di ticketing.
- **Funzionalità del sistema di ticketing:**
  - **Visualizzazione di eventi:** Consentire ai client di visualizzare gli eventi disponibili.
  - **Prenotazione di ticket:** Permettere ai client di prenotare ticket per eventi specifici.

- **Cancellazione Prenotazioni:** Consentire ai client di cancellare le prenotazioni effettuate.

- **Sicurezza ed Autenticazione:** Implementare un sistema di autenticazione per utenti e amministratori. Assicurarsi che solo gli utenti autenticati possano effettuare prenotazioni e che solo gli amministratori possano gestire gli eventi.

- **Test e Documentazione:** Testare il sistema per assicurarsi che tutte le funzionalità funzionino correttamente e che non ci siano problemi di concorrenza o inconsistenza dei dati. Documentare il codice e fornire istruzioni su come installare, configurare e utilizzare il sistema.

2.2 Sfide e Problemi

Ci sono diverse sfide e problemi che potrebbero emergere. Ecco un elenco di alcune delle principali sfide e problemi potenziali:

- **Gestione della Concorrenza:**
  - **Concorrenza dei dati:** Assicurarsi che le operazioni concorrenti sul database non causino inconsistenza dei dati. Ad esempio, due utenti che cercano di prenotare l'ultimo biglietto disponibile simultaneamente.
  - **Locking:** Implementare un sistema di locking adeguato per evitare condizioni di gara (race conditions) e deadlock.

- **Scalabilità e prestazione:** Progettare uno schema di dati che supporti efficientemente le operazioni richieste, Assicurandosi che il server possa gestire efficacemente le connessioni al database.

- **Sicurezza:**
  - **Autenticazione ed Autorizzazione:** Implementare un sistema sicuro per l'autenticazione degli utenti e l'autorizzazione delle operazioni, garantendo che solo gli utenti autorizzati possano effettuare modifiche ai dati.
  - **Protezione:** Proteggere il sistema contro attacchi comuni come SQL Injection (nel caso di MongoDB, NoSQL Injection), Cross-Site Scripting (XSS), e attacchi di tipo DDoS.

- **Gestione degli errori:**
  - **Errori del Server:** Gestire correttamente gli errori che possono verificarsi nel server, come problemi di connessione al database o errori di lettura/scrittura.
  - **Errori del client:** Implementare un sistema robusto per la gestione degli errori lato client, in modo che gli utenti siano informati e guidati in caso di problemi.

3 Implementazione

3.1 Server:

Questo server implementa un sistema di gestione dei biglietti con funzionalità di registrazione, login, prenotazione di biglietti, aggiornamento del profilo, creazione di eventi, eliminazione di biglietti e logout. Utilizza MongoDB per la memorizzazione dei dati degli utenti, eventi e biglietti. Ecco una descrizione dettagliata di come funziona:

- **Configurazione e Inizializzazione:**
  - **Configurazione MongoDB:** Si collega a un'istanza di MongoDB locale e utilizza un database chiamato ticketing_system con tre collezioni: users, events e tickets.
  - **Blocchi di Threading:** Viene utilizzato il blocco (threading.Lock) per gestire l'accesso concorrente alle risorse condivise (in questo caso, user_lock e log_lock).
- **Funzioni del server:**
  - **log delle azioni:** Registra le azioni degli utenti in un file di log actions.log con data, ora, ID dell'utente e azione effettuata (C: create, R: read, U: update, D: delete).
  - **Gestione del client:** Gestisce le richieste dei client in modo concorrente. Riceve le richieste JSON, le decodifica e chiama la funzione appropriata in base all'azione richiesta (ad esempio, register, login, logout, list_events, book_ticket, create_event, update_account, delete_ticket), e invia la risposta al client.
  - **Registrazione e Login:**

    - **register_user(username, password, email):** Registra un nuovo utente. Cripta la password utilizzando bcrypt e memorizza i dettagli dell'utente nel database. Registra l'azione di registrazione nel file di log.
    - **login_user(username, password):** Effettua il login di un utente. Verifica le credenziali e, se valide, acquisisce il user_lock, segnala l'utente come loggato e registra l'azione di login nel file di log.
    - **logout_user(user_id):** Effettua il logout di un utente, rilascia il user_lock e registra l'azione di logout nel file di log.
  - **Gestione degli eventi:**
    - **book_ticket(user_id, event_id, num_tickets):** Prenota un biglietto per un utente specifico per un determinato evento. Controlla se l'utente ha già prenotato un biglietto e se ci sono abbastanza biglietti disponibili. Registra l'azione di prenotazione nel file di log.
    - **delete_ticket(user_id, ticket_id):** Elimina un biglietto prenotato da un utente. Aggiorna il numero di biglietti disponibili per l'evento e registra l'azione di eliminazione nel file di log.
  - **Aggiornamento del Profilo utente:**
    - **update_account(user_id, new_username, new_password, current_email):** Aggiorna le informazioni dell'account di un utente (username e password). Cripta la nuova password e registra l'azione di aggiornamento nel file di log.
  - **Caricamento degli eventi iniziali:**
    - **load_initial_events():** Carica un set di eventi iniziali nel database all'avvio del server.

- **Esecuzione del server:**
  - **main():** Carica gli eventi iniziali, configura il server per ascoltare le connessioni in entrata sulla porta 9999 e gestisce le connessioni dei client in modo concorrente utilizzando threading.

3.2 Client:

Il client è un'interfaccia per un sistema di gestione dei biglietti, che consente agli utenti di registrarsi, effettuare il login, aggiornare il proprio account, visualizzare eventi, prenotare biglietti, eliminare biglietti e fare il logout. Comunica con un server tramite socket, inviando e ricevendo messaggi in formato JSON. Ecco una descrizione dettagliata delle sue funzionalità:

- **Funzioni Ausiliarie:**
  - **Validazione Email**
    - **check(email):** Utilizza il modulo email_validator per verificare se l'email inserita è valida. Restituisce True se l'email è valida, altrimenti stampa un messaggio di errore e restituisce False.
  - **Invio richieste**
    - **send_request(action, data):** Invia una richiesta JSON al server. Includere l'azione (action) richiesta e i dati aggiuntivi (data). Riceve la risposta dal server e la decodifica dal formato JSON.
- **Funzioni principali**
  - **Aggiornamento Account:**
    - **update_account(user_id):** Richiede all'utente di inserire un nuovo username e una nuova password, quindi invia una richiesta al server per aggiornare l'account dell'utente. Utilizza l'email corrente salvata (current_email).
  - **Eliminazione Biglietto**
    - **delete_ticket(user_id):** Richiede l'elenco dei biglietti dell'utente dal server, li stampa e chiede all'utente di inserire l'ID del biglietto da eliminare. Invia una richiesta al server per eliminare il biglietto selezionato.
- **Ciclo Principale**
  - **Connessione al server**
    - **client_socket:** Configura e stabilisce una connessione al server all'indirizzo IP 127.0.0.1 sulla porta 9999.

- **Menù e scelte utente**
  - **Menù Opzioni:**
    1. **Register:** Registra un nuovo utente chiedendo username, password ed email. Verifica l'email prima di inviare la richiesta.
    2. **Login:** Effettua il login chiedendo username e password. Se il login ha successo, salva l'ID utente e l'email corrente.
    3. **Logout:** Effettua il logout dell'utente corrente se loggato.
    4. **Update acoount:** Permette all'utente loggato di aggiornare il proprio account (username e password).
    5. **List events:** Richiede e stampa l'elenco degli eventi disponibili.
    6. **Book Ticket:** Prenota un biglietto per un evento specificato. Limita la prenotazione a un solo biglietto per volta.
    7. **Delete Ticket:** Elimina un biglietto prenotato dall'utente loggato.
    8. **Exit:** Effettua il logout (se necessario) e chiude il client.

4 Descrizione del codice

4.1 Parti del server

- **Librerie e Configurazione:** Queste operazioni sono fondamentali per preparare l'ambiente di lavoro necessario per il sistema di gestione dei ticket. Una volta configurato, il sistema sarà in grado di interagire con il database MongoDB per memorizzare e recuperare informazioni sugli utenti, sugli eventi e sui biglietti in modo efficiente e sicuro.

```
        import socket
import threading
import json
from pymongo import MongoClient
from bson.objectid import ObjectId
import bcrypt
import datetime

# Configurazione MongoDB
client = MongoClient('localhost', 27017)
db = client['ticketing_system']
users_collection = db['users']
events_collection = db['events']
tickets_collection = db['tickets']
```

- **Definizione del lock:** questo codice fornisce un meccanismo di locking per garantire la sicurezza e la coerenza nell'accesso condiviso alla variabile logged_in_user e al file di log "actions.log". La funzione log_action permette di registrare in modo sicuro e sincronizzato le azioni degli utenti all'interno del sistema, facilitando la tracciatura delle attività e la risoluzione dei problemi.

```
        user_lock = threading.Lock()
log_lock = threading.Lock()
logged_in_user = None
def log_action(user_id, action):
    with log_lock:
        with open("actions.log", "a") as
           log_file:
            log_file.write(f"{datetime.datetime.
               now()}
            User ID: {user_id} - Action: {action
               }\n")
```

- **handle_client:** la funzione handle_client rappresenta il cuore del server per il sistema di gestione dei ticket, gestendo tutte le richieste dei client tramite un socket, eseguendo le azioni appropriate sul sistema e restituendo le risposte corrispondenti. Assicura anche la gestione sicura delle sessioni utente attraverso l'uso di lock e la gestione delle eccezioni per garantire una robustezza operativa.

```
def handle_client(client_socket):
    global logged_in_user
    print(f"Client {client_socket} connesso")
    try:
        while True:
            request = client_socket.recv(1024).
               decode()
            if not request:
                break

            request_data = json.loads(request)
            action = request_data['action']

            if action == 'register':
                response = register_user(
                   request_data['username'],
                   request_data['password'],
                   request_data['email'])
            elif action == 'login':
                response = login_user(
                   request_data['username'],
                   request_data['password'])
            elif action == 'logout':
                response = logout_user(
                   request_data['user_id'])
            elif action == 'list_events':
                response = list_events()
            elif action == 'book_ticket':
                response = book_ticket(
                   request_data['user_id'],
                   request_data['event_id'],
                   request_data['num_tickets'])
            elif action == 'create_event':
                response = create_event(
```

```
                    request_data['event_name'],
                    request_data['event_date'],
                    request_data['event_location'
                    ])
            elif action == 'update_account':
                response = update_account(
                    request_data['user_id'],
                    request_data['new_username'],
                     request_data['new_password'
                    ], request_data['
                    current_email'])
            elif action == 'delete_ticket':
                response = delete_ticket(
                    request_data['user_id'],
                    request_data['ticket_id'])
            else:
                response = {'status': 'error', '
                    message': 'Invalid action'}

            client_socket.send(json.dumps(
                response).encode())
    except Exception as e:
        print(f"Error: {e}")
    finally:
        if logged_in_user:
            user_lock.release()
            logged_in_user = None
        client_socket.close()
```

- **Funzioni dell'user:**
  - **Register_user:** la funzione register_user() gestisce il processo di registrazione di un nuovo utente nel sistema, garantendo che la password venga hashata in modo sicuro prima di essere memorizzata nel database MongoDB. La registrazione dell'azione nel log fornisce una traccia dell'attività di creazione degli utenti per scopi di audit e monitoraggio del sistema.

```
            def register_user(username, password,
                email):
        hashed = bcrypt.hashpw(password.encode(),
            bcrypt.gensalt())
```

```
user = {
    "username": username,
    "password": hashed,
    "email": email,
    "role": "user"
}
users_collection.insert_one(user)
log_action("N/A", "C Register")  #
   Registrazione utente (azione create)
return {'status': 'success', 'message': '
   User registered successfully'}
```

– **Login_user:** a funzione login_user() gestisce in modo sicuro l'autenticazione degli utenti nel sistema. Verifica le credenziali fornite con quelle memorizzate nel database, registra l'azione nel log e gestisce il lock dell'utente per garantire che solo un utente per volta possa essere autenticato nel sistema.

```
    def login_user(username, password):
global logged_in_user
if not user_lock.locked():
    user = users_collection.find_one({"
       username": username})
    if user and bcrypt.checkpw(password.
       encode(), user['password']):
        log_action(str(user['_id']), "R
           Login")  # Login utente (
           azione read)
        user_lock.acquire()
        logged_in_user = str(user['_id'])
        return {'status': 'success', '
           user_id': str(user['_id']), '
           email': user['email']}
    else:
        return {'status': 'error', '
           message': 'Invalid username or
            password'}
else:
    return {'status': 'error', 'message':
        'System already in use'}
```

– **Logout_user:** la funzione logout_user() gestisce il processo di logout di un utente dal sistema. Verifica la validità dell'utente che sta tentando di fare il logout, rilascia il lock dell'utente, registra l'azione di logout nel log e aggiorna lo stato di logged_in_user di conseguenza.

```
        def logout_user(user_id):
    global logged_in_user
    if user_lock.locked() and logged_in_user
       == user_id:
        user_lock.release()
        logged_in_user = None
        log_action(user_id, "R Logout")
        return {'status': 'success', 'message
           ': 'User logged out successfully'}
    else:
        return {'status': 'error', 'message':
            'Logout failed'}
```

- **Lista Eventi:** la funzione list_events() permette di ottenere una lista di tutti gli eventi presenti nel database MongoDB, convertendo gli _id degli eventi in stringhe per facilitare la manipolazione e la trasmissione dei dati. Registra anche l'azione di lettura degli eventi nel log per scopi di monitoraggio e audit del sistema.

```
def list_events():
events = list(events_collection.find())
for event in events:
    event['_id'] = str(event['_id'])
print("Eventi trovati nel database:", events
   )  # Debug
log_action("N/A", "R List Events")  # Elenco
    degli eventi (azione read)
return {'status': 'success', 'events':
   events}
```

- **Funzioni eventi:**
  - **create_event():** la funzione create_event() gestisce il processo di creazione di un nuovo evento nel sistema, salvando le informazioni dell'evento nel database MongoDB e registrando l'azione di creazione nel log per tracciare le operazioni eseguite.

```
        def create_event(event_name,
            event_date, event_location):
    event = {
        "name": event_name,
        "date": event_date,
        "location": event_location,
        "available_tickets": 100  # Numero di
            biglietti disponibili iniziali
    }
    events_collection.insert_one(event)
    log_action("N/A", "C Create Event")  #
        Creazione evento (azione create)
    return {'status': 'success', 'message': '
        Event created successfully'}
```

  - **Book_ticket:** la funzione book_ticket() gestisce in modo sicuro il processo di prenotazione di biglietti per un evento specifico nel sistema, utilizzando MongoDB per la memorizzazione dei dati e registrando le azioni pertinenti nel log per tracciare le operazioni eseguite dagli utenti.

```
        def book_ticket(user_id, event_id,
            num_tickets):
    existing_ticket = tickets_collection.
        find_one({"user_id": ObjectId(user_id)
        })
    if existing_ticket:
        return {'status': 'error', 'message':
            'You already have a ticket booked
            . Cannot book more than one.'}

    event = events_collection.find_one({"_id"
        : ObjectId(event_id)})
    if event and event['available_tickets']
        >= num_tickets:
        tickets_collection.insert_one({
```

```
            "user_id": ObjectId(user_id),
            "event_id": ObjectId(event_id),
            "number_of_tickets": num_tickets
        })
        events_collection.update_one(
            {"_id": ObjectId(event_id)},
            {"$inc": {"available_tickets": -
               num_tickets}}
        )
        log_action(user_id, "C Book Ticket")
            # Prenotazione biglietto (azione
           create)
        return {'status': 'success', 'message
           ': 'Tickets booked successfully'}
    else:
        return {'status': 'error', 'message':
            'Not enough tickets available'}
```

– **load_initial_event:** la funzione load_initial_events() è progettata per inizializzare il database con un insieme predefinito di eventi. Carica questi eventi nella collezione events_collection del database MongoDB e fornisce un feedback visivo nel terminale per informare che l'operazione è stata eseguita con successo.

```
        def load_initial_events():
    initial_events = [
        {"name": "Concerto Rock", "date": "
           2024-07-15", "location": "Stadio
           Olimpico", "available_tickets":
           150},
        {"name": "Spettacolo di Magia", "date
           ": "2024-08-10", "location": "
           Teatro Nazionale", "
           available_tickets": 200},
        {"name": "Festival del Cibo", "date":
            "2024-09-05", "location": "Piazza
            Centrale", "available_tickets":
           300}]
    events_collection.insert_many(
       initial_events)
    print("Eventi iniziali caricati con
       successo.")
```

- **def Main():** la funzione main() inizializza il sistema carreggiando gli eventi iniziali, configura un server socket TCP per ascoltare le connessioni dei client sulla porta specificata, e gestisce le connessioni dei client in thread separati utilizzando la funzione handle_client(). Questo approccio permette al server di essere reattivo e gestire simultaneamente più richieste da parte dei client.

```
def main():
# Carica eventi iniziali
load_initial_events()

server = socket.socket(socket.AF_INET,
   socket.SOCK_STREAM)
server.bind(('127.0.0.1', 9999))
server.listen(5)
print("Server listening on port 9999")

while True:
    client_socket, addr = server.accept()
    print(f"Accepted connection from {addr}"
       )
    client_handler = threading.Thread(target
       =handle_client, args=(client_socket,)
       )
    client_handler.start()

if __name__ == "__main__":
    main()
```

- **update_account:** la funzione update_account() permette agli utenti di modificare il loro nome utente e la loro password nel sistema. Gestisce l'aggiornamento dei dati utente nel database MongoDB, utilizzando il hashing per la sicurezza della password e registrando l'azione di aggiornamento nel log per tracciare le modifiche effettuate dagli utenti.

```
def update_account(user_id, new_username,
   new_password, current_email):
hashed = bcrypt.hashpw(new_password.encode()
   , bcrypt.gensalt())
users_collection.update_one(
    {"_id": ObjectId(user_id)},
```

```
        {"$set": {"username": new_username, "
            password": hashed}}
    )
    log_action(user_id, "U Update Account")  #
        Aggiornamento account (azione update)
    return {'status': 'success', 'message': '
        Account updated successfully'}
```

- **Delete_ticket:** la funzione delete_ticket() gestisce in modo sicuro il processo di cancellazione di un biglietto prenotato da parte di un utente nel sistema. Utilizza le operazioni di query e aggiornamento di MongoDB per manipolare i dati e registra le azioni pertinenti nel log per tracciare le operazioni effettuate dagli utenti.

```
def delete_ticket(user_id, ticket_id):
ticket = tickets_collection.find_one({"_id":
    ObjectId(ticket_id), "user_id": ObjectId
    (user_id)})
if ticket:
    tickets_collection.delete_one({"_id":
        ObjectId(ticket_id)})
    events_collection.update_one(
        {"_id": ObjectId(ticket['event_id'])
            },
        {"$inc": {"available_tickets":
            ticket['number_of_tickets']}}
    )
    log_action(user_id, "D Delete Ticket")
        # Cancellazione biglietto (azione
        delete)
    return {'status': 'success', 'message':
        'Ticket deleted successfully'}
else:
    return {'status': 'error', 'message': '
        Ticket not found or you do not have
        permission to delete this ticket'}
```

4.2 Parti del Client:

- **Librerie e Configurazione:** la funzione send_request(action, data) crea una richiesta basata sull'azione e sui dati forniti, invia questa richiesta al server tramite socket, riceve la risposta dal server, e restituisce la risposta come un dizionario Python. La funzione check(email) valida un indirizzo email e restituisce True se l'email è valida, altrimenti stampa un errore e restituisce False.

```
        import socket
import json
from getpass import getpass
from email_validator import validate_email,
   EmailNotValidError

def check(email):
    try:
        v = validate_email(email)
        return True
    except EmailNotValidError as e:
        print(str(e))
        return False

def send_request(action, data):
    request = {'action': action}
    if data:
        request.update(data)
    client_socket.send(json.dumps(request).
       encode())
    response = client_socket.recv(1024).decode()
    return json.loads(response)
```

- **Funzione update_account:** La funzione update_account è progettata per raccogliere i nuovi dati dell'account dall'utente, inviare questi dati al server per l'aggiornamento, e restituire la risposta del server all'utente. Tuttavia, si nota che la variabile current_email non è stata definita all'interno della funzione o passata come argomento. Pertanto, per funzionare correttamente, current_email deve essere definita nel contesto in cui viene chiamata questa funzione.

```
        def update_account(user_id):
    new_username = input("Enter new username: ")
```

```
    new_password = getpass("Enter new password:
        ")
    response = send_request('update_account', {
        'user_id': user_id,
        'new_username': new_username,
        'new_password': new_password,
        'current_email': current_email  # Usa l'
            email corrente
    })
    return response
```

- **Funzione delete_ticket:** La funzione delete_ticket(user_id) consente quindi all'utente di visualizzare i biglietti che ha prenotato, selezionare il biglietto che desidera cancellare e inviare una richiesta di cancellazione al server. Il server gestisce la cancellazione del biglietto e restituisce una risposta che indica il successo o l'eventuale errore dell'operazione.

```
        def delete_ticket(user_id):
    tickets = send_request('list_user_tickets',
        {'user_id': user_id})
    for ticket in tickets:
        print(ticket)
    ticket_id = input("Enter ticket_ID to delete
        : ")
    response = send_request('delete_ticket', {'
        user_id': user_id, 'ticket_id': ticket_id
        })
    return response
```

- **Stabilimento della connessione:** Il codice stabilisce una connessione di rete con un server locale che ascolta sulla porta 9999, pronto per scambiare messaggi. Le variabili user_id e current_email sono preparate per essere utilizzate nel contesto dell'applicazione, ad esempio, per autenticare l'utente o gestire le operazioni relative all'account dell'utente. Questo setup è fondamentale per un'applicazione client-server, dove il client si connette a un server per eseguire diverse operazioni, come registrazione, login, prenotazione di biglietti, ecc.

```
        client_socket = socket.socket(socket.
            AF_INET, socket.SOCK_STREAM)
client_socket.connect(('127.0.0.1', 9999))
```

```
user_id = None
current_email = None  # Aggiungi questa
   variabile per salvare l'email corrente
...
client_socket.close()
```

- **Blocco di condizione:** Il blocco di codice è un ciclo infinito che implementa un semplice menu interattivo per un client di un sistema di gestione eventi e prenotazione biglietti. L'utente può selezionare varie opzioni per registrarsi, effettuare il login, prenotare biglietti, ecc. Ecco una descrizione dettagliata di cosa fa il codice:

```
        while True:
    print("1. Register")
    print("2. Login")
    print("3. Logout")
    print("4. Update Account")
    print("5. List Events")
    print("6. Book Ticket")
    print("7. Delete Ticket")
    print("8. Exit")
    choice = input("Enter choice: ")

    if choice == '1':
        username = input("Enter username: ")
        password = getpass("Enter password: ")
        email = input("Enter email: ")
        if not check(email):
            continue
        response = send_request('register', {'
           username': username, 'password':
           password, 'email': email})
        if response['status'] == 'success':
            print("Registration successful!")
        else:
            print("Registration failed:",
               response['message'])

    elif choice == '2':
        username = input("Enter username: ")
        password = getpass("Enter password: ")
```

```
            response = send_request('login', {'
                username': username, 'password':
                password})
            if response['status'] == 'success':
                user_id = response['user_id']
                current_email = response['email']  #
                    Salva l'email corrente
                print("Login successful!")
            else:
                print("Login failed:", response['
                    message'])

        elif choice == '3':
            if user_id is None:
                print("Devi prima effettuare il
                    login!")
                continue
            response = send_request('logout', {'
                user_id': user_id})
            if response['status'] == 'success':
                print("Logout successful!")
                user_id = None
                current_email = None
            else:
                print("Logout failed:", response['
                    message'])

        elif choice == '4':
            if user_id is None:
                print("Devi prima effettuare il
                    login!")
                continue
            response = update_account(user_id)
            if response['status'] == 'success':
                print("Account updated successfully!
                    ")
            else:
                print("Update failed:", response['
                    message'])

        elif choice == '5':
```

```
            response = send_request('list_events',
                {})
            if response['status'] == 'success':
                print("Eventi disponibili:")
                for event in response['events']:
                    print(f"ID: {event['_id']}, Name
                        : {event['name']}, Date: {
                        event['date']}, Location: {
                        event['location']}, Tickets:
                        {event['available_tickets']}"
                        )
            else:
                print("Failed to list events:",
                    response['message'])

        elif choice == '6':
            if user_id is None:
                print("Devi prima effettuare il
                    login!")
                continue
            event_id = input("Enter event ID: ")
            num_tickets = 1  # Limita la
                prenotazione a un solo biglietto
            response = send_request('book_ticket', {
                'user_id': user_id, 'event_id':
                event_id, 'num_tickets': num_tickets
                })
            if response['status'] == 'success':
                print("Ticket booked successfully!")
            else:
                print("Booking failed:", response['
                    message'])

        elif choice == '7':
            if user_id is None:
                print("Devi prima effettuare il
                    login!")
                continue
            response = delete_ticket(user_id)
            if response['status'] == 'success':
                print("Ticket deleted successfully!"
```

```
                )
            else:
                print("Delete failed:", response['
                    message'])

        elif choice == '8':
            if user_id:
                response = send_request('logout', {'
                    user_id': user_id})
                if response['status'] == 'success':
                    print("Logout successful!")
                    user_id = None
                    current_email = None
                else:
                    print("Logout failed:", response
                        ['message'])
            break

        else:
            print("Scelta non valida. Riprova.")
            continue
```

5 Risultati

il client fornisce un'interfaccia utente per interagire con il server, mostrando messaggi di successo o errore in base all'esito delle operazioni richieste:

5.1 Account:

Se l'utente sceglie di effettuare il login, viene richiesto di inserire username e password. Viene inviata una richiesta di login al server e, se il login ha successo, l'ID dell'utente e l'email corrente vengono salvati nelle variabili user_id e current_email. Se il login fallisce, viene visualizzato un messaggio di errore.

Per il logout, se l'utente è già loggato, viene inviata una richiesta di logout al server. Se il logout ha successo, le variabili user_id e current_email vengono resettate a None. Se il logout fallisce, viene mostrato un messaggio di errore. Se l'utente non è loggato, viene richiesto di effettuare prima il login.

Se l'utente sceglie di aggiornare l'account, viene richiesto di inserire un nuovo username e una nuova password. Viene inviata una richiesta di aggior-

namento al server e, se l'operazione ha successo, viene mostrato un messaggio di conferma. In caso contrario, viene visualizzato un messaggio di errore. Anche in questo caso, se l'utente non è loggato, viene richiesto di effettuare prima il login.

5.2 Biglietto:

Per prenotare un biglietto, l'utente deve essere loggato. Viene richiesto di inserire l'ID dell'evento per cui si desidera prenotare un biglietto. La prenotazione è limitata a un solo biglietto. Viene inviata una richiesta di prenotazione al server e, se l'operazione ha successo, viene mostrato un messaggio di conferma. In caso contrario, viene visualizzato un messaggio di errore.

Per cancellare un biglietto, l'utente deve essere loggato. Viene chiamata una funzione che elenca i biglietti dell'utente e richiede di inserire l'ID del biglietto da cancellare. Viene inviata una richiesta al server per cancellare il biglietto e, se l'operazione ha successo, viene mostrato un messaggio di conferma. In caso contrario, viene visualizzato un messaggio di errore.

6 Conclusioni

Il progetto che ho implementato rappresenta un sistema di gestione eventi e prenotazione biglietti, che include funzionalità per la registrazione e il login degli utenti, la gestione degli eventi, la prenotazione e cancellazione dei biglietti, e l'aggiornamento delle informazioni dell'account. Il sistema utilizza socket per la comunicazione tra il client e il server, e MongoDB per la gestione dei dati. Questo progetto offre un'interfaccia di base ma funzionale per la gestione degli eventi, con un focus sulla semplicità e l'efficienza. Il sistema sviluppato permette di gestire le principali operazioni necessarie per un servizio di ticketing. Gli utenti possono registrarsi e accedere al sistema, visualizzare gli eventi disponibili, prenotare biglietti e gestire i propri account. Il server gestisce le richieste degli utenti e interagisce con il database MongoDB per memorizzare e recuperare i dati. La sicurezza è garantita tramite la criptazione delle password con bcrypt e l'uso di ID unici per identificare utenti ed eventi. Durante lo sviluppo del progetto, ho affrontato diverse sfide, ad esempio la sincronizzazione dei thread e la gestione sicura delle sessioni utente. 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. Sono fiduciosi che le competenze acquisite e le tecnologie implementate mi permetteranno di affrontare con successo le sfide future nel campo dell'ingegneria del software e delle comunicazioni digitali.