UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE

DIPARTIMENTO DI SCIENZE E TECNOLOGIE

CORSO DI RETI DI CALCOLATORI

ParthEventi

Relazione progetto

PROPONENTI:

Tammaro Rita 0124/2632

Treglia Alessandro 0124/2710

Centore Antonio 0124/2957

Nappo Assunta 0124/2800

Parolisi Flora 0124/2764

DATA DI CONSEGNA:

13/02/2025

ANNO ACCADEMICO:

2024-2025

Sommario

1. Traccia 3

2. Descrizione 3

3. Struttura 3

4. Comunicazione 4

a. Funzioni 5

i. Client 6

ii. Server 8

5. Manuale utente 10

a. Come installare 10

6. Database 11

1. Traccia

Creare un sistema per la gestione di tessere di accesso ad aventi. Un partecipante si registra utilizzando un client che comunica con un ente organizzatore. L'ente invia i dettagli al ServerE, che gestisce le tessere e il periodo di validità. Un ClientC consente di verificare la validità delle tessere. Un ClientO consente di sospendere o attivare l'accesso per un partecipante.

2. Descrizione

Questo progetto ha lo scopo di gestire le tessere di accesso per vari eventi.

Include funzionalità per creare, aggiornare, eliminare e visualizzare le tessere d'accesso.

Il progetto utilizza un server concorrente. Il file ''serverE.py" utilizza ThreadedTCPServer per gestire le richieste in arriva, tale server serve a dimostrare l'utilizzo di un server concorrente. ThreadedTCPServer è una classe che eredita da socketserver.ThreadingMixln e socketserver.TCPServer, il che significa che ogni richiesta che arriva viene gestita in un thread separato. Questo permette al server di gestire più connessioni contemporaneamente.

3. Struttura

Il progetto "ParthEventi" è organizzato in diverse cartelle e file che separano le funzionalità, l'interfaccia utente, la gestione dei dati e altri componenti chiave. Di seguito viene presentata la struttura dei file e la loro funzione all'interno del progetto:

- **Client**: contiene i codici relativi ai client che interagiscono con il server per registrare partecipanti, verificare la validità delle tessere e valutare l'eventuale attivaizone per consentire l'accesso ad un partecipante.

- **Common**: contiene gli script comuni, sia per quanto riguarda la parte del server che quella riferita all'utente.

- **db**: al suo interno ci sono i file relativi al database, script relativi alla memorizzazione delle informazioni sulle tessere e i partecipanti.

- **serverE**.py: implementa il server concorrente che gestisce la richieste in arrivo dai client
- **setup_db.py**: script per l'inizializzazione del database

- **view_db.py**: script per la visualizzazione dei dati registrati

- **server_address.json**: contiene la configurazione dell'indirizzo del server per consentire al client di connettersi

- **README**.rd: Panoramica del progetto, al suo interno c'è la descrizione e le istruzioni su come usarlo

- **main.py**: Script principale per eseguire l'applicazione, collegando le varie componenti (client-server, logiche, interfaccia).

- **requirements.txt**: Vengono elencate le dipendenze necessarie per il progetto (librerie Python come PyQt5, socket, ecc.).

Tecnologie utilizzate:

- Linguaggio: Python.
- Comunicazione: Socket TCP.
- Piattaforma: macOS e Windows.

4. Comunicazione

Il protocollo di comunicazione tra client e server utilizza **socket TCP** con scambio dati in **JSON**

Le comunicazioni sono suddivise in due principali categorie: **request** e **response**

Request

Le richieste sono strutturate in JSON e contengono due attributi principali: header e payload.

- **Header**: Specifica il tipo di operazione o endpoint richiesto. Ad esempio, un header come "verificaTessera" per controllare la validità di una tessera.

- **Payload**: Contiene i dati necessari per elaborare la richiesta. Ad esempio, il numero della tessera da verificare

Esempio di una richiesta JSON:

```
{
  "header": "verificaTessera",
  "payload": {
    "numero_tessera": "123456"
  }
```

```
}
```

Questa struttura viene costruita utilizzando la funzione ‘’build_communication_protocol” che genera il messaggio che inviamo al server.

Response

Le risposte dal server sono anch'esse in formato JSON e contengono un attributo result, che include i dati elaborati o lo stato dell'operazione richiesta.

Esempio di una risposta JSON:

```
{
  "result": {
    "stato": "valida",
    "scadenza": "2025-12-31"

  }
}
```

Questa risposta indica che la tessera che stiamo verificando è valida, di conseguenza riportiamo anche la sua data di scadenza.

Funzionamento del Flusso

1. **Costruzione della Richiesta**: Viene utilizzata request_constructor_obj per creare un oggetto JSON e request_constructor_str per convertirlo in una stringa JSON.
2. **Invio della Richiesta**: La funzione ‘’client_echo’’ viene utilizzata per inviare la richiesta al server. Questa funzione invia i dati attraverso il socket e attende la risposta.
3. **Elaborazione della Risposta**: La risposta del server viene letta tramite ‘’read_socket’’, decidificata e interpretat dal client per determinare l’azione successiva, come mostrare un messaggio all’utente o aggiornare l’interfaccia.

a. Funzioni

Nel sistema di comunicazione tra client e server, ciascun lato svolge un ruolo specifico e utilizza funzioni dedicate per gestire le operazioni richieste. Il lato client è responsabile di inviare richieste al server e di gestire le risposte ricevute, mentre il lato server elabora le richieste e fornisce i dati o i servizi richiesti. Di seguito, vengono descritte le principali funzioni utilizzate su entrambi i lati, evidenziando le loro differenze e il loro contributo al funzionamento complessivo del sistema.

Sul lato client, le funzioni come ***generate_card_id***, ***register_partecipant*** e ***save_credentials_to_db***, sono fondamentali per la costruzione e l'invio delle richieste, nonché per la ricezione e l'elaborazione delle risposte. Queste funzioni lavorano insieme per garantire una comunicazione efficace e affidabile con il server.

Sul lato server, le classi ***method_switch***, ***register_user***, ***validate_user, suspend_user, activate_user*** , gestiscono l'elaborazione delle richieste e la generazione delle risposte. Queste componenti server sono progettate per gestire simultaneamente più richieste, mantenendo l'integrità e la coerenza dei dati forniti ai client.

i. Client

- ***Generate_card_id***

```python
def generate_card_id():
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()
    while True:
        card_id = ''.join([str(random.randint(0, 9)) for _ in range(6)])
        cursor.execute("SELECT * FROM utenti_registrati WHERE id_tessera = ?", (card_id,))
        if not cursor.fetchone():
            break
    conn.close()
    return card_id
```

La funzione generate_card_id genera un ID tessere univoco composto da 6 cifre casuali e si assicura che non ne esista uno già nel database prima di restituirlo

1. Connessione al database: si collega al database specificato da DB_PATH
2. Generazione dell'ID tessera:
    a. Crea un numero casuale di 6 cifre (tra 0 e 9)
    b. Controlla se l'id generato è gia presente nella tabella utenti_registrati nella colonna id_tessera
    c. Se l'ID è già usato, ne genera un altro fino a trovarne uno unico.

- ***Register_partecipant***

```
def register_participant(name, surname, email, password):
    card_id = generate_card_id()
    payload = {
        "name": name,
        "surname": surname,
        "email": email,
        "password": password,
        "card_id": card_id
    }
    request_data = request_constructor_str(payload, "register")
    response = launchMethod(request_data, SERVER_ADDRESS, SERVER_PORT)
    if not response:
        return {"status": "error", "message": "Risposta vuota dal server"}, card_id
    return json.loads(response), card_id
```

La funzione register_partecipant si occupa di registrare un partecipante nel sistema e di inviare la richiesta al server.

Genra l'id della tessere chiamando ''generate_card_id()'' per ottenere un ID univoco, poi crea un dizionario con ''payload'' con al suo interno tutti i dati del partecipante.

Con ''request_constructor_str(payload, ''register'')'' si crea una richiesta con i dati e l'operazione ''register''. Ciò suggerisce l'utilizzo di una comunicazione strutturata, con un socket nel nostro caso. ''launchMetod(request_data, SERVER_ADDRESS, SERVER_PORT) invia la richiesta al server e attende una risposta.

Restituisce infine un dizionario con la risposta del server, un card_id generato.

- ***Save_credentials_to_db***

```
def save_credentials_to_db(name, surname, email, password, card_id):
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()

    try:
        # Inserisci il nuovo utente senza verificare se esiste già
        cursor.execute("""
        INSERT INTO utenti_registrati (nome, cognome, email, password, data_validita, id_tessera)
        VALUES (?, ?, ?, ?, DATE('now', '+1 year'), ?)
        """, (name, surname, email, password, card_id))

        conn.commit()
    except sqlite3.IntegrityError as e:
        pass
    finally:
        conn.close()
```

Questa funzione save_credentials_to_db(name, surname, email, password, card_id) salva le credenziali di un nuovo partecipante nel database SQLite.

Possiamo dividerla in 4 fasi fondamentalmente, la prima riguarda la connessione al database, dove si collega a quest'ultimo specificato da DB_PATH e crea un cursore per esereguire le query.

Successivamente passiamo all'inserimento di un nuovo utente con una query che in questo caso è INSERT INTO utetnti_registrati per salvare tutto ciò che riguarda l'utente.

C’è poi una fase in cui vengono gestiti eventuali errori IntegrityError.

Alla fine invece utilizziamo ‘’finally’’ per chiudere definitivamente la connessione anche in caso di errori.

ii. Server

- ***Method_switch***

```python
def method_switch(header, payload):
    if header == "register":
        return register_user(payload)
    elif header == "validate":
        return validate_user(payload)
    elif header == "suspend":
        return suspend_user(payload)
    elif header == "activate":
        return activate_user(payload)
    else:
        return {"status": "error", "message": "Invalid header"}
```

Questa funzione method_switch(header, payload) funge da **dispatcher** per instradare le richieste ai metodi appropriati in base al valore di header.

1. **Controlla il valore di header**:
    - Se header == "register", chiama register_user(payload).
    - Se header == "validate", chiama validate_user(payload).
    - Se header == "suspend", chiama suspend_user(payload).
    - Se header == "activate", chiama activate_user(payload).
    - Se il valore di header non è riconosciuto, restituisce un **errore**.

- ***Register_user***

```python
def register_user(payload):
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()

    try:
        # Verifica se l'utente esiste già
        cursor.execute("SELECT * FROM utenti_registrati WHERE email = ?", (payload['email'],))
        user = cursor.fetchone()
        if user:
            return {"status": "error", "message": "User already exists"}

        # Verifica se l'ID tessera esiste già
        cursor.execute("SELECT * FROM utenti_registrati WHERE id_tessera = ?", (payload['card_id'],))
        card = cursor.fetchone()
        if card:
            return {"status": "error", "message": "Card ID already exists"}

        # Inserisci il nuovo utente
        cursor.execute("""
        INSERT INTO utenti_registrati (nome, cognome, email, password, data_validita, id_tessera, isAttivo)
        VALUES (?, ?, ?, ?, DATE('now', '+1 year'), ?, 1)
        """, (payload['name'], payload['surname'], payload['email'], payload['password'], payload['card_id']))

        conn.commit()
        return {"status": "success", "message": "User registered successfully"}
    except sqlite3.IntegrityError as e:
        return {"status": "error", "message": "Database integrity error"}
    finally:
        conn.close()
```

Questa funzione register_user(payload) si occupa della **registrazione di un nuovo utente** nel database, garantendo che non esistano **duplicati** per email o ID tessera.

Si collega al database, verifica se l'utente esiste già, verifica se l'ID della tessera è gia un uso e dopodichè inserisce il nuovo utente.

- ***Validate,suspend,activate user***

```python
def validate_user(payload):
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()

    cursor.execute("SELECT * FROM utenti_registrati WHERE id_tessera = ? AND isAttivo = 1", (payload['card_id'],))
    user = cursor.fetchone()
    conn.close()

    if user:
        return {"status": "success", "message": "Tessera valida", "data_validita": user[6]}
    else:
        return {"status": "error", "message": "Tessera non trovata o utente non attivo"}

Qodo Gen: Options | Test this function
def suspend_user(payload):
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()

    cursor.execute("UPDATE utenti_registrati SET isAttivo = 0 WHERE email = ?", (payload['email'],))
    conn.commit()
    conn.close()

    return {"status": "success", "message": "Accesso sospeso"}

Qodo Gen: Options | Test this function
def activate_user(payload):
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()

    cursor.execute("UPDATE utenti_registrati SET isAttivo = 1 WHERE email = ?", (payload['email'],))
    conn.commit()
    conn.close()

    return {"status": "success", "message": "Accesso attivato"}
```

Queste tre funzioni (validate_user, suspend_user, activate_user) gestiscono la **validazione** e la **gestione dell'accesso** degli utenti nel sistema delle tessere.

La prima verifica se una tessera è valida e attiva, si connette al database, cerca nella tabella degli utenti registrati verifica e poi chiude la connessione.

La seconda invece sospende l'accesso di un utente disattivando la sua tessera, aggiorna il valore di ''isAttivo'' conferma la modifica e poi restituisce il messaggio di successo.

La terza infine ne riattiva l'accesso, abilitando di nuovo la tessera di un utente sospeso, ripristinando di conseguenza l'accesso.

5. Manuale utente

a. Come installare

1. Clonare la repository:

   Utilizzando il comando:

   git clone https://github.com/ritataa/ParthEventi.git

2. Naviga nella directory del progetto tramite terminale:

cd ParthEventi

3. Installa le dipendeze:

pip install -r requirements.txt

6. Database

Il sistema implementato utilizza un database SQLite per la gestione dei dati relativi agli utenti registrati. Sebbene SQLite sia un database relazione completo, nel contesto del progetto viene utilizzato principalmente per garantire un accesso strutturato e rapido alle informazioni degli utenti.

Il database viene creato e inizializzato tramite il file ''setup_db.py'', che utilizza la libreria nativa sqlite3 di Python. Il database viene salvato localmente nella cartella ''db'' con il nome ''database.db''. Qualora il database esista già, il sistema provvede alla sua eliminazione e ricreazione per garantire un'inizializzaione pulita.

La struttura del database è definita all'interno del file ''login.sql'', che specifica la creazione della tabella ''utenti_registrati''. Questa tabella contiene le seguenti colonne:

- **Id_utente**: identificativo univoco utente, chiave primaria e autoincrementato.
- **Nome e cognome**: nome e cognome del partecipante.
- **Email**: campo univoco per la mail dell'utente.
- **Password**: password dell'utente.
- **Data_registrazione**: data registrazione dell'utente, impostata di default a quella corrente.
- **Data_validità**: data di scadenza della tessera
- **Id_tesssera**: identificativo univoco della tessera.
- **isAttivo**: flag che indica se la tessera è attiva(1) o sospesa (0).

Nel contesto di un ambiente concorrente, l'accesso al database viene regolato per evitare conflitti tra i processi. L'interazione con il database avviene tramite operazioni atomiche di ''sqlite3'', che garantiscono la consistenza dei dati. Inoltre, l'uso delle transazioni assicura che ogni operazione di scrittura venga esaurita completamente o annullata in caso di errore.

Questo approccio permette una gesione efficiente delle informazioni e garantisce che solo il server abbia accesso in lettura e scrittura al database, mantenendo cosi il controllo centralizzato dei dati.