

PROGRAMMAZIONE DI RETI - UNIVERSITÀ DI BOLOGNA - A.A. 2020-2021
BARONCINI UGO UGO.BARONCINI@STUDIO.UNIBO.IT 0000842092
CONTI ALICE ALICE.CONTI7@STUDIO.UNIBO.IT 0000925054
MASTRILLI ALICE ALICE.MASTRILLI@STUDIO.UNIBO.IT 0000925517

Traccia 3 - Chat Game

Introduzione

Il progetto è stato realizzato utilizzando Python 3.9.5 e i socket TCP-IP per creare le connessioni client-server.

Il gioco realizzato prevede, dopo il login da parte del giocatore (`client`), una domanda iniziale con 3 possibili risposte: due di queste corrette, l'altra errata. Scegliendo la risposta errata, il server termina automaticamente la connessione del giocatore e si smette di giocare. Se la fase iniziale viene superata, il gioco passa alla fase successiva, in cui si susseguiranno una serie di domande. Ad ogni risposta corretta viene accreditato un punto, altrimenti viene decurtato. Il giocatore che arriva per primo ai 5 punti decreta la fine del gioco, e fa comparire la classifica con i relativi punteggi. Dopo aver mandato la classifica, il server termina tutte le connessioni.

Descrizione

L'applicazione è composta da due diversi programmi che gestiscono rispettivamente i client e il server del gioco:

- `client.py`
- `server.py`

I programmi sono entrambi eseguibili con Python `v3.9.5` . In aggiunta a questi, è presente un file JSON `config.json` in cui sono salvate le domande, le risposte e alcuni parametri di configurazione.

Strutture dati

Le principali strutture dati utilizzate sono dizionari (struttura dati che memorizza coppie chiavi-valori) e le liste. Le liste sono state utilizzate per l'implementazione dell'interfaccia grafica. La scelta del dizionario è stata fatta per comodità e perché facilita la serializzazione tramite JSON.

Server

Come si può osservare nel file `server.py` tra i parametri più rilevanti da considerare ci sono gli indirizzi IP e la porta del server, usati per aprire il socket in ascolto. Inoltre `client_sockets` è un dizionario in cui vengono aggiunti i socket dei client, man mano che si collegano. `server_socket` è l'oggetto socket su cui il server si mette in ascolto e viene attivato nella funzione `open_socket()` , chiamata una volta avviato il server.

```
def open_socket():  
    global server_socket
```

```

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_socket.bind((HOST_ADDR, HOST_PORT))
server_socket.listen(5) # allowed connection attempts

threading._start_new_thread(accept_new_clients, (server_socket,))

```

Inizialmente il socket del server viene associato all'indirizzo `HOST_ADDR` e alla porta `HOST_PORT`, dichiarati precedentemente. Successivamente il server crea un nuovo thread per ascoltare il socket e accettare le connessioni in entrata. Per ogni nuova connessione viene creato un nuovo socket per gestire la connessione con quello specifico client; connessione che viene gestita da un thread dedicato che esegue la funzione `handle_player_connection()`.

```

def accept_new_clients(server_socket):
    global client_sockets
    while True:
        client_socket, client_addr = server_socket.accept()

        # genera un ConnectionID
        id = get_next_connection_id()

        # memorizza il socket e i dati utili in una struttura, usando ConnectionID come
identificativo
        client_sockets[id] = {
            'socket': client_socket,
            'address': client_addr,
            'player_name': '',
            'id': id
        }

        # spawna un nuovo thread e gli da in gestione quella connessione (tramite
ConnectionID)
        threading._start_new_thread(handle_player_connection, (id,))

```

Richiamando la funzione `get_next_connection_id()`, viene assegnato un ID ad ogni connessione, che serve per identificare il socket nel dizionario `client_sockets`.

La gestione del player consiste nel ricevere dal client il nome di login e successivamente assegnargli un ruolo scelto casualmente dal file JSON. A questo punto si mette in attesa che un client avvii il gioco. Se non viene specificato altrimenti il metodo `recv()` di `socket` agisce in modo `blocking`, per cui aspetta fin quando non riceve dei dati. Specificando il parametro `socket.MSG_DONTWAIT` la chiamata esegue senza aspettare, lanciando un errore nel caso in cui non trovi niente. Usare questo parametro ci permette di continuare a controllare se un giocatore inizia il gioco, e allo stesso tempo smettere di controllare nel momento in cui un altro thread riceve il comando di inizio gioco.

```

while not has_game_started:
    try:
        start_msg = player_socket.recv(4096, socket.MSG_DONTWAIT).decode()
        if start_msg and start_msg == "start_game":
            has_game_started = True

```

```

        else:
            return
    except Exception as e:
        pass
    sleep(0.1)

```

Ricevuto il messaggio di inizio, viene ripetuto a tutti i client. A questo punto ogni thread procede autonomamente a gestire la sessione di gioco del giocatore.

Estrae casualmente una domanda preliminare e gliela invia. La domanda preliminare ha 3 possibili risposte, due giuste e una sbagliata. Si può procedere al gioco solo se si sceglie una delle due opzioni corrette, altrimenti il server chiude la connessione e il giocatore smette di giocare.

```

if not msg:
    print(f'{player_name}> Lost connection')
player_choice = json.loads(msg)['choice']

```

I client che hanno scelto la risposta giusta proseguono il gioco e ricevono la prima domanda.

```

# while (number of questions < X):
while score < GAME_CONFIG['winning_score'] and not is_game_over:
    # Serve question
    question = random.choice(GAME_CONFIG['questions'])
    player_socket.send(json.dumps(question).encode())

    # Wait for answer
    msg = player_socket.recv(4096).decode()
    if not msg:
        return

    # expected object
    # {'answer' : '3'}
    player_answer = json.loads(msg)['answer']

    # Check answer and assign points
    if player_answer == question['correct_answer']:
        score += 1
    else:
        score -= 1

    # Send new score to client
    score_update = {'score' : score}
    player_socket.send(json.dumps(score_update).encode())
    sleep(0.5)

```

A ogni scelta del giocatore il server valuta la risposta ricevuta, aggiorna di conseguenza il punteggio e manda ai client la domanda successiva. Questa sequenza si ripete finchè un player non arriva a 5 punti. A quel punto il gioco aspetta che i giocatori finiscano di rispondere alla domanda in corso, e viene stilata una classifica che il server manda ad ognuno.

```
# Serialize ranking
serialized_ranking = ''
for key in sorted_ranking:
    serialized_ranking += f'{key}: {sorted_ranking[key]}\n'
msg = {'ranking': serialized_ranking}

# Send ranking
player_socket.send(json.dumps(msg).encode())
```


Dopo aver mandato la classifica il server chiude le connessioni e si arresta.

Client

Una volta avviato il client viene subito richiesto un nome utente e l'avvio della connessione. Queste funzionalità vengono gestite tramite le funzioni `connect()` e `connect_to_server()`. In quest'ultima viene creato un nuovo thread su cui viene stabilita la connessione con il server (dal momento che il thread principale viene utilizzato per l'interfaccia grafica). Come in `server`, la funzione `handle_server_communication()` gestisce la comunicazione tra server e client. Dopo aver inviato al server il proprio username di login, riceve il ruolo e avvia il gioco, o attende che lo faccia un altro giocatore.

A questo punto riceverà la domanda preliminare, che verrà mostrata a video insieme alle possibili risposte; scelta una risposta viene inviata al server che deciderà se terminare la connessione o far proseguire il gioco.

Ciascun client riceverà una serie di domande, fin quando uno dei giocatori collegati non raggiungerà il punteggio prestabilito per la vittoria. Raggiunto il punteggio, ognuno può terminare la propria domanda e a quel punto riceverà la classifica.

 alt text

Thread attivi

Come precedentemente spiegato, server e client lavorano su più thread:

- Il server ha un proprio thread sempre attivo dove rimane in attesa di nuove connessioni, più tanti altri thread quanti sono i client collegati.
- Il client ha un thread per l'interfaccia grafica e un thread per gestire la comunicazione con il server e la logica di gioco.