tic-tac-toe Code Description

Matteo Sepa, Daniel Schiop, Lorenzo Muzaka, Samuel Hofer
Siam Islam Shomapto

Contents

1 Server

Questa pagina descrive le classi ed i metodi fondamentali del modulo Server

1.1 Classi

1.1.1 SocketConnection

La classe **SocketConnection** gestisce la connessione attraverso socket:

La classe specifica porta ed IP del server:

```
# Server IP and port
IP = "127.0.0.1"
PORT = 12345
```

Poi dichiara il tipo di socket e le opzioni di connessione:

```
# Socket type and options
server_socket = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

server_socket.bind((IP, PORT))
server_socket.listen()
```

1.2 Metodi

1.2.1 broadcast()

Il metodo **broadcast()** invia un messaggio di conferma a tutti i client connessi:

```
def broadcast(self, message, client_socket):
    """Metodo che invia un messaggio in broadcast a gli host connessi"""
    # Send messages to all clients except to the original sender
    for client in self.clients.keys():
        if client is not client_socket:
            client.send(message.encode("utf-8"))
```

1.2.2 handle()

Il metodo **handle()** viene chiamata per ogni client connesso e gestisce le connessioni con i client

```
def handle(self, client_socket):
    """Metodo che gestisce la connessione"""
```

1.2.3 receive()

Il metodo **receive()** gestisce le connessioni, i turni di gioco ed i thread per ogni client.

Il metodo inizializza la connessione, notifica il client e setta il nickname al client:

```
def receive(self):
    """Metodo che gestisce le connessioni e fa partire il thread"""
    global turn
    while True:
        # Accept Connection
        client_socket, address = self.server_socket.accept()
        print("Connected with {}".format(str(address)))
        client_socket.send(self.turn.encode('utf-8'))
        nickname = f'player {self.turn}'
```

Poi setta il turno della partita:

```
if self.turn == "x":
    self.turn = "o"
elif self.turn == "o":
    self.turn = "x"
self.clients.update({client_socket: nickname})
```

In fine avvia il thread:

```
thread = threading.Thread(target=self.handle, args=(client_socket,))
thread.start()
```

2 Client

Questa pagina descrive le classi ed i metodi fondamentali del modulo Server

2.1 Classi

2.1.1 SocketChat

La classe **SocketChat** gestisce le connessioni al socket

Metodo che inizializza la connessione:

```
def __init__(self):
    """Metodo che inizializza la connessione"""
    self.nickname = ""
    # Server Ip and Port
    self.IP = "127.0.0.1"
    self.PORT = 12345
    self.client_socket = socket.socket(
        family=socket.AF_INET, type=socket.SOCK_STREAM
    )
```

2.1.2 Game

La classe **Game** contiene la logica di gioco

L'attributo *winning_ states* contiene le combinazioni che determinano la vincita della partita:

```
winning_states = [
    [(0, 0), (0, 1), (0, 2)],
    [(1, 0), (1, 1), (1, 2)],
    [(2, 0), (2, 1), (2, 2)],
    [(0, 0), (1, 0), (2, 0)],
    [(0, 1), (1, 1), (2, 1)],
    [(0, 2), (1, 2), (2, 2)],
    [(0, 0), (1, 1), (2, 2)],
    [(0, 2), (1, 1), (2, 0)],
]
```

Inizializzazione della classe:

```
def __init__(self):
    super().__init__()
    self.player = ""
    self.turn = "x"
    self.initUI()
    self.p1_score = 0
    self.p2_score = 0
```

2.2 Metodi

2.2.1 receive()

Il metodo **receive()** della classe **SocketChat** riceve il messaggio dal server:

```
def receive(self):
    """Metodo che riceve il messaggio dal server"""
    message = self.client_socket.recv(1024).decode("utf-8")
    return message
```

2.2.2 write()

Il metodo **write()** della classe **SocketChat** invia il messaggio di conferma al server:

```
def write(self, msg: str):
    message = msg
    self.client_socket.send(message.encode("utf-8"))
```

2.2.3 initUI()

Il metodo **initUI** della classe **Game** inizializza l'interfaccia grafica:

```
def initUI(self):
    """Metodo che inizializza l'interfaccia grafica"""
    self.game_size = 3
    self.buttons = [
        [],
        [],
        [],
    ]
    grid = QGridLayout()
    self.setLayout(grid)
```

Poi crea i pulsanti:

```
for i in range(self.game_size):
        for j in range(self.game_size):
            button = QPushButton()
            button.setFixedSize(200, 200)
            button.clicked.connect(self.takeTurn(button, i, j))
            font = button.font()
            font.setPointSize(60)
            button.setFont(font)
            grid.addWidget(button, i, j)
            self.buttons[i].append(button)
```

Ed infine assegna dei metodi ai pulsanti.

2.2.4 initDatabase()

Il metodo **initDatabase()** della classe **Game** crea la tebella contenente i risultati delle partite nel database nel caso non esistesse.

Crea una connessione con il server di MariaDB:

```
def initDatabase(self):
    """Metodo che inizializza la connessione al database"""
    # Connect to MariaDB Platform
    conn = mariadb.connect(
        user="tictactoeuser",
        password="password",
        host="localhost",
        database="tictactoe")
    cur = conn.cursor()
```

Poi crea la tabella nel caso non esistesse:

```
statement = ("CREATE TABLE IF NOT EXISTS results ("
        "gameID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,"
        "p1_score INT(25),"
        "p2_score INT(25),"
        "date DATE"
        ");"
        )
        cur.execute(statement)
        conn.commit()
```

Ed infine chiude la connessione:

```
conn.close()
```

2.2.5 writeToDbServer()

Il metodo **writeToDbServer()** della classe **Game** scrive i risultati della partita nel database

Crea una connessione con il server di MariaDB:

```
def initDatabase(self):
    """Metodo che inizializza la connessione al database"""
    # Connect to MariaDB Platform
    conn = mariadb.connect(
        user="tictactoeuser",
        password="password",
        host="localhost",
        database="tictactoe")
    cur = conn.cursor()
```

Poi inserisce i dati nel database:

```
statement = (
            "INSERT INTO results (p1_score,p2_score,date) "
            "VALUES (%s,%s,%s)"
            )
        items_to_insert = (p1_score, p2_score, date)

        cur.execute(statement, items_to_insert)
        conn.commit()
```

Ed infine chiude la connessione:

```
conn.close()
```

2.2.6 newGame()

Il metodo **newGame()** della classe **Game** fa iniziare una nuova partita azzerando i componenti dell'interfaccia grafica:

```
def newGame(self):
    """Metodo che fa iniziare una nuova partita"""
    for row in self.buttons:
        for btn in row:
            btn.setText("")
```

2.2.7 checkGame()

Il metodo **chackGame()** della classe **Game** controlla lo stato della partita facendo riferimento al attributo winning_states:

```
def checkGame(self):
    """Metodo che controlla lo stato della partita"""
    win = ""
    for win_state in Game.winning_states:
```

Nel caso di vincita aumenta il punteggio del giocatore che ha vinto e richiama il metodo **newGame()**:

```
win = state
print(f"'{win}' vince")
self.player_won_label.setText("{} ha vinto".format(win))
if win == 'x':
    self.p1_score = self.p1_score + 1
else:
    self.p2_score = self.p2_score + 1
self.newGame()
```

Stessa cosa in caso di pareggio senza però alterare il punteggio dei giocatori.

2.2.8 otherPlayerTurn()

Il metodo **otherPlayerturn()** della classe **Game** avvia il thread che gestisce il turno per il prossimo giocatore:

```
def otherPlayerTurn(self):
    threading.Thread(target=self._otherPlayerTurn).start()
```

2.2.9 toggleTurn()

Il metodo **toggleTurn()** della classe **Game** alterna i turni durante la partita:

```
def toggleTurn(self):
    """Metodo che alterna i turni durante la partita"""
    if self.turn == "x":
        self.turn = "o"
    else:
        self.turn = "x"
    self.turn_label.setText("{}\nTurn".format(self.turn))
```

2.2.10 endTurn()

Il metodo **endTurn()** della classe **Game** fa terminare il turno in corso:

```
def endTurn(self):
    self.toggleTurn()
    self.checkGame()
    self.otherPlayerTurn()
```