

Università degli studi di Napoli Parthenope

Dipartimento di Scienze e Tecnologie



Relazione del progetto Reti di Calcolatori

Gym Management System

Studenti:

Mario Esposito
Salvatore Gargano
Augusto Pio Tontaro
Carmela Di Biase

Professore:

Umberto Scafuri

ANNO ACCADEMICO 2024/2025

Indice

Traccia	1
1 Descrizione del progetto	2
1.1 Schemi dell'architettura	2
1.1.1 Client	2
1.1.2 Server	2
1.2 Deployment Diagram	3
2 Schemi del protocollo applicazione	4
2.1 Comunicazione TCP/IP	4
2.1.1 Gestione della concorrenza	4
2.1.2 Dettagli delle porte	5
2.1.3 Flusso delle connessioni	5
2.1.4 Gestione delle connessioni e delle interruzioni	5
2.2 Flusso delle operazioni	6
2.2.1 Registrazione di un abbonamento	6
2.2.2 Verifica di un abbonamento	6
2.2.3 Modifica dello stato di un abbonamento	7
2.3 Formato delle richieste	7
2.4 Formato delle risposte	8
3 Manuale utente	10
3.1 Prerequisiti	10
3.2 Scaricare il progetto	10
3.3 Configurazione dell'ambiente	11
3.4 Esecuzione del server	11
3.5 Esecuzione dei client	12

4	Simulazione dell'applicazione	13
4.1	Avvio del server A	13
4.2	Avvio del Server G	13
4.3	Registrazione abbonamento	14
4.4	Modifica stato	15
4.5	Verifica stato	15
4.6	Messaggi di debug dei server	16
4.6.1	Messaggi di debug di Server A	16
4.6.2	Messaggi di debug di Server G	17

Elenco delle figure

1.1	UML Deployment Diagram	3
4.1	Avvio del server A	13
4.2	Avvio del server G	14
4.3	Registrazione di un abbonamento	14
4.4	Modifica stato di un abbonamento	15
4.5	Verifica stato di un abbonamento	16
4.6	Debug serverA	17
4.7	Debug serverG	18

Traccia

Progettare un sistema per gestire abbonamenti e palestre. Un cliente, dopo aver sottoscritto un abbonamento, utilizza un client per registrarlo presso il sistema centrale della palestra. La palestra comunica al ServerA i dettagli dell'abbonamento e la sua durata. Un ClientP consente di verificare se un abbonamento è attivo, comunicando con il ServerG, che interagisce con il ServerA per il controllo. Un ClientM permette di sospendere o riattivare un abbonamento, ad esempio a seguito di richieste di pausa o rinnovo.

1. Descrizione del progetto

Il progetto è stato interamente scritto in **Python**, utilizzando i socket per gestire la comunicazione tra client e server. È stato realizzato per simulare un sistema di gestione degli abbonamenti a una palestra, con l'utilizzo di un database **SQLite** per la memorizzazione dei dati. Il progetto è compatibile con sistemi operativi Unix-like e Windows, ed è stato sviluppato utilizzando **PyCharm**.

1.1 Schemi dell'architettura

1.1.1 Client

- **Palestra.py (client principale)**: Usato dall'utente per registrare un abbonamento alla palestra, specificando il nome del cliente e la durata. Il client comunica con **ServerA**, che registra l'abbonamento nel database.
- **clientP.py (client verifica)**: Consente di verificare lo stato di un abbonamento esistente, utilizzando l'ID dell'abbonamento. Questo client si collega al **ServerG**, che inoltra la richiesta di verifica a **ServerA**.
- **clientM.py (client modifica)**: Permette di modificare lo stato di un abbonamento (ad esempio, da "attivo" a "sospeso"). Comunica direttamente con **ServerA**.

1.1.2 Server

- **ServerA (server principale)**: È il cuore del sistema, che si occupa della gestione degli abbonamenti. Riceve richieste dai client per registrare nuovi abbonamenti, modificare lo stato di abbonamenti esistenti e rispondere alle verifiche inviate tramite **ServerG**. Inoltre, è l'unico server che comunica direttamente con il database **SQLite** per eseguire operazioni di lettura e scrittura.

- **ServerG (server intermedio)**: Funziona come intermediario tra **clientP.py** e **ServerA**. Riceve richieste di verifica da **clientP.py** e le inoltra a **ServerA** per ottenere i dati richiesti, mantenendo separata la logica di verifica dalla gestione diretta del database.

1.2 Deployment Diagram

Per rappresentare l'architettura del sistema è stato realizzato un **Deployment Diagram UML**, che mostra i componenti principali (client, server e database) e le connessioni TCP/IP tra di essi.

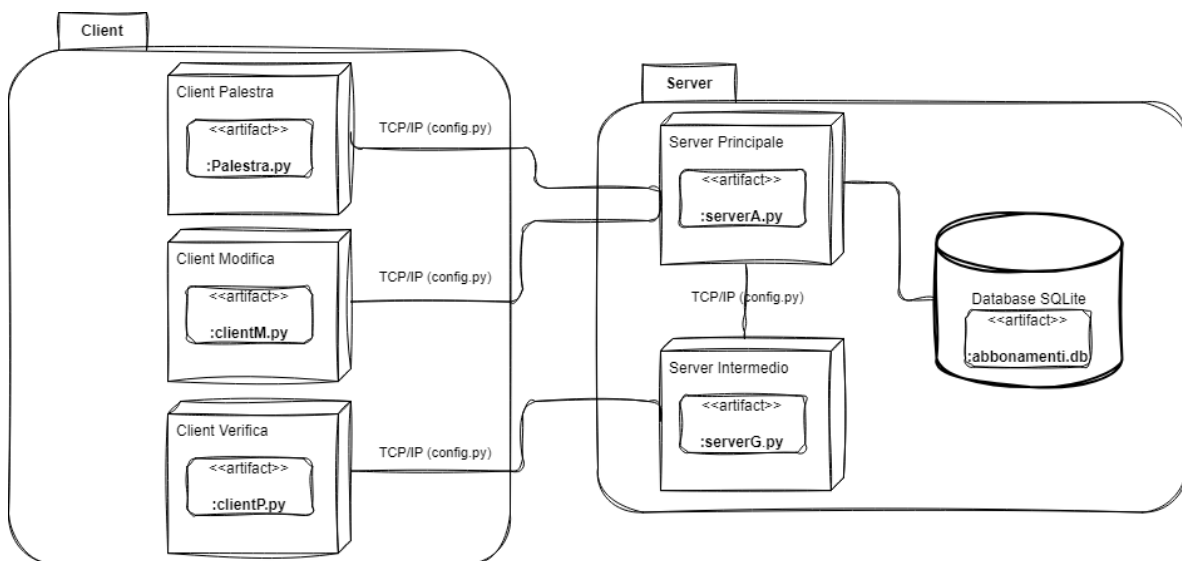


Figura 1.1: UML Deployment Diagram

2. Schemi del protocollo applicazione

Il protocollo applicativo implementato nel progetto si basa sulla comunicazione client-server tramite socket **TCP/IP**. Il protocollo definisce i formati delle richieste e delle risposte scambiate tra i client e i server, garantendo la corretta elaborazione delle operazioni richieste dagli utenti. Ogni server è configurato per ascoltare su una porta specifica e gestire richieste dai client o da altri server.

2.1 Comunicazione TCP/IP

2.1.1 Gestione della concorrenza

Ogni server crea un nuovo **thread** per gestire ciascuna connessione. Questo approccio garantisce che il sistema possa gestire richieste multiple in parallelo, mantenendo la separazione tra le connessioni e migliorando la flessibilità operativa. Il *mutex* viene utilizzato in **serverA** per sincronizzare l'accesso al database SQLite. Poiché il server è multithreaded, più thread possono tentare di eseguire operazioni sul database contemporaneamente. Questo potrebbe causare condizioni di competizione o corruzione dei dati.

Il **mutex** garantisce che:

- Solo un thread alla volta possa accedere al database per eseguire operazioni di lettura o scrittura.
- Le operazioni sui dati siano atomiche, preservando la coerenza del database.

Ad esempio, durante l'inserimento di un nuovo abbonamento o l'aggiornamento dello stato di un abbonamento, il *mutex* protegge la sezione critica del codice. In questo modo, anche se più richieste vengono elaborate simultaneamente, il database rimane in uno stato consistente.

2.1.2 Dettagli delle porte

Di seguito viene riportato la tabella dell'assegnazioni delle porte descritta nel file `config.py`:

Server	Porta
ServerA	5001
ServerG	5002

Tabella 2.1: Assegnazione porte ai server

2.1.3 Flusso delle connessioni

1. I client (`Palestra.py`, `clientP.py`, `clientM.py`) stabiliscono una connessione **TCP/IP** con i rispettivi server (`ServerA` o `ServerG`) per inviare richieste e ricevere risposte.
2. `ServerG` utilizza una connessione **TCP/IP** per inoltrare le richieste di verifica a `ServerA` e ricevere i dati richiesti.
3. I dati scambiati tra client e server sono strutturati in formato JSON, che garantisce leggibilità e interoperabilità.

Origine	Destinazione	Porta	Tipo di comunicazione
<code>Palestra.py</code>	<code>ServerA</code>	5001	TCP/IP
<code>clientP.py</code>	<code>ServerG</code>	5002	TCP/IP
<code>clientM.py</code>	<code>ServerA</code>	5001	TCP/IP
<code>ServerG</code>	<code>ServerA</code>	5001	TCP/IP

Tabella 2.2: Riepilogo connessioni

2.1.4 Gestione delle connessioni e delle interruzioni

Per garantire che i server possano essere chiusi in modo controllato senza lasciare risorse in uno stato inconsistente viene utilizzato un segnale di interruzione. I segnali di

interruzione, come il **SIGINT** (attivato tramite la combinazione **Ctrl+C**), sono utilizzati per garantire una chiusura ordinata dei server. Quando un segnale di interruzione viene catturato, il server esegue le seguenti operazioni:

- Termina il ciclo principale e interrompe l'accettazione di nuove connessioni.
- Chiude in modo sicuro tutti i socket aperti, rilasciando le risorse associate.
- Completa o interrompe in modo sicuro eventuali operazioni attive, come connessioni al database o thread in esecuzione.

2.2 Flusso delle operazioni

Il flusso delle operazioni principali del sistema prevede tre tipi di interazioni tra i client e i server.

2.2.1 Registrazione di un abbonamento

1. Il client `Palestra.py` invia una richiesta di registrazione a **ServerA**, contenente il nome del cliente e la durata dell'abbonamento.
2. **ServerA** registra l'abbonamento nel database SQLite, assegnando automaticamente un ID progressivo e impostando lo stato iniziale su **attivo**.
3. **ServerA** restituisce al client una conferma della registrazione e l'ID generato.

2.2.2 Verifica di un abbonamento

1. Il client `clientP.py` invia una richiesta di verifica a **ServerG**, specificando l'ID dell'abbonamento.
2. **ServerG** inoltra la richiesta a **ServerA**.
3. **ServerA** recupera i dati dell'abbonamento dal database e li restituisce a **ServerG**.
4. **ServerG** invia la risposta finale al client, che visualizza i dettagli dell'abbonamento (nome cliente, durata, stato) o un messaggio di errore se l'ID non è valido.

2.2.3 Modifica dello stato di un abbonamento

1. Il client `clientM.py` invia una richiesta di modifica dello stato a `ServerA`, contenente l'ID dell'abbonamento e il nuovo stato (`attivo` o `sospeso`).
2. `ServerA` aggiorna lo stato nel database e restituisce al client una conferma dell'avvenuta modifica.

2.3 Formato delle richieste

Le richieste inviate dai client ai server sono strutturate come **oggetti JSON**. Ogni richiesta contiene i seguenti campi:

- **azione**: Specifica il tipo di operazione richiesta (es. `registra`, `verifica`, `modifica`).
- **id**: Identificativo univoco dell'abbonamento (obbligatorio per alcune azioni come `verifica` e `modifica`).
- **nome_cliente**: Nome del cliente (utilizzato solo nella registrazione di un nuovo abbonamento).
- **durata**: Durata dell'abbonamento in mesi (utilizzato solo nella registrazione).
- **stato**: Nuovo stato dell'abbonamento, che può essere `attivo` o `sospeso` (utilizzato nella modifica dello stato).

Esempi di richieste:

```
1 {  
2     "azione": "registra",  
3     "nome_cliente": "Mario Rossi",  
4     "durata": 12  
5 }
```

Code 2.1: Richiesta di registrazione

```
1 {  
2     "azione": "verifica",  
3     "id": 1  
4 }
```

Code 2.2: Richiesta di verifica

```
1 {  
2     "azione": "modifica",  
3     "id": 1,  
4     "stato": "sospeso"  
5 }
```

Code 2.3: Richiesta di modifica dello stato

2.4 Formato delle risposte

Le risposte inviate dai server ai client sono strutturate come **oggetti JSON**. Ogni risposta contiene i seguenti campi:

- **status**: Indica l'esito dell'operazione (**successo** o **errore**).
- **message**: Messaggio descrittivo dell'esito dell'operazione.
- **dati**: Contiene i dettagli di un abbonamento richiesti (in caso di verifica).

Esempi di risposte:

```
1 {  
2     "status": "successo",  
3     "message": "Abbonamento registrato",  
4     "id": 1  
5 }
```

Code 2.4: Risposta di registrazione

```
1 {  
2   "status": "successo",  
3   "dati": {  
4     "id": 1,  
5     "nome_cliente": "Mario Rossi",  
6     "durata": 12,  
7     "stato": "attivo"  
8   }  
9 }
```

Code 2.5: Risposta di verifica (successo)

```
1 {  
2   "status": "errore",  
3   "message": "Abbonamento non trovato"  
4 }
```

Code 2.6: Risposta di verifica (errore)

```
1 {  
2   "status": "successo",  
3   "message": "Stato aggiornato"  
4 }
```

Code 2.7: Risposta di modifica dello stato

3. Manuale utente

In questa sezione viene descritto il processo per installare, configurare ed eseguire il progetto *Gym Management System*. Si presuppone che l'utente disponga di un sistema operativo Windows e che abbia accesso al terminale o a un prompt dei comandi.

3.1 Prerequisiti

Prima di procedere con l'utilizzo del progetto, assicurati di avere installati i seguenti strumenti:

- **Python 3.8 o superiore:** Puoi scaricare Python dal sito ufficiale (<https://www.python.org/>). Durante l'installazione, assicurati di selezionare l'opzione per aggiungere Python al PATH.
- **Git:** È necessario per clonare il repository del progetto. Puoi installarlo eseguendo il comando seguente:

```
1 pip install git
```

3.2 Scaricare il progetto

Per scaricare il codice sorgente del progetto, esegui i seguenti comandi nel terminale:

1. Clona il repository Git:

```
1 git clone https://github.com/luxifersamael/gym-  
management-system.git
```

2. Accedi alla cartella del progetto:

```
1 cd gym-management-system
```

3.3 Configurazione dell'ambiente

Per configurare correttamente il progetto, segui i passaggi indicati:

1. Crea un ambiente virtuale:

```
1 python3 -m venv .venv
```

2. Attiva l'ambiente virtuale (per sistemi Windows):

```
1 .venv\Scripts\activate
```

3. Attiva l'ambiente virtuale (nel caso di sistemi linux):

```
1 source ../.venv/bin/activate
```

3.4 Esecuzione del server

Dopo aver configurato l'ambiente, esegui i seguenti comandi:

1. Inizializza il database (se non è già stato inizializzato):

```
1 python3 server/serverA.py
```

2. Avvia il server intermedio (`serverG`):

```
1 python3 server/serverG.py
```

3.5 Esecuzione dei client

Il progetto include diversi client per interagire con il sistema. Di seguito sono riportate le istruzioni per utilizzare ciascun client:

- Registrare un nuovo abbonamento:

```
1 python3 clients/Palestra.py
```

- Verificare lo stato di un abbonamento:

```
1 python3 clients/clientP.py
```

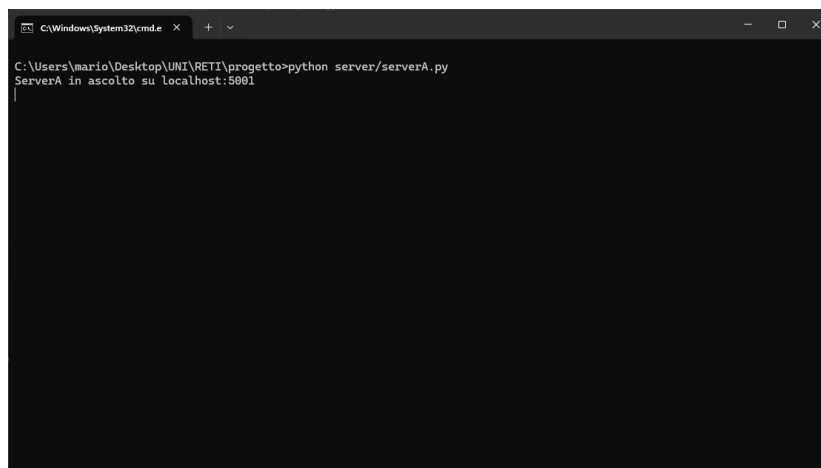
- Modificare lo stato di un abbonamento:

```
1 python3 clients/clientM.py
```


4. Simulazione dell'applicazione

4.1 Avvio del server A

Quando Server A viene avviato, inizializza il database verificando l'esistenza della tabella degli abbonamenti e creandola se necessario. Successivamente, si mette in ascolto su una porta configurata, pronto a gestire connessioni in arrivo dai client. Il server è configurato per rispondere alle richieste in modo concorrente e può essere interrotto in maniera ordinata tramite il segnale **SIGINT**.

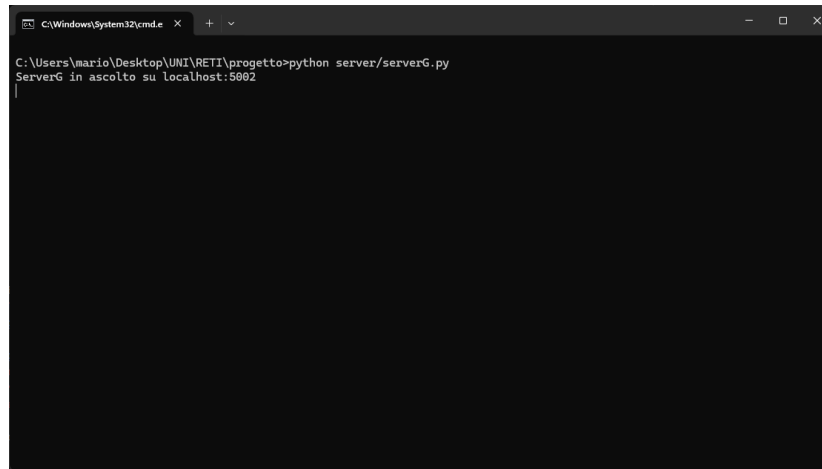


```
C:\Windows\System32\cmd.exe
C:\Users\mario\Desktop\UNI\RETI\progetto>python server/serverA.py
ServerA in ascolto su localhost:5001
```

Figura 4.1: Avvio del server A

4.2 Avvio del Server G

All'avvio, Server G si mette in ascolto su una porta configurata per ricevere richieste dai client. Ogni connessione in arrivo viene gestita in modo concorrente, inoltrando le richieste a Server A per ottenere le informazioni richieste. Anche Server G può essere interrotto ordinatamente tramite il segnale **SIGINT**, garantendo una chiusura sicura delle risorse utilizzate.

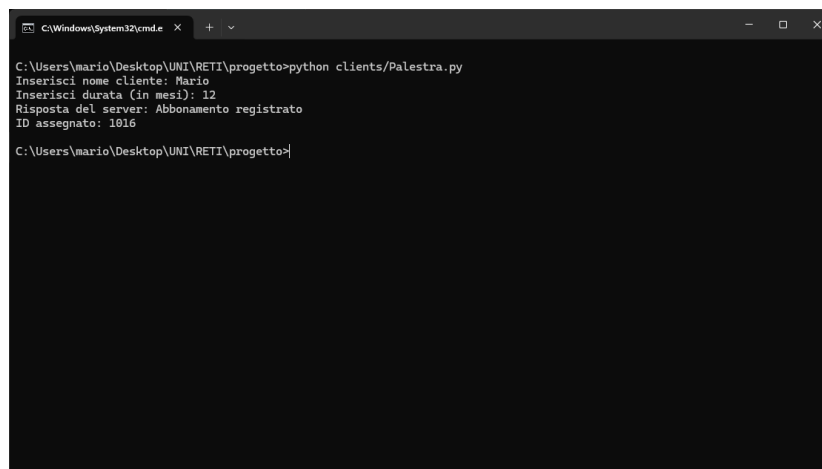


```
C:\Windows\System32\cmd.exe
C:\Users\mario\Desktop\UNI\RETI\progetto>python server/serverG.py
ServerG in ascolto su localhost:5902
```

Figura 4.2: Avvio del server G

4.3 Registrazione abbonamento

Quando il client **Palestra.py** viene avviato, richiede all'utente di inserire i dati necessari per la registrazione di un nuovo abbonamento, come il nome del cliente e la durata dell'abbonamento. Una volta inseriti i dati, il client stabilisce una connessione con Server A e invia la richiesta di registrazione. Server A elabora i dati, salva l'abbonamento nel database e restituisce un messaggio di conferma con l'ID assegnato, che viene mostrato all'utente.

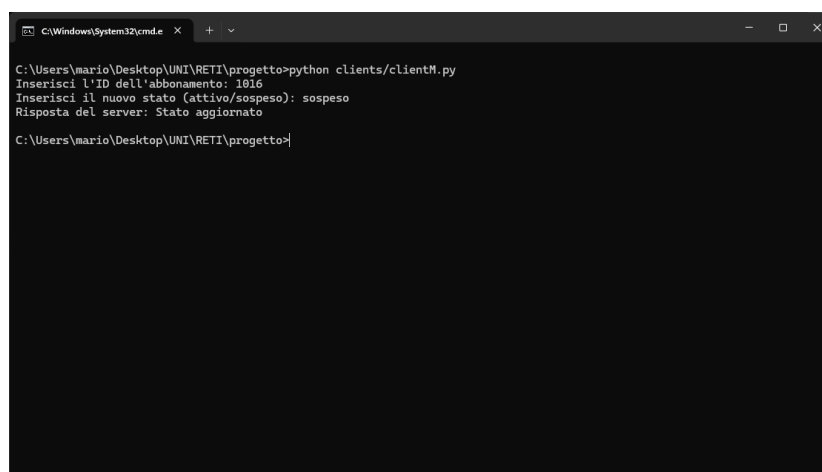


```
C:\Windows\System32\cmd.exe
C:\Users\mario\Desktop\UNI\RETI\progetto>python clients/Palestra.py
Inserisci nome cliente: Mario
Inserisci durata (in mesi): 12
Risposta del server: Abbonamento registrato
ID assegnato: 1016
C:\Users\mario\Desktop\UNI\RETI\progetto>
```

Figura 4.3: Registrazione di un abbonamento

4.4 Modifica stato

All'avvio, il client **clientP.py** consente all'utente di verificare lo stato di un abbonamento esistente, richiedendo l'inserimento dell'ID dell'abbonamento. Il client stabilisce una connessione con Server G, inviando la richiesta di verifica. Server G inoltra la richiesta a Server A, che accede al database per controllare i dati dell'abbonamento. Una volta completata l'operazione, il risultato viene restituito a Server G e poi mostrato all'utente tramite **clientP.py**.

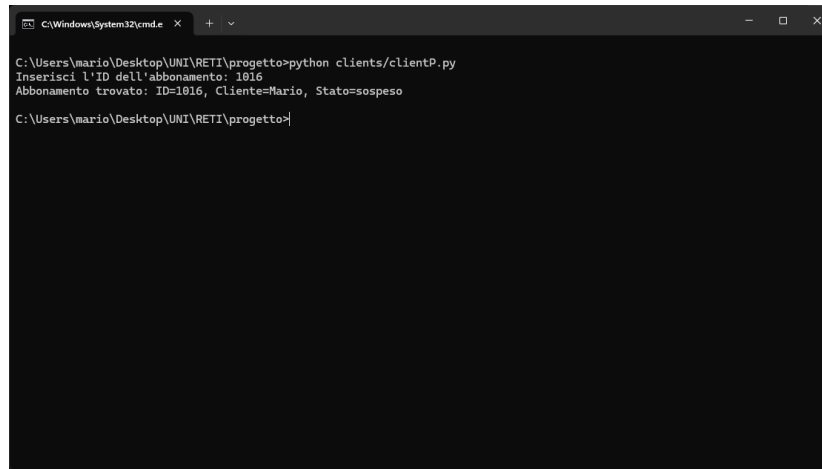


```
C:\Windows\System32\cmd.exe
C:\Users\mario\Desktop\UNI\RETI\progetto>python clients/clientM.py
Inserisci l'ID dell'abbonamento: 1016
Inserisci il nuovo stato (attivo/sospeso): sospeso
Risposta del server: Stato aggiornato
C:\Users\mario\Desktop\UNI\RETI\progetto>
```

Figura 4.4: Modifica stato di un abbonamento

4.5 Verifica stato

Quando il client **clientM.py** viene avviato, permette all'utente di modificare lo stato di un abbonamento esistente, richiedendo l'inserimento dell'ID dell'abbonamento e del nuovo stato desiderato (**attivo** o **sospeso**). Il client stabilisce una connessione con Server A e invia la richiesta di modifica. Server A aggiorna il database con le nuove informazioni e restituisce un messaggio di conferma, che viene visualizzato all'utente.



```
C:\Windows\System32\cmd.exe
C:\Users\mario\Desktop\UNI\RETI\progetto>python clients/clientP.py
Inserisci l'ID dell'abbonamento: 1016
Abbonamento trovato: ID=1016, Cliente=Mario, Stato=sospeso
C:\Users\mario\Desktop\UNI\RETI\progetto>
```

Figura 4.5: Verifica stato di un abbonamento

4.6 Messaggi di debug dei server

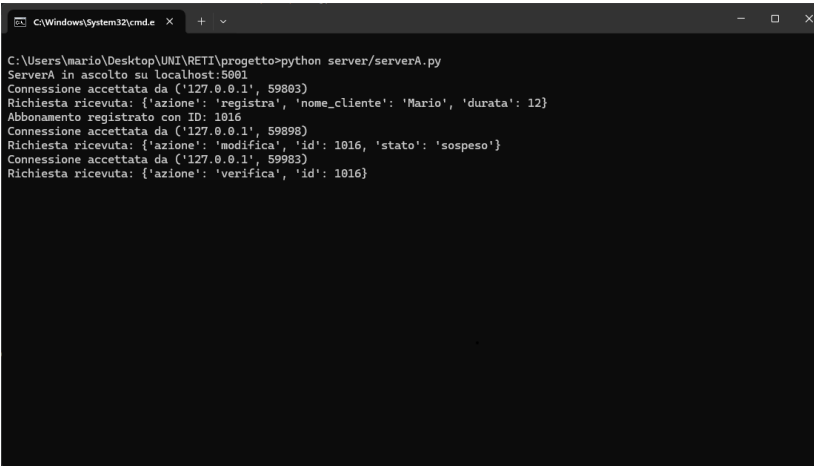
Durante l'esecuzione dei server, vengono stampati sul terminale una serie di messaggi di debug che permettono di monitorare lo stato e le operazioni eseguite.

4.6.1 Messaggi di debug di Server A

Quando Server A è in esecuzione, il terminale mostra informazioni utili per il tracciamento delle operazioni:

- La conferma di avvio del server con l'indirizzo e la porta in ascolto.
- Le connessioni accettate dai client, indicando l'indirizzo IP e la porta del client connesso.
- Il contenuto delle richieste ricevute, con dettagli sull'azione richiesta (**registra**, **verifica**, **modifica**).
- Risultati delle operazioni, come la conferma di registrazione di un nuovo abbonamento, il recupero dei dati di un abbonamento esistente o l'aggiornamento dello stato.
- Eventuali errori, come ID non trovati nel database o azioni non riconosciute.

Esempio di messaggi:



```
C:\Users\mario\Desktop\UNI\RETI\progetto>python server/serverA.py
ServerA in ascolto su localhost:5981
Connessione accettata da ('127.0.0.1', 59883)
Richiesta ricevuta: {'azione': 'registra', 'nome_cliente': 'Mario', 'durata': 12}
Abbonamento registrato con ID: 1016
Connessione accettata da ('127.0.0.1', 59898)
Richiesta ricevuta: {'azione': 'modifica', 'id': 1016, 'stato': 'sospeso'}
Connessione accettata da ('127.0.0.1', 59983)
Richiesta ricevuta: {'azione': 'verifica', 'id': 1016}
```

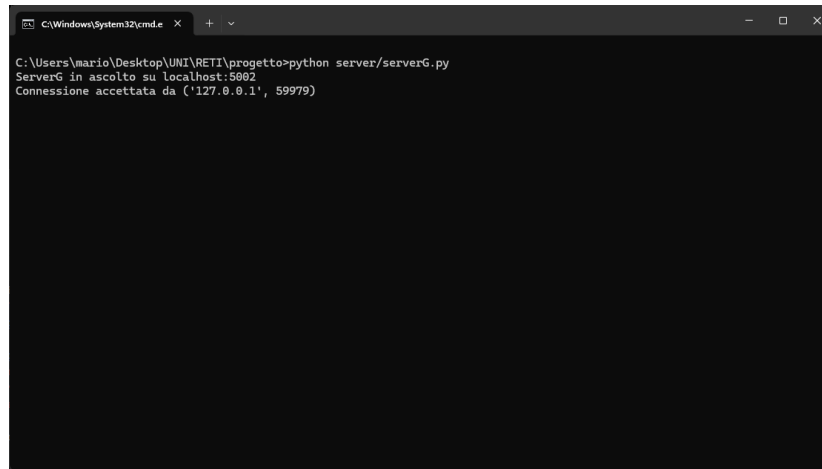
Figura 4.6: Debug serverA

4.6.2 Messaggi di debug di Server G

Anche Server G produce messaggi di debug per monitorare il suo funzionamento:

- La conferma di avvio del server con l'indirizzo e la porta in ascolto.
- Le connessioni accettate dai client per la verifica degli abbonamenti.
- Il contenuto delle richieste ricevute dai client e l'inoltro delle richieste a Server A.
- I risultati ricevuti da Server A e inoltrati al client richiedente.
- Eventuali errori nella comunicazione con Server A o con il client.

Esempio di messaggi:



```
C:\Windows\System32\cmd.exe
C:\Users\mario\Desktop\UNI\RETI\progetto>python server/serverG.py
ServerG in ascolto su localhost:5902
Connessione accettata da ('127.0.0.1', 59979)
```

Figura 4.7: Debug serverG

Questi messaggi sono fondamentali per verificare il corretto funzionamento del sistema e individuare rapidamente eventuali problemi durante l'esecuzione.