Relazione Reti UniWorld

1 Indice

Sommario

Giovanni Marsala 0121003154
Emanuele Ruppi 0124003084
Samuele Guadagno 0124003091
Paolino Santella 0124003093

2. Traccia

L'obiettivo del progetto è la realizzazione di un'applicazione **Client-Server parallela** (denominata *UniWorld*) progettata per la gestione informatizzata degli appelli d'esame e delle prenotazioni universitarie.

Il sistema deve garantire la gestione concorrente di più utenti e soddisfare i seguenti requisiti funzionali, suddivisi per tipologia di utente:

Segreteria:

- **Gestione Esami:** Deve poter inserire nuovi appelli, corsi e lauree sul server dell'università, con salvataggio persistente dei dati.
- **Gestione Prenotazioni:** Deve poter visualizzare le richieste di prenotazione inviate dagli studenti e inoltrarle al sistema centrale (accettazione/rifiuto).
- **Consultazione:** Deve fornire agli studenti le date degli esami disponibili in risposta alle loro richieste.

Studente:

- **Richiesta Informazioni:** Deve poter interrogare il sistema per sapere se ci sono appelli disponibili per un determinato corso.
- **Prenotazione:** Deve poter inviare una richiesta di prenotazione per un esame specifico.

Server Universitario:

- **Concorrenza:** Deve gestire connessioni multiple simultanee (segreteria e studenti).
- **Persistenza:** Deve ricevere e memorizzare l'aggiunta di nuovi esami e le prenotazioni.
- **Protocollo di Prenotazione:** Ad ogni richiesta di prenotazione, il server deve generare un **numero di prenotazione progressivo** univoco assegnato allo studente. Tale numero viene comunicato alla segreteria che, a sua volta, lo inoltra allo studente come conferma.

3. Introduzione

Il progetto, denominato **UniWorld**, consiste nella realizzazione di un sistema software distribuito basato su architettura **Client-Server** per la gestione informatizzata delle sessioni d'esame e delle prenotazioni universitarie.
L'applicazione è stata sviluppata interamente nel linguaggio di programmazione **Python**, sfruttando la libreria **PyQt5** per la creazione di interfacce grafiche (GUI) user-friendly e reattive. Il sistema prevede due livelli di accesso distinti:

- **Segreteria:** con privilegi amministrativi per la gestione di corsi, appelli e validazione delle richieste.
- **Studente:** con permessi per la consultazione degli appelli e l'invio di richieste di prenotazione.

L'accesso alle funzionalità è protetto da una procedura di autenticazione (Login). Il componente centrale dell'architettura è il **Server Universitario**, il quale funge da gestore delle risorse. Esso si occupa delle operazioni di lettura e scrittura sul database (implementato tramite file CSV) e risponde in maniera **parallela e concorrente** alle richieste provenienti dai diversi Client connessi, utilizzando protocolli di comunicazione basati su **Socket TCP**.

4. Struttura

Il codice sorgente è organizzato in una struttura modulare per separare logicamente l'interfaccia utente, la logica applicativa e la gestione dei dati. Di seguito viene illustrata l'organizzazione delle directory:

- **root (Cartella Principale):** Contiene i file di avvio del sistema, tra cui main.py (entry point unico per Client e Server), combined_multiplex_concurrent_server.py (gestione delle connessioni socket) e server_side.py (smistamento delle richieste).

- **common:** Contiene le librerie e gli script di utilità condivisi tra le varie parti del progetto, come il protocollo di comunicazione (communication.py) e le funzioni di I/O (full_read.py, full_write.py).

- **db:** Rappresenta il livello di persistenza dei dati, organizzato in file CSV:
  - **esami:** Contiene i dati strutturali dell'ateneo (appelli.csv, corsi.csv, laurea.csv).
  - **prenotazioni:** Memorizza le richieste e le prenotazioni effettuate dagli studenti (richiesteDateEsami.csv, richiestePrenotazioneEsami.csv).
  - **users:** Contiene le credenziali di accesso criptate per Studenti e Segreteria.

- **gui:** Contiene i file relativi al *Presentation Layer* (Interfaccia Grafica):
  - **designer:** Contiene i file sorgenti .ui generati con Qt Designer.
  - **segreteria:** File Python convertiti per le interfacce della Segreteria.
  - **students:** File Python convertiti per le interfacce degli Studenti.

- **logic:** Contiene il *Business Logic Layer*, ovvero gli script Python che gestiscono il comportamento delle finestre, collegano i segnali dei pulsanti alle funzioni e gestiscono la comunicazione di rete con il Server. È suddivisa anch'essa in segreteria e students.

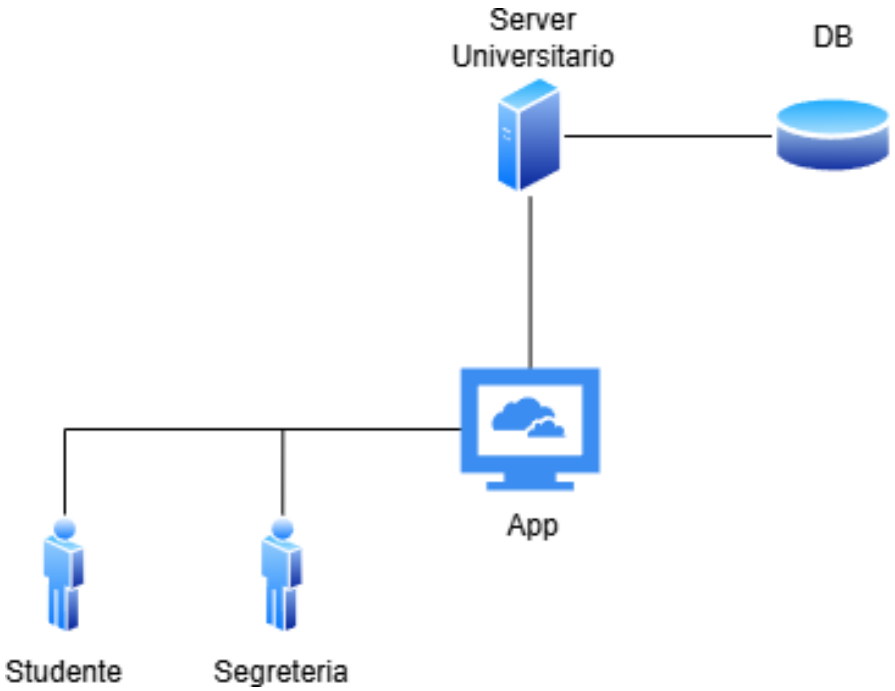

Figure 1: Connessioni

4.1 Connessioni

Figure 1 Descrive come avvengono le connessioni. I 2 diversi tipi di utenti effettuano richieste al server attraverso l' applicazione. Il server `e l' unica entità che legge e scrive sul DB

5. Comunicazione

La comunicazione tra i nodi della rete (Client Studente, Client Segreteria e Server Universitario) avviene mediante l'utilizzo di **Socket TCP** (Transmission Control Protocol). La scelta del protocollo TCP garantisce un canale di trasmissione affidabile (*reliable*), assicurando che i pacchetti arrivino integri e nell'ordine corretto, aspetto fondamentale per la consistenza dei dati universitari (es. evitare che una prenotazione vada persa).

Lo scambio di informazioni avviene serializzando i dati nel formato **JSON** (JavaScript Object Notation), che permette di trasmettere strutture dati complesse (come dizionari e liste) sotto forma di stringhe di testo.

5.1 Request

La fase di *Request* rappresenta l'invio di un comando dal Client al Server. Ogni qualvolta un utente effettua un'azione sull'interfaccia grafica (es. preme "Login" o "Prenota"), il software costruisce un pacchetto dati strutturato.

Nel file communication.py è definita la struttura standard di ogni richiesta, che si compone di un dizionario JSON con due chiavi principali:

- **Header**: Una stringa che funge da identificativo univoco della funzione che il server deve eseguire (es. "StudentsLogin", "InsertAppello"). Questo permette al server di sapere immediatamente come processare i dati.
- **Payload**: Un oggetto (dizionario o lista) che contiene i parametri necessari all'esecuzione del comando (es. Matricola e Password, oppure i dettagli di un esame).

Esempio di struttura Request (JSON):

```
{
  "header": "StudentsLogin",
  "payload": {
      "Matricola": "0124003084",
      "Password": "test123"
  }
}
```

5.2 Give

In questo secondo esempio, viene presentata la tipologia di JSON definita **Give** (Risposta). Questo formato è caratterizzato da un singolo attributo fondamentale, **result**, che contiene l'esito elaborato dall'endpoint richiesto. Nel caso specifico mostrato di seguito, si tratta del risultato di un login studente andato a buon fine:

```
"result": [
    "0124003084",
    "Emanuele",
    "Ruppi",
    "emanuele.ruppi001@studenti.uniparthenope.it",
    "1811819030",
    "0124"
]
```

I dati contenuti nell'attributo result vengono restituiti sotto forma di **array** e vengono successivamente interpretati ed elaborati direttamente lato Client.

A livello implementativo, la funzione deputata ad effettuare le chiamate agli endpoint è **launchMethod**, la quale richiede tre parametri di input:

- input (str): la stringa JSON della richiesta;
- server_address (str): l'indirizzo IP del server;
- server_port (int): la porta di comunicazione.

Per migliorare l'usabilità del codice e garantire una maggiore modularità, la funzione launchMethod è stata abbinata alle due funzioni ausiliarie **request_constructor_str** e **request_constructor_obj**. Nello specifico, request_constructor_obj restituisce il JSON in formato *object* (dizionario), che viene passato a request_constructor_str per essere serializzato in stringa e infine inviato a launchMethod, delineando il seguente **flow operativo**:

```
def request_constructor_obj(input_object, header):
    return {
        "header": header,
        "payload": input_object
    }

def request_constructor_str(input_object, header):
    return json.dumps(request_constructor_obj(input_object, header))

# Esempio di chiamata nel flusso:
launchMethod(request_constructor_str(None, "GetCorsi"),
```

```
server_coords['address'], server_coords['port'])
```

5.3 Funzioni per la comunicazione

5.3.1Client

In questa sezione vengono analizzate le funzioni implementate nel file SelMultiplexClient.py e nella libreria comune common, responsabili della gestione della comunicazione lato Client.

5.3.2 full_write(fd, buf)

```
def full_write(fd, buf):
    nleft = len(buf)
    while nleft > 0:
        try:
            nwritten = fd.send(buf)
            nleft -= nwritten
            buf = buf[nwritten:]
        except socket.error as e:
            if e.errno == errno.EINTR:
                continue
            else:
                raise
    return nlef
```

Questa funzione ha il compito fondamentale di garantire che tutti i dati presenti nel buffer buf vengano effettivamente inviati sul file descriptor fd (la socket). Il funzionamento è il seguente:

1. **Calcolo di nleft:** Inizializza il contatore dei byte rimanenti con la lunghezza totale del buffer.

2. **Ciclo di invio:** Continua a ciclare finché ci sono dati da inviare (nleft > 0).

3. **Invio effettivo:** Utilizza il metodo fd.send(buf) per inviare una porzione di dati.

4. **Aggiornamento:** Sottrae il numero di byte realmente scritti (nwritten) da nleft e aggiorna il buffer rimuovendo la parte già inviata (slicing).

5. **Gestione errori:** Intercetta eccezioni di tipo socket.error. Se l'errore è un'interruzione di sistema (errno.EINTR), il ciclo continua senza interrompersi; altrimenti, l'eccezione viene sollevata.

5.3.3. read_socket(sock)

```
async def read_socket(sock):
    received_data = ""
    while True:
        recvbuff = await asyncio.to_thread(sock.recv, MAXLINE)
        if not recvbuff:
            print("[CLIENT] Nessun dato ricevuto (EOF)")
            break
        if not recvbuff.strip():
            print("[CLIENT] Dati vuoti. Chiusura connessione.")
            break
        received_data += recvbuff.decode()
    return received_data
```

Funzione asincrona deputata alla lettura dei dati dalla socket fino alla ricezione del segnale di EOF (End Of File).

1. **Accumulo:** Inizializza una stringa vuota received_data.
2. **Lettura Asincrona:** Utilizza asyncio.to_thread per eseguire la chiamata bloccante sock.recv in un thread separato, senza congelare l'interfaccia grafica.
3. **Controllo Chiusura:** Se non vengono ricevuti dati (EOF) o se i dati sono vuoti, interrompe il ciclo.
4. **Decodifica:** I byte ricevuti vengono decodificati in stringa e concatenati al risultato finale.

5.3.4 client_echo(data, sock)

```
async def client_echo(data, sock):
    # Scrive i dati usando full_write in un thread separato
    await asyncio.to_thread(full_write, sock, data.encode())

    # Chiude il canale di scrittura per segnalare fine invio
    await asyncio.to_thread(sock.shutdown, socket.SHUT_WR)

    # Legge la risposta
    return await read_socket(sock)
```

Funzione asincrona che orchestra il ciclo richiesta-risposta:

1. **Invio:** Invia i dati codificati al server tramite full_write.
2. **Shutdown:** Chiude il canale di scrittura della socket (SHUT_WR) per segnalare al server che

l'invio della richiesta è terminato.

3. **Ricezione:** Attende e restituisce la risposta elaborata dal server invocando read_socket

5.3.5. launchMethod(input_str, server_address, server_port)

```
def launchMethod(input_str: str, server_address: str, server_port: int):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    serv_add = (server_address, server_port)

    try:
        sock.connect(serv_add)
        print(f"[CLIENT] Connesso con successo a {serv_add}")
    except Exception as e:
        print(f"[CLIENT] Errore di connessione: {e}")
        return None

    result = asyncio.run(client_echo(input_str, sock))

    # Chiusura finale
    sock.close()
    return result
```

È la funzione principale ("entry point") per la comunicazione:

1. **Setup:** Crea una socket TCP/IP (SOCK_STREAM).
2. **Connessione:** Tenta la connessione all'indirizzo e porta specificati. Gestisce eventuali errori di connessione stampando il log nel terminale.
3. **Esecuzione:** Avvia il loop asincrono di asyncio per eseguire client_echo.
4. **Cleanup:** Chiude la socket e restituisce il risultato (la stringa JSON di risposta) al chiamante.

5.3.6 Server

In questa sezione viene analizzato il funzionamento del Server, implementato nel file combined_multiplex_concurrent_server.py (o server_core.py). Il server è il cuore pulsante dell'architettura UniWorld, progettato per essere concorrente e gestire molteplici connessioni simultanee.

5.3.7Classe MyHandler

Questa classe, che eredita da socketserver.StreamRequestHandler, definisce la logica di gestione per ogni singola connessione client.

```
class MyHandler(socketserver.StreamRequestHandler):
    def handle(self):
        host, port = self.client_address
        print(f"[SERVER] Richiesta ricevuta da {host}:{port}")

        while True:
            try:
                # Lettura dati (bufferizzata)
                data = self.rfile.readline(MAXLINE)
                if not data:
                    print(f"[SERVER] Connessione chiusa dal client
{self.client_address}")
                    break

                # Parsing e Decodifica
                data_decoded = data.decode().replace('\n', '')
                data_decoded = json.loads(data_decoded)

                # Elaborazione tramite Dispatcher (method_switch)
                result = method_switch(data_decoded["header"],
data_decoded["payload"])

                # Serializzazione e Invio Risposta
                response = f"{json.dumps(result)}".encode("utf-8")
                full_write(self.request, response)
```

```
            except socket.error as e:
                print(f"Errore Socket: {e}")
                break
            except Exception as e:
                print(f"Errore Generico: {e}")
                break
```

Il metodo handle esegue i seguenti passaggi logici:

1. **Ciclo di Ascolto:** Entra in un ciclo while True per mantenere viva la comunicazione finché il client non si disconnette.
2. **Elaborazione (Dispatcher):** Invoca la funzione method_switch (definita in server_side.py), passando l'Header e il Payload. Questa funzione agisce da "smistatore", eseguendo la logica di business richiesta (es. login, prenotazione) e restituendo il risultato.
3. **Gestione Errori:** Intercetta disconnessioni improvvise o errori imprevisti per evitare il crash dell'intero server.
4. **Identificazione:** Registra e stampa a video l'indirizzo IP e la porta del client appena connesso.
5. **Invio Risposta:** Il risultato viene serializzato nuovamente in JSON (json.dumps), codificato in byte e inviato al client tramite la funzione full_write, che garantisce l'invio completo del pacchetto.
6. **Lettura e Parsing:** Legge i byte dalla socket, li decodifica in stringa e utilizza json.loads per trasformarli in un oggetto Python (dizionario).

5.3.8 Classe ThreadedTCPServer

```
class ThreadedTCPServer(socketserver.ThreadingMixIn,
socketserver.TCPServer):
    pass
```

Questa classe è fondamentale per soddisfare il requisito di parallelismo del progetto. Ereditando da socketserver.ThreadingMixIn e socketserver.TCPServer, permette al server di creare un nuovo Thread indipendente per ogni client che si connette. In questo modo, se un client sta eseguendo un'operazione lunga, gli altri client (Segreteria o altri Studenti) non vengono bloccati ma possono continuare a operare parallelamente.

5.3.9 Funzione server_main

```
def server_main(server_address, server_port):
    # Creazione istanza server multithread
    server = ThreadedTCPServer((server_address, server_port), MyHandler)

    # Avvio del server in un thread separato
    server_thread = threading.Thread(target=server.serve_forever)
    server_thread.start()
    print(f"[SERVER] In ascolto sulla porta {server_port}...")

    try:
        # Mantiene il main thread attivo
        server_thread.join()
    except KeyboardInterrupt:
        print("[SERVER] Server terminato dall'utente")
```

Questa funzione gestisce il ciclo di vita del server:

1. **Istanziazione:** Crea l'oggetto server collegandolo all'indirizzo IP e alla porta specificati (9000).
2. **Avvio asincrono:** Lancia il metodo serve_forever in un thread separato, permettendo al server di accettare connessioni indefinitamente.
3. **Gestione Chiusura:** Rimane in attesa (join) finché non riceve un segnale di interruzione (es. chiusura dell'applicazione), garantendo uno spegnimento pulito.

6. Manuale Utente

6.1 Installazione e Avvio (Linux/Ubuntu)

Di seguito vengono illustrati i passaggi per configurare l'ambiente di esecuzione su una distribuzione Linux (Ubuntu/Debian).

1. **Aggiornamento del sistema:**

```
sudo apt update && sudo apt upgrade
```

2. **Installazione di Python 3.11 e Git:** Se non presenti nel sistema, installare Python, il gestore pacchetti pip e Git:

```
sudo apt install python3 python3-pip python3-venv git
```

3. **Download del Progetto:** Scaricare la cartella del progetto **UniWorld** o clonare il repository:

```
git clone https://github.com/emanuele121004/UniWorld.git
```

4. **Configurazione della cartella:** Spostarsi nella directory del progetto:

```
cd UniWorld
```

5. **Creazione Virtual Environment (Opzionale ma consigliato):** Creare un ambiente isolato per non creare conflitti con le librerie di sistema:

```
python3 -m venv .venv

source .venv/bin/activate
```

6. **Installazione delle dipendenze Python:** Installare le librerie necessarie (PyQt5) elencate nel file di configurazione:

```
pip install -r requirements.txt
```

7. **Installazione dipendenze grafiche di sistema:** Per il corretto funzionamento di PyQt5 su Linux, è necessario installare le seguenti librerie di supporto:

```
sudo apt-get install libxcb-xinerama0 libxkbcommon-x11-0 libxcb-icccm4
libxcb-image0 libxcb-keysyms1 libxcb-randr0 libxcb-render-util0 libxcb-
xinerama0 libxcb-xfixes0 libegl1-mesa
```

8. **Avvio dell'applicazione:** Il sistema è progettato per avviare automaticamente sia il processo Server che l'Interfaccia Grafica Client tramite un unico entry-point. Eseguire il comando:

```
python3 main.py
```

6.2 Installazione e Avvio (Windows)

1. **Assicurarsi di aver installato Python 3.**
2. **Aprire il terminale (CMD o PowerShell) nella cartella del progetto UniWorld.**
3. **Installare le dipendenze:**

```
pip install -r requirements.txt
```

4. Avviare il programma:

```
python main.py
```

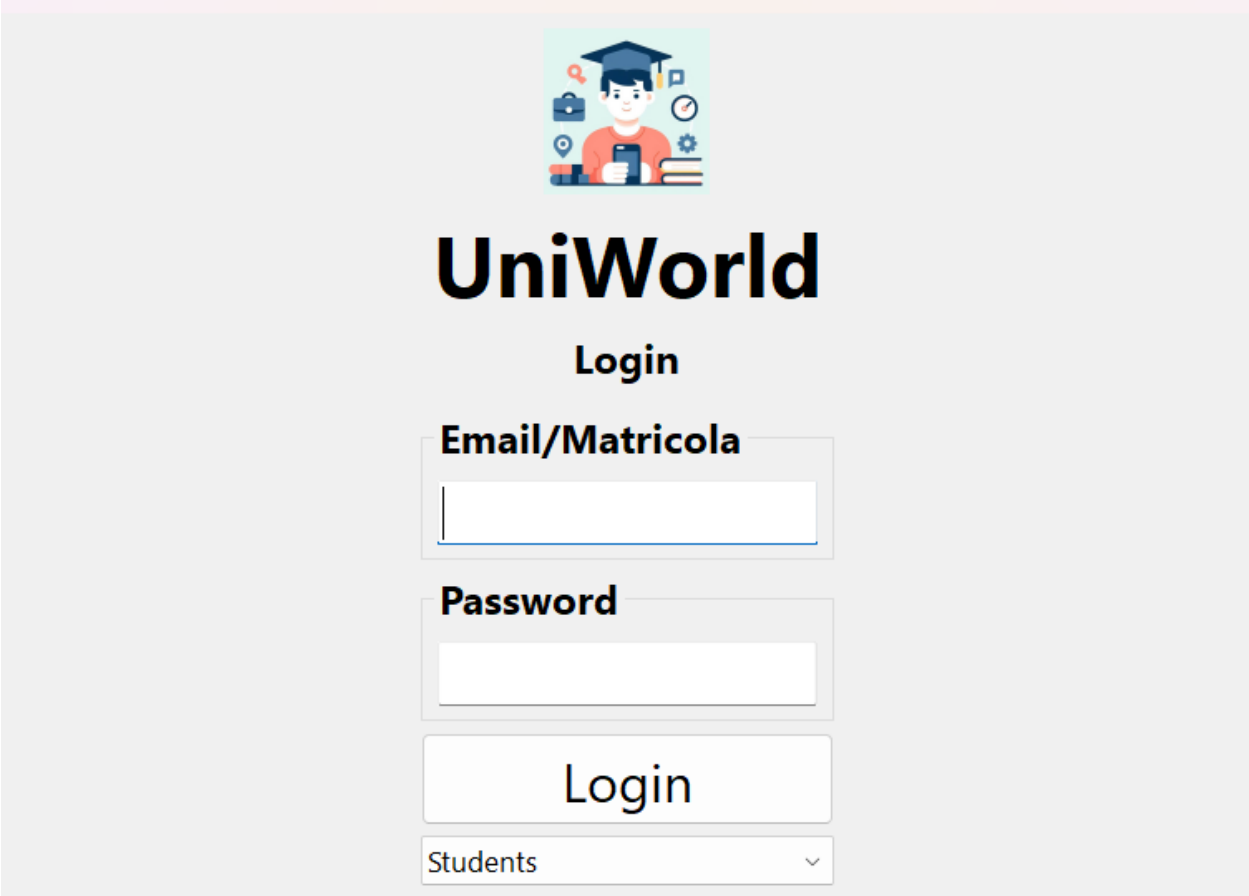

Figure 2: Interfaccia Login

6.3 Login

Figura 2 rappresenta l'interfaccia di login, si necessita di inserire e-mail/matricola in caso di studente e password personale, gli utenti vengono inseriti manualmente nei file CSV con le password crittografate tramite un algoritmo custom. Bisogna inoltre specificare la tipologia di utente che sta eseguendo l'accesso.
Se le credenziali inserite sono corrette si apre l' interfaccia Home dell'utente selezionato, altrimenti si ottiene un messaggio di errore "email, password or user type incorrest. Check your info and retry"

6.4 Segreteria

Quando un utente della segreteria effettua il login, viene mostrata questa interfaccia dove `e possibile accedere a tutte le funzionalità

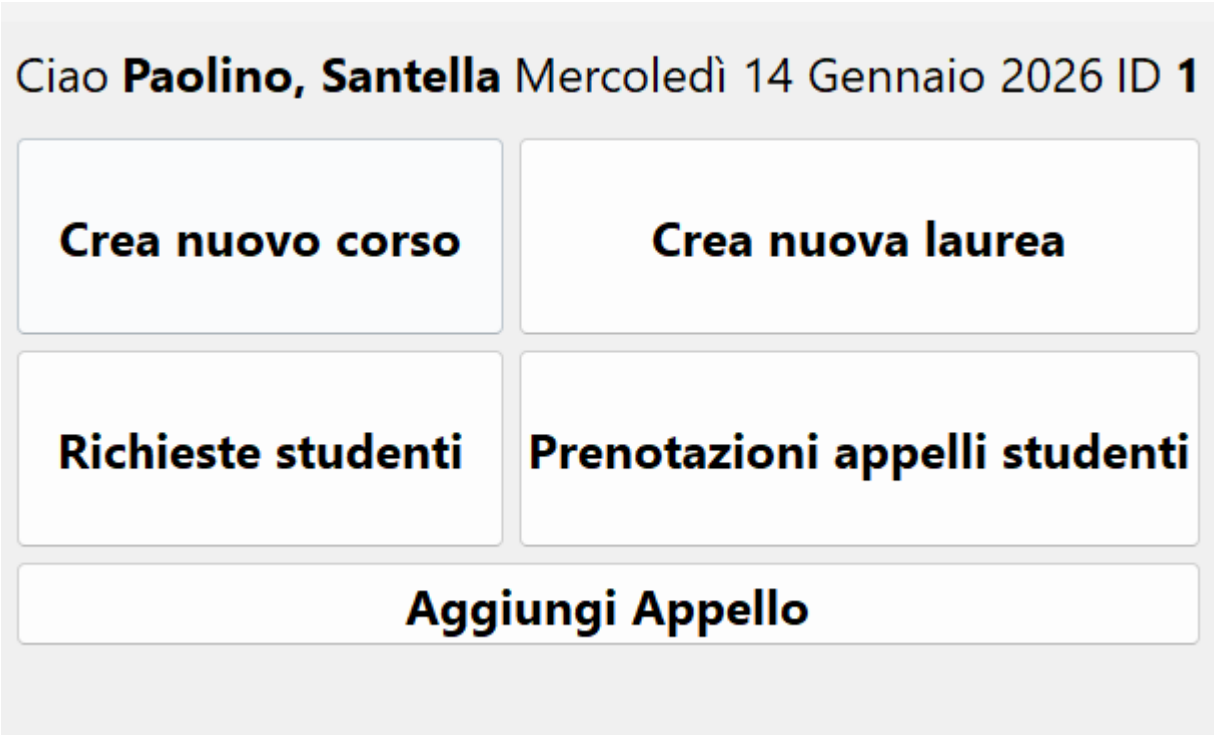

Figure 3: Home segreteria

6.4.1 Aggiungi Corso Di Laurea

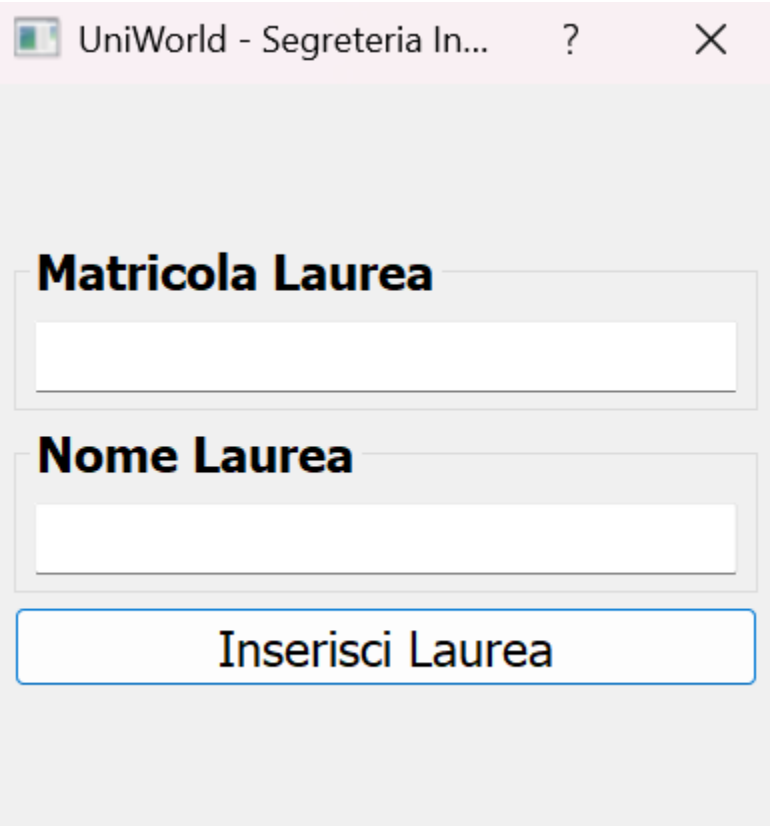

Figure 4: Aggiungi Laurea

La figura 4 mostra l'interfaccia aggiungi corso di laurea, i campi richiesti sono la matricola e il nome di riconoscimento della laurea. Una volta inserititi i dati e premuto sul pulsante "Inserisci laurea", verrà mostrato un messaggio di avvenuto inserimento del corso di laurea se non si verifica nessun errore.

6.4.2 Aggiungi Appello

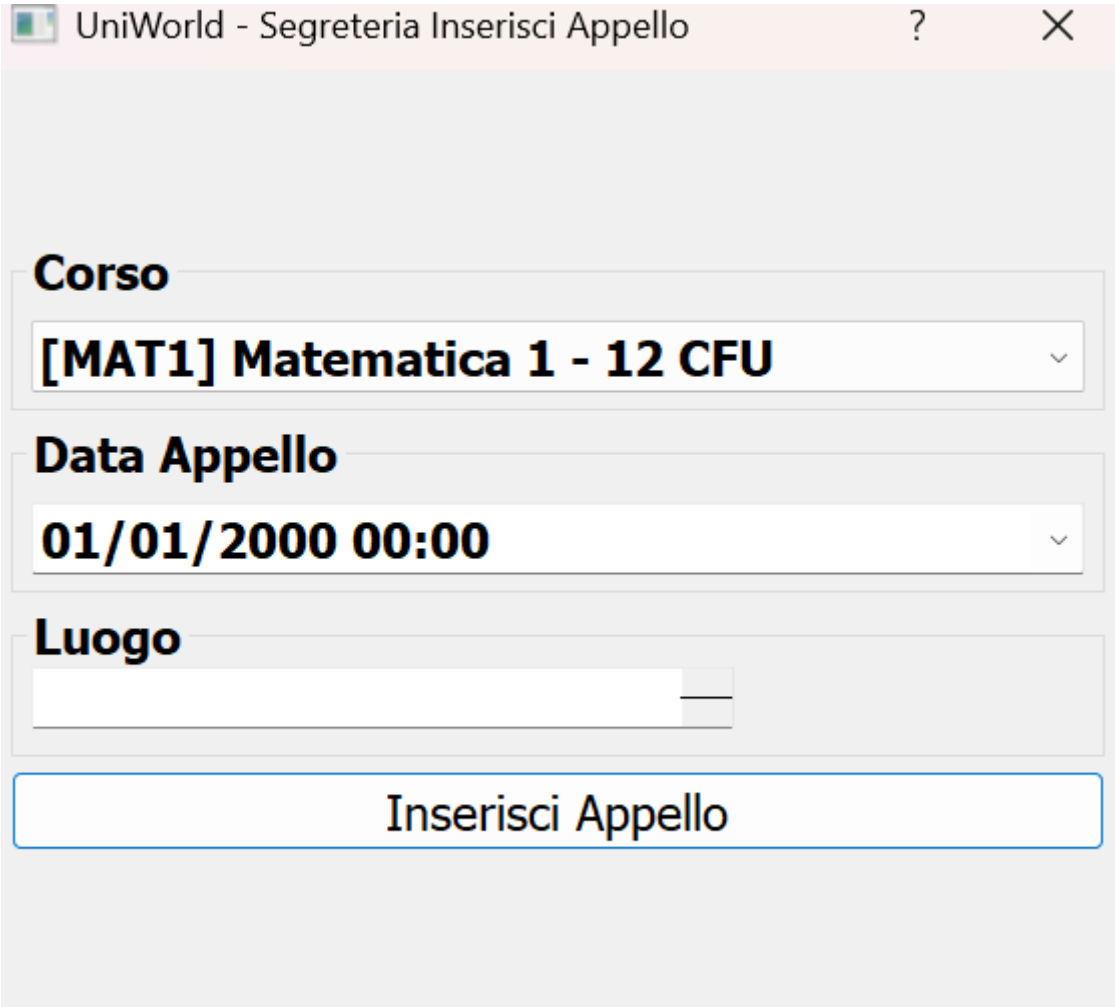

Figure 5: Inserisci Appello

La figura 5 mostra l'interfaccia inserisci appello, i campi richiesti sono

- Il corso del quale si vuole aggiungere un appello, selezionabile tramite menù a tendina, verranno mostrati i corsi aggiunti in precedenza tramite le funzione "Aggiungi Corso"
- Data e ora dell'appello
- Luogo in cui si effettuerà la prova

Una volta inserititi i dati e premuto sul pulsante" Inserisci appello", verrà mostrato un messaggio di avvenuto inserimento dell'appello se non si verifica nessun errore.

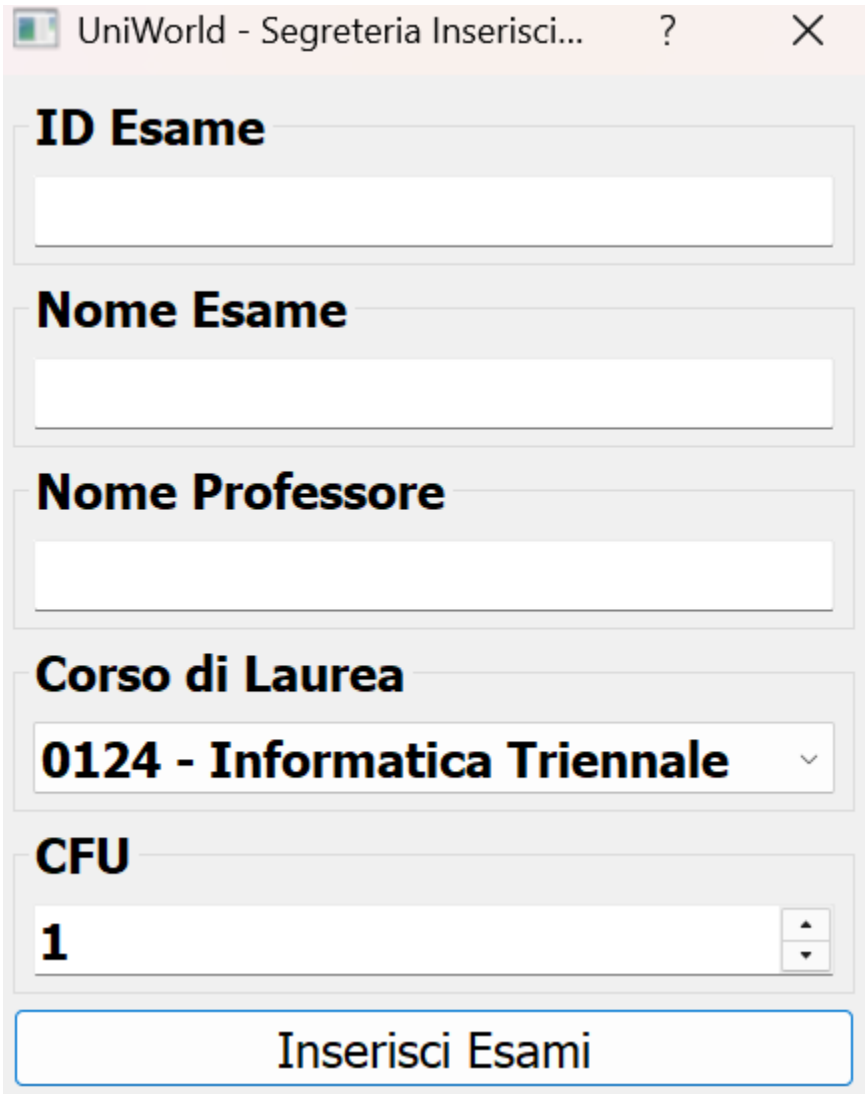

Figure 6: Aggiungi Corso

6.4.3 Aggiungi Corso

La figura 6 mostra l'interfaccia aggiungi Corso, i campi richiesti sono:

- Id esame, un codice univoco che identifica l'esame
- Nome esame, il nome del' esame
- Nome professore che tiene in corso
- Corso di laurea, selezionabile tramite il menu a tendina, si possono scegliere i corsi precedentemente inseriti
- Numero di CFU riconosciuti per quell' esame

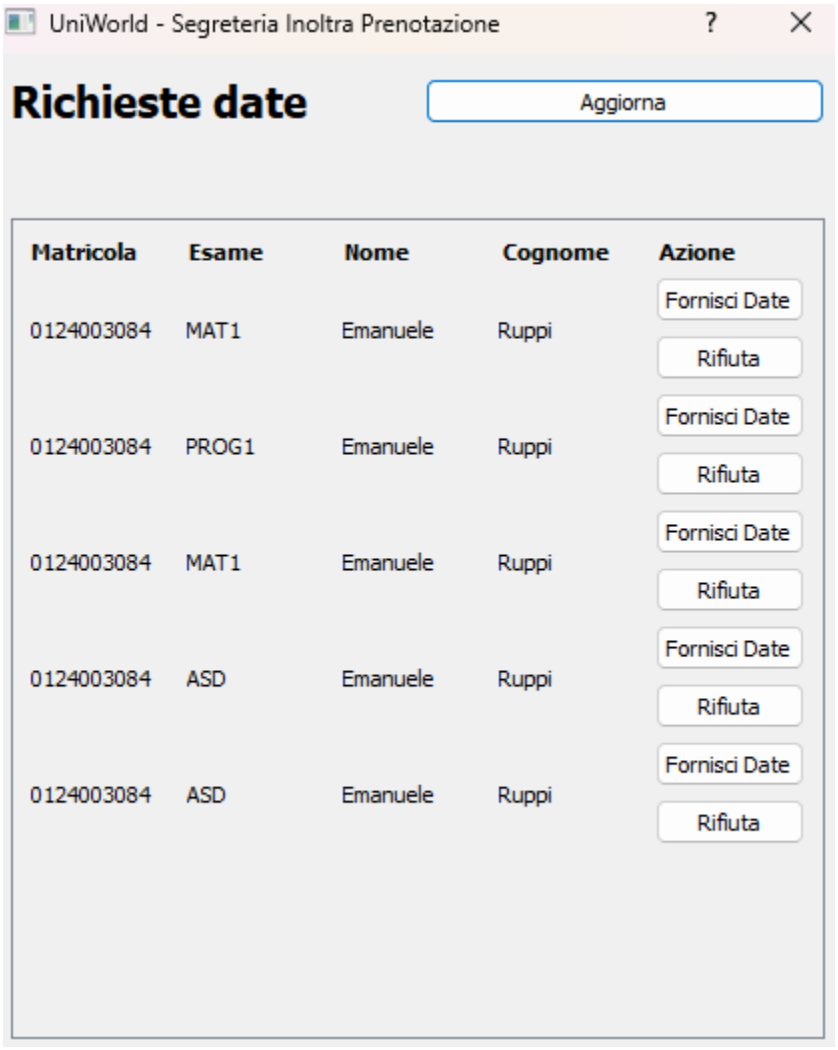

Figure 7: Richieste date

6.4.4 Gestisci Richieste Date

la figura 7 mostra l' interfaccia "richieste date" Qui sono riportate tutte le richieste di date effettuate dagli studenti, per ogni richiesta `e possibile:

- Fornire le date: Il server si occuperà di fornire le date dell'esame richiesto allo studente, che potrà visualizzarle dalla sua interfaccia
- Rifiutare: lo studente non otterrà le date e potrà visualizzare la sua proposta come "Rifiutata"

A prescindere dalla scelta, la riga con la richiesta verrà eliminata dalla tabella e verrà mostrato un messaggio in base all' esito dell' elaborazione

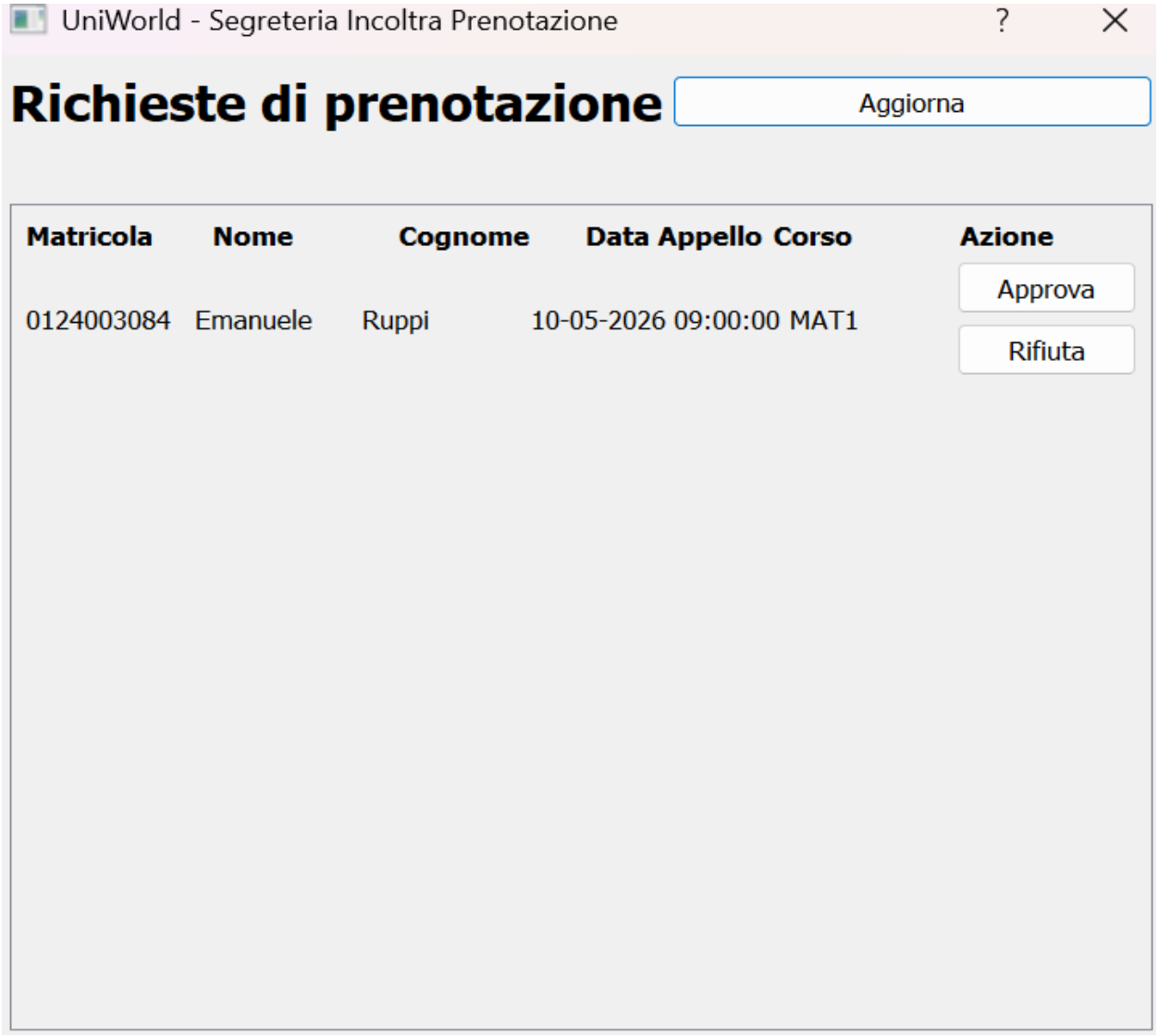

Figure 8: Richieste Prenotazioni

6.4.5 Gestisci Prenotazioni Appelli

la figura 8 mostra l' interfaccia "richieste prenotazione" Qui sono riportate tutte le richieste di prenotazione effettuate dagli studenti, per ogni richiesta `e possi-bile:

- Approvare: Lo studente visualizzerà lo stato della sua richiesta come "evasa" e quindi `e a tutti gli effetti prenotato all' esame
- Rifiutare: Lo studente vedrà dalla sue interfaccia "Respinta" e dovrà effettuare un'altra richiesta per essere accettato.

A prescindere dalla scelta, la riga con la richiesta verrà eliminata dalla tabella e verrà mostrato un messaggio in base all' esito dell' elaborazione

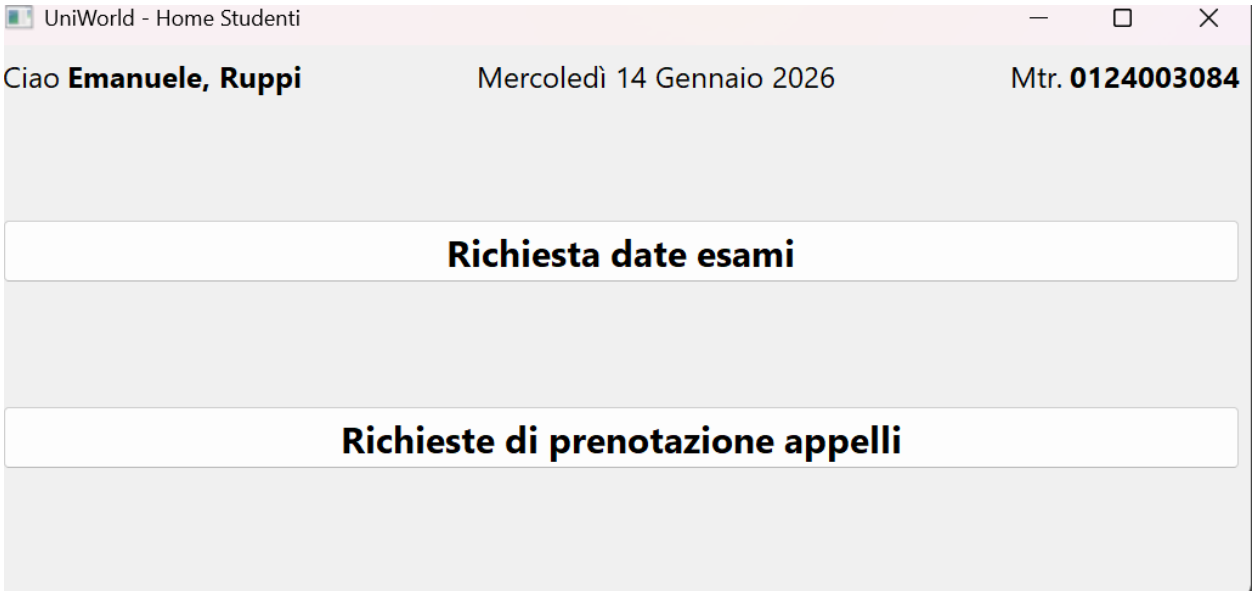

Figure 9: Enter Caption

6.5 Studente

Quando un utente studente effettua il login, viene mostrata questa interfaccia dove `e possibile accedere a tutte le funzionalità

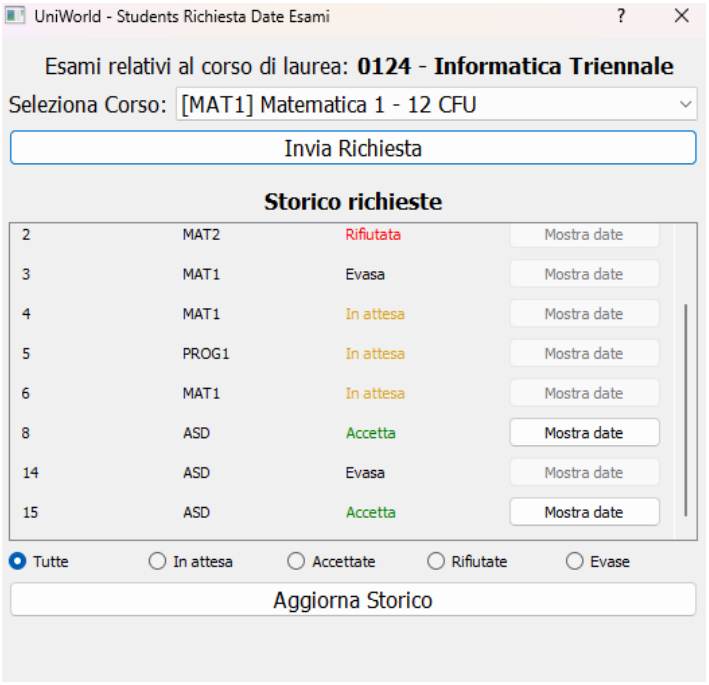

Figure 10: Richiedi date

6.5.1 Richiedi date

La figura 10 mostra l' interfaccia "richiedi date", dove `e possibile fare richiesta alla segreteria per sapere le date disponibili per un determinato esame, per fare richiesta basta:

1. Selezionare l' esame interessato tramite il menu a tendina
2. Fare click su invia richiesta

È possibile inoltre visualizzare uno storico di tutte le richieste inviate in precedenza. Le richieste possono avere i seguenti stati:

- In attesa: la segreteria ha ricevuto la richiesta ma ancora deve approvarla/respingerà
- Accettata: la segreteria ha accettato la richiesta e sono state fornite le date di appello per quell' esame
- Rifiutata: la segreteria ha respinto la richiesta
- Evasa: la segreteria ha accettato non solo la richiesta di date ma anche la prenotazione ad un appello in una determinata data

È possibile filtrare le richieste per status

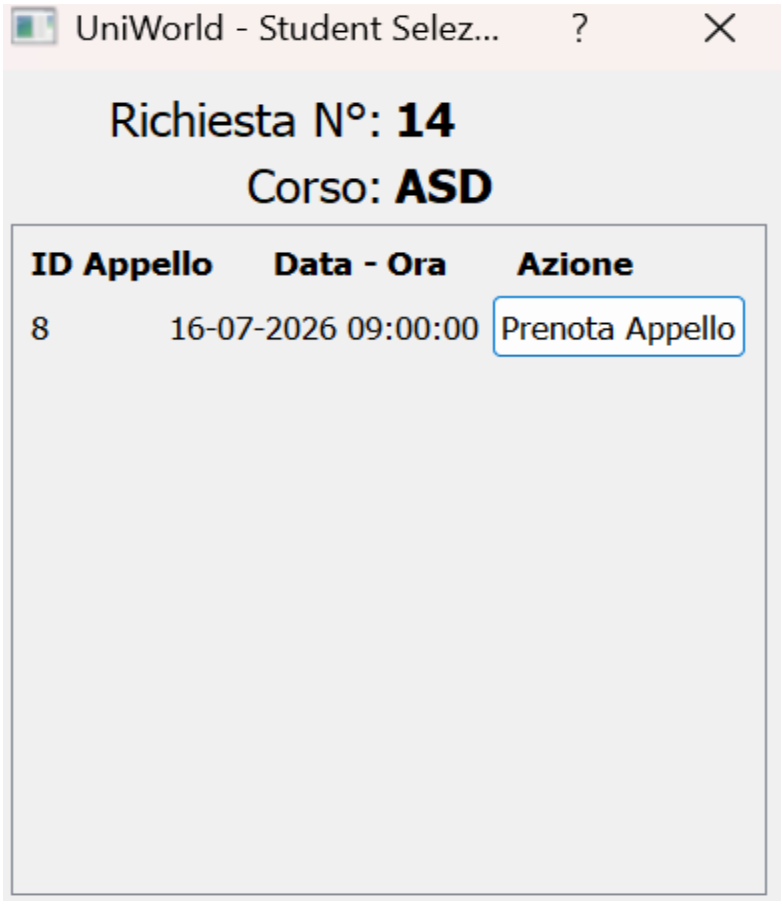

Figure 11: Richiedi Prenotazione

6.5.2 Richiedi prenotazione

In figura 11 viene mostrata l' interfaccia "Richiedi prenotazione", questa interfaccia può essere raggiunta cliccando su "mostra date" in figura 10, dopo che la segreteria ha fornito le date richieste per un determinato esame. Per prenotarsi all' appello basta cliccare su "Prenota appello" sulla riga che si desidera in base alla data dell' appello. In seguito la segreteria dovrà accettare/rifiutare la richiesta di prenotazione

Figure 12: Storico Prenotazioni

6.5.3 Visualizza Storico Prenotazioni

In figura 12 viene mostrata l' interfaccia "Storico Prenotazioni" dove `e possibile vedere lo storico delle prenotazioni passate con tanto di stato, `e anche possibile filtrare per stato

7 Database

Il sistema di persistenza dei dati è stato implementato attraverso l'utilizzo di file strutturati in formato **CSV** (Comma-Separated Values). Sebbene non si utilizzi un DBMS (Database Management System) relazionale classico, questa architettura *file-based* è stata progettata per garantire un accesso rapido e leggero alle informazioni, organizzate gerarchicamente nelle directory del progetto (esami, prenotazioni, users).
L'architettura del sistema prevede che **solamente il Server Universitario** abbia i permessi di accesso a questi file, sia in lettura che in scrittura. I Client non interagiscono mai direttamente con il database, ma inviano richieste al server che agisce da interfaccia unica.

7.1 Gestione della Concorrenza e Mutua Esclusione

Considerando la natura **parallela e concorrente** del server (che gestisce più studenti e la segreteria contemporaneamente), l'accesso ai file CSV rappresenta una **Sezione Critica**. Se più processi tentassero di scrivere sullo stesso file nello stesso istante, si verificherebbero perdite di dati o corruzione del file (*Race Condition*).
Per risolvere questo problema e garantire l'integrità dei dati, è stato implementato un meccanismo di **Mutua Esclusione** per le operazioni di scrittura. Questo viene realizzato tramite la classe Lock (o Mutex) disponibile nelle librerie di sistema di Python (threading o multiprocessing).

Il protocollo di accesso sicuro ai dati segue questi passaggi fondamentali:

1. **Istanziazione:** Si definisce un oggetto Lock condiviso.

```
locker = multiprocessing.Lock()  # o threading.Lock()
```

2. **Acquisizione (Lock):** Prima di iniziare la scrittura, il processo richiede l'accesso esclusivo. Se il file è occupato, attende.

```
locker.acquire()
```

3. **Sezione Critica:** Vengono eseguite le operazioni di scrittura sul file CSV.
4. **Rilascio (Unlock):** Al termine della scrittura, il processo rilascia immediatamente la risorsa, permettendo ad altri processi di accedervi.

```
locker.release()
```

Grazie a questo meccanismo, il sistema garantisce che le operazioni di scrittura siano atomiche e sequenziali, preservando la consistenza del database universitario.