

# Relazione sull'elaborato

Samuele Casadei, 0001097772

May 2024

# Indice

0.1	Introduzione . . . . .	1
0.2	Analisi lato server . . . . .	1
0.3	Analisi lato client . . . . .	6
0.3.1	Write thread . . . . .	7
0.3.2	Receive thread . . . . .	8
0.4	Note . . . . .	8
0.4.1	Istruzioni . . . . .	8

## 0.1 Introduzione

La tipologia di elaborato scelta è la chat client-server, per quanto riguarda la tipologia di connessione ho scelto il protocollo TCP, il server gestisce le varie connessioni con un approccio multithreaded.

## 0.2 Analisi lato server

In questo snippet di codice si può vedere che creo un server TCP tramite il metodo `socket` dell'omonimo modulo, specificando negli argomenti di creazione **AF\_INET** e **SOCK\_STREAM**, il primo indica che decido di utilizzare la famiglia di indirizzi di internet (coppia di IP e porta), il secondo invece specifica che le connessioni devono essere di natura TCP.

```
# per convenzione scelgo la porta 8080
port = 8080
# creazione server TCP
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('localhost', port))
# setto un timeout sul socket del server per far si che possa rispondere
# ad un interrupt da tastiera anche mentre è bloccato nella accept()
server.settimeout(0.5)
server.listen(1)
```

Figure 1: apertura del server

Dopo aver creato il socket, lo associo all'IP **localhost**, che per convenzione corrisponde a 127.0.0.1 e metto il socket in ascolto sulla porta 8080. In tutto questo ho impostato un timeout del socket, vedremo successivamente qual è il suo utilizzo specifico.

```
# dictionary dove ad ogni client-socket è associato un nickname
# al momento della connessione
connections = {}

print("ready to serve...")
try:
    while True:
        try:
            # accetto la connessione e chiedo il nickname
            user, address = server.accept()
            # apro un processo per gestire questo client e ritorno in ascolto
            threading.Thread(target=handle_client, args=((user, address))).start()
        except KeyboardInterrupt:
            # il key board interrupt deve passare fuori dal while true
            pass
        except socket.timeout:
            pass
    except KeyboardInterrupt:
        # attuo la close routine per avvisare ogni client che
        # il server è stato chiuso
        close_routine()
        # una volta fuori dal ciclo chiudo il server
        server.close()
        print("server shutdown.")
```

Figure 2: ciclo di gestione delle connessioni

Per tenere traccia degli utenti che si connettono durante l'attività del server tengo traccia localmente di un **dictionary di nome connections**, connections associa ad ogni connessione (socket) una stringa, la stringa coincide con il nickname che il client fornirà una volta connesso al server. Il main del server è strutturato in un blocco try che contiene il vero e proprio

svolgimento dell'attività di ascolto del server, il server opera tramite un `while True` dove accetta tutte le connessioni entranti e, per ognuna di esse apre un nuovo thread contenente la routine di gestione di un singolo client. Il blocco `try` più interno deve ignorare due eccezioni in particolare:

- **KeyboardInterrupt**: eccezione generata dalla pressione della combinazione di tasti `Ctrl + C`, che indica che il **server deve eseguire uno shutdown** e, dunque, terminare l'esecuzione; l'eccezione viene ignorata per poi essere gestita esattamente fuori dal `while True` con la routine di chiusura.
- **socket.timeout**: errore generato dal socket una volta raggiunto il timeout (in questo caso 0,5 secondi), non necessita di essere gestito, è necessario perché il metodo `accept` è un'istruzione bloccante, il timeout è necessario al server per fare **polling e controllare che non sia stato ordinato uno shutdown**.

Il thread avviato alla connessione del client esegue la funzione `handle_client` che si occupa di gestire tutte le sue richieste fino alla disconnessione:

```

# funzione per gestire il generico client, diventerà un thread
def handle_client(user, addr):
    print(f"[thread] client {addr} connected")
    user.send("Your name:".encode('utf-8'))
    # registro socket e nickname
    nickname = user.recv(1024).decode('utf-8')
    connections[user] = nickname
    # messaggio personalizzato per il client
    user.send("You joined the chat".encode('utf-8'))
    # notifica per ogni altro utente che un nuovo client è entrato
    broadcast(f"{nickname} joined the chat", user, noName = True)
    while True:
        # msg deve essere inizializzato fuori dal try, altrimenti
        # non è visibile a r.39
        msg = ""
        try:
            # ricevo ogni eventuale messaggio dal client
            msg = user.recv(1024).decode('utf-8')
            if msg == "{quit}":
                raise KeyboardInterrupt
        except:
            broadcast(f"{connections[user]} has left the chat", user, noName = True)
            del connections[user]
            # in caso di chiusura della connessione da parte del client:
            print(f"[thread] client {addr} disconnected")
            user.close()
            break
        # se non viene lanciata nessuna eccezione, invio il messaggio
        # a tutti i membri della chatroom
        broadcast(msg, user)

```

Figure 3: routine di gestione del client

La prima parte di `handle_client` opera fuori da un ciclo, si occupa di chiedere il nickname con cui il client vuole essere registrato, dopo aver ricevuto una risposta dal client decodifica il messaggio e **allaccia il socket del client al nickname in connections**.

Successivamente avvisa tutti gli utenti dell'ingresso del client (con un messaggio personalizzato per il client) tramite il metodo `broadcast`.

Dopo di che il thread si mette in ascolto di un messaggio da parte del client, se il messaggio arriva viene trasmesso a tutti gli utenti della chat (chiarirò meglio la procedura nella spiegazione del metodo `broadcast`), tuttavia c'è un'eccezione: se il messaggio inviato corrisponde esattamente con la stringa `"{quit}"` il server riconosce la **chiusura della connessione da parte del client**, dunque lancia un'eccezione che verrà poi gestita nel blocco `except`.

N.B. qualsiasi altro tipo di errore nella connessione come la disconnessione forzata da parte del client chiudendo la console porterà comunque all'esecuzione

del codice nel blocco except.

Il metodo `broadcast` riceve come argomenti un messaggio (non codificato) e due parametri opzionali: **user** che indica chi è il mittente del messaggio (esso sarà escluso dall'invio del messaggio formattato, dato che per lui è già visibile sulla console) e **noName** che (se settato a **True**) invia il messaggio senza indicare il mittente, è utile per comunicazione generali ai client, come notificare l'uscita di un utente connesso. Per il resto il corpo del ciclo itera l'insieme delle chiavi (socket) di `connections` e per ognuno controlla se si tratta del mittente e se `noName` è `True`, se lo è invia il messaggio non formattato, altrimenti mette come prefisso il nickname del mittente (in modo che gli altri utenti sappiano chi ha inviato il messaggio).

```
# funzione per trasmettere il messaggio a tutti gli utenti
def broadcast(msg, user = None, noName = False):
    # per ogni utente connesso
    for usr in connections.keys():
        if usr != user:
            if noName:
                usr.send(msg.encode('utf-8'))
            else:
                # viene inviato il messaggio formattato
                usr.send(f"{connections[user]}: {msg}".encode('utf-8'))
```

Figure 4: metodo broadcast

Nel caso in cui venisse ordinato uno shutdown del server con `Ctrl + C` da terminale, l'eccezione uscirebbe dal costrutto `try` interno al `while`, interrompendo il ciclo `while`; una volta fuori dal ciclo `while` viene invocata la funzione `close_routine`.

```
def close_routine():
    for client in connections.keys():
        client.send("{shutdown}".encode('utf-8'))
```

Figure 5: struttura della procedura di chiusura

La funzione **close\_routine** fa riferimento a `connections` per vedere i client che sono attualmente connessi al server e, per ognuno di essi, invia il messaggio `"{shutdown}"`, che il client interpreterà come la chiusura della connessione lato server, e a sua volta gestirà l'evento.

## 0.3 Analisi lato client

Il lato client inizia come un processo unico: tenta di stabilire la connessione con il server (se il server non è attivo termina, si basa sul flag **connected**), dopodiché setta un timeout sul socket del server, questo perché anche il client deve essere pronto a reagire ad un possibile interrupt da tastiera che chiede di terminare la connessione.

```
port = 8080

# creazione socket e tentativo di connessione
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connected = True
try:
    client_socket.connect(('localhost', port))
except:
    print(f"There's no server active at {'localhost'} on port {port}")
    connected = False
if connected:
    client_socket.settimeout(0.5)
    stop = False
    # la ricezione dei messaggi eseguirà su un thread separato
    r = threading.Thread(target=receive, args=((client_socket, stop)))
    r.start()
    # il processo di scrittura invece verrà gestito nel main
    write(client_socket, stop)
    print("you succesfully disconnected, thank you")
    # provo a chiudere la connessione, se non è già chiusa
    try:
        client_socket.send("{quit}".encode('utf-8'))
        client_socket.close()
    except:
        pass

time.sleep(1.5)
```

Figure 6: Enter Caption

In seguito client si biforca in due processi, di cui uno opera su un thread distinto:

- **write**: rimane costantemente in attesa di un input da parte del client, tutto ciò che viene inviato è poi trasmesso al server, che poi si occuperà di spedire il messaggio agli altri client connessi.
- **receive**: rimane costantemente in attesa di messaggi da parte del server,

ogni qual volta ne arriva uno lo mostra al client.

Entrambi i processi ciclano sulla variabile globale **stop**, nel momento in cui il client lancia un interrupt da tastiera, la variabile stop viene aggiornata e ferma entrambi i thread correttamente.

Una volta fermati i thread, il client si disconnette *gracefully* dal server mandando il messaggio ”**{quit}**”, di cui abbiamo visto precedentemente la routine di gestione da parte del server.

### 0.3.1 Write thread

Write è una funzione molto semplice, essa viene eseguita direttamente nel main, si basa sul riferimento della variabile **stop**, che viene passata per parametro con il nome di **stop\_flag**, su cui poi andrà a iterare con un while.

```
# metodo eseguito alla creazione della connessione, permette al client
# di scrivere messaggi nella chatroom, mantiene anche il controllo della
# ricezione dei messaggi, in caso venisse interrotto, è write a gestire
# la procedura di chiusura lato client
def write(client, stop_flag):
    while not stop_flag:
        try:
            msg = input()
            client.send(msg.encode('utf-8'))
        except socket.timeout:
            pass
        except KeyboardInterrupt:
            # setting the condition for the read thread to stop
            stop_flag = True
```

Figure 7: funzione write

Il ciclo si basa, sostanzialmente, sull’attendere un input da parte del client tramite la funzione **input**, dopodiché codifica il messaggio in input e lo invia al server, tutta questa operazione è *wrappata* in un blocco **try/except**, dato che l’invio da parte del client di un input da tastiera deve poter svegliare il write dal suo stato di attesa e portarlo in un blocco di gestione dell’eccezione dove andrà ad aggiornare la variabile **stop\_flag**.

Dato che stop\_flag è passata by reference, aggiornare stop\_flag dentro write comporta anche l’aggiornamento della variabile **stop** esterna alla funzione.



### 0.3.2 Receive thread

Come è possibile vedere dall'immagine 6, la funzione write viene lanciata su un **thread separato** dal main, questo per poter eseguire **parallelamente a write**.

```
# metodo che viene eseguito in un thread per mostrare i messaggi degli
# altri utenti in tempo reale
def receive(client, stop_flag):
    while not stop_flag:
        try:
            # stampo ogni messaggio ricevuto dal server
            msg = client.recv(1024).decode('utf-8')
            if msg == "{shutdown}":
                client.close()
                print("Lost connection with the server")
                stop_flag = True
            else:
                print(msg)
        except socket.timeout:
            # intervallo di polling con cui controllo se
            # il client vuole interrompere la connessione
            pass
        except:
            break
```

Figure 8: funzione receive

Anche receive cicla su stop\_flag, nel suo blocco più interno opera in modo diametralmente opposto a write: si mette in ascolto sulla connessione con il server tramite il metodo **socket.recv**, non appena riceve un messaggio, controlla che non si sia verificato uno **shutdown per keyboard interrupt** del server (in tal caso **modifica stop\_flag per far fermare anche write**), se non è così, stampa il messaggio sullo standard output.

## 0.4 Note

Il programma è stato realizzato precedentemente alla lezione 7 di laboratorio, dunque non ho realizzato un'interfaccia grafica adeguata per la chat.

### 0.4.1 Istruzioni

Si prega di eseguire i programmi su shell cmd, si avvia prima il server, dopo di che si possono avviare un numero a piacimento di client.

Una volta avviato il programma client, si inserisce lo username che si preferisce utilizzare, solo dopo si può iniziare a scrivere.

Nel frattempo il server mostrerà sullo stdout le connessioni attualmente attive e gli utenti che si disconnettono.

Se si decide di terminare la connessione lato client basta premere la combinazione di tasti Ctrl + C o semplicemente chiudere la shell. Se invece si desidera terminare la connessione lato server è indispensabile premere Ctrl + C.

Quando il server viene chiuso i client riceveranno un messaggio che avverte della connessione persa, la shell da quel momento in poi non sarà più interattiva e potrà essere chiusa sia manualmente che con Ctrl + C.