RELAZIONE ELABORATO PROGRAMMAZIONE DI RETI

Traccia: Chat Client-Server

Alessandro Ronchi

Organizzazione generale del sistema

Il sistema si compone di due script Python che implementano le due entità principali all'interno di un sistema di chat Client-Server:

- Il Server, che ha il compito di gestire la connessione degli utenti alla chat e lo scambio di messaggi al suo interno. In particolare, il Server deve gestire l'invio di messaggi alla chat da parte di ogni utente e deve consentire ad ogni utente di visualizzare ogni messaggio che viene inviato all'interno della chat.
- Il Client, che costituisce l'applicazione che consente ad ogni utente di accedere alla chatroom e scambiare messaggi con gli altri utenti connessi al Server.

La comunicazione tra Client e Server prevede l'utilizzo del protocollo TCP, in quanto richiede una connessione affidabile che garantisca il corretto invio di ogni messaggio che viene scambiato nella chat.

A questo scopo, entrambi gli script fanno uso del modulo "socket", che fornisce un'implementazione di una porta di comunicazione che consente di scambiare messaggi tra due dispositivi secondo uno specifico protocollo, nel nostro caso TCP.

Le sezioni seguenti illustrano nel dettaglio il funzionamento dei due script.

Server

Il Server deve poter gestire la connessione alla chat di client multipli che possono simultaneamente inviare dei messaggi all'interno della chat. Ogni messaggio ricevuto dal Server deve inoltre poter essere visualizzato da tutti gli utenti connessi.

Per questo motivo, il Server deve stabilire un canale di comunicazione separato per ogni client che decide di connettersi alla chat. La gestione di connessioni multiple è possibile grazie all'operazione di *fork*: per ogni connessione in entrata, il Server crea una socket fittizia che si occupa di gestire le richieste del singolo client. La socket principale del Server, invece, ha come unico scopo quello di intercettare le connessioni di nuovi client per poi delegarle ad una socket fittizia. La comunicazione con i vari client deve avvenire in modo concorrente, e per questo motivo ogni operazione di fork genera un thread separato che si occupa di gestire lo scambio di messaggi con un singolo client. Per gestire i thread è stato importato il modulo "threading" e in particolare la classe "Thread" definita al suo interno.

Avvio del server

Innanzitutto, vengono definite le strutture dati necessarie alla gestione dei client. Il dizionario "addresses" serve per associare ogni socket di un client connesso al proprio indirizzo IP. La lista "names" consente di tenere traccia dei nomi scelti dai client connessi, in modo da evitare ripetizioni.

```
# Dizionario usato per tenere traccia dei client connessi al server.
addresses = {}
# Lista usata per tenere traccia dei nomi scelti, in modo da evitare ripetizioni.
names = []
```

Successivamente, viene stabilito l'indirizzo IP del server e il numero di porta su cui è attivo il servizio e viene creata la socket principale del Server, che verrà utilizzata per accettare le richieste di connessione da parte dei client.

Il parametro SOCK_STREAM permette di specificare che la connessione deve avvenire secondo il protocollo TCP.

```
HOST = ''
PORT = 53000
BUFFSIZE = 1024
ADDR = (HOST, PORT)

# Crea la socket del server selezionando il protocollo TCP.
SERVER = socket(AF_INET, SOCK_STREAM)
SERVER.bind(ADDR)
```

Nel main, viene impostata sulle richieste al server una coda di lunghezza 5, grazie all'istruzione listen(), e viene avviato il thread principale di gestione delle connessioni in entrata.

```
if __name__ == "__main__":
    SERVER.listen(5)
    print("Waiting for connections...")
    ACCEPT_THREAD = Thread(target=accept_connections)
    ACCEPT_THREAD.start()
    wait_signal(1)
```

Dopo aver avviato il thread, il main entra nella funzione wait_signal, che consiste in un loop infinito in cui, dopo un certo intervallo di tempo specificato come parametro di ingresso, il main controlla se è stato ricevuto un segnale di interrupt da tastiera per la chiusura del server. Infatti, vogliamo che il server si chiuda se da terminale viene digitata la sequenza CTRL+C. Ulteriori dettagli sul meccanismo di chiusura del server sono specificati più avanti.

La funzione wait_signal() è definita come segue:

```
def wait_signal(wait_interval):
    while ACCEPT_THREAD.is_alive():
        time.sleep(wait_interval)
```

La condizione di uscita dal loop è la terminazione del thread di accettazione delle richieste.

Questo loop di "polling" è necessario in quanto, su Windows, non è possibile sbloccare il main con un interrupt da tastiera se la sua

esecuzione è ferma in un'istruzione bloccante come join(), accept() o recv().

Accettazione delle connessioni

La ricezione delle connessioni da parte di nuovi server è gestita dalla funzione accept_connections, che non fa altro che mettersi in attesa di una nuova connessione grazie all'istruzione accept().

Quando il server riceve una nuova connessione, aggiunge il socket appena creato all'insieme di client da gestire e avvia il thread di gestione del singolo client.

Il costrutto try attorno all'istruzione accept() serve ad intercettare l'eccezione generata nel caso in cui il server chiuda la socket principale al termine della propria attività. Se si verifica questa eccezione, il thread di accettazione delle connessioni viene terminato.

```
def accept_connections():
    while True:
        # Se il server chiude la connessione, cattura l'eccezione corrispondente
        try:
            client, client_address = SERVER.accept()
        except OSError:
            break

        # Gestisce la connessione in entrata avviando il thread separato.
        print("%s:%s has connected to the server." % client_address)
        addresses[client] = client_address
        Thread(target=handle_client, args=(client,)).start()
```

Gestione dei client

La gestione della comunicazione con il singolo client è gestita attraverso la funzione handle_client().

Questa funzione fa uso di varie funzioni di utility, tra cui la funzione broadcast(), che si occupa di inviare uno specifico messaggio a tutti i client connessi:

```
def broadcast(msg, prefix=""):
    for client in addresses:
        client.send(bytes(prefix, "utf8")+msg)
```

I parametri di ingresso della broadcast() sono il messaggio da inviare e un prefisso che contiene il nome dell'utente che ha inviato il messaggio.

Per prima cosa, la handle_client() invia un messaggio all'utente, chiedendogli di inserire il nome con cui verrà identificato all'interno della chat. In questa porzione di codice, la funzione svolge un controllo sui nomi disponibili, rifiutando quelli che sono già stati salvati all'interno della lista "names". Il controllo è svolto dalla funzione valid_name(). Se il nome è già stato preso, il server chiede all'utente di inserirne un altro.

```
def handle_client(client):
    # Richiede al client l'inserimento del nome.
    # Per evitare nomi duplicati, verifica la validità del nome inserito,
    # ed eventualmente ne richiede nuovamente l'inserimento.
    client.send(bytes("Salve! Digita il tuo Nome seguito dal tasto Invio!", "utf8"))
    try:
        name = client.recv(BUFFSIZE).decode("utf8")
        while not valid_name(name):
            invalid_name = "Il nome scelto è già assegnato, inserire un nome diverso:"
            client.send(bytes(invalid_name, "utf8"))
            name = client.recv(BUFFSIZE).decode("utf8")
        # Se l'utente esce durante la scelta del nome, chiude la connessione.
        if name == "{quit}":
            close_client_connection(client)
            return 0

        names.append(name)
        greetings = 'Benvenuto %s! Se vuoi lasciare la Chat, scrivi {quit} per uscire.' % name
        client.send(bytes(greetings, "utf8"))
        # Avvisa tutti gli utenti connessi che un nuovo utente si è unito alla chat
        msg = "%s si è unito alla chat!" % name
        broadcast(bytes(msg, "utf8"))
```

Se il nome inserito è valido, il server invia un messaggio di benvenuto all'utente e avvisa tutti gli utenti connessi che un nuovo utente si è unito alla chat. Se, invece, la stringa digitata dall'utente equivale al messaggio di uscita, cioè "{quit}", il server chiude la connessione con il client grazie alla funzione close_client_connection(), che chiude la socket del client e la rimuove dal dizionario contenente le connessioni attive:

```
def close_client_connection(client):
    client.close()
    print("%s:%s has left the server." % addresses[client])
    del addresses[client]
```

Dopo la scelta del nome, inizia la fase di gestione dei messaggi inviati dal client. Ogni messaggio ricevuto attraverso il socket viene

inviato in broadcast a tutti gli utenti connessi. Il messaggio viene corredato di un prefisso che specifica il nome dell'utente che l'ha inviato.

```
while True:
    msg = client.recv(BUFFSIZE)
    if msg != bytes("{quit}", "utf8"):
        broadcast(msg, name+": ")
    else:
        close_client_connection(client)
        # Avvisa tutti gli utenti connessi che il client ha lasciato la chat.
        broadcast(bytes("%s ha abbandonato la Chat." % name, "utf8"))
        names.remove(name)
        break
```

Anche in questo caso, la ricezione del messaggio di uscita fa sì che il server chiuda la connessione con il client. Dopo aver chiuso la connessione, il server avvisa tutti gli utenti che il client corrente ha lasciato la chat, dopodiché rende nuovamente disponibile il nome che era associato a tale utente.

Anche in questa funzione si ha un costrutto try, che come nel caso della accept_connections(), intercetta l'eccezione che si verifica quando il server decide di chiudere la connessione con il client. Questa eccezione viene gestita facendo terminare il loop di gestione dei messaggi in entrata.

Chiusura del Server

L'esecuzione del server deve poter essere interrotta quando da terminale si digita la sequenza CTRL+C. La pressione di questi due tasti invia allo script in esecuzione il messaggio SIGINT. Per poter intercettare questo messaggio, è necessario servirsi del pacchetto "signal" che a sua volta contiene la funzione "signal" che ci consente di definire il comportamento del programma alla ricezione di un certo segnale.

```
#Gestisce la chiusura del server quando viene premuto CTRL+C.
signal.signal(signal.SIGINT, close_server)
```

Questa istruzione imposta come procedura di risposta al segnale la funzione close_server, che si occupa di chiudere tutte le risorse allocate dal Server e provocare la sua terminazione.

```python
def close_server(signal, frame):
    print("CTRL+C pressed. Closing the server...")
    # Invia a tutti i client il messaggio di fine connessione
    msg = "{end_conn}"
    broadcast(bytes(msg, "utf8"))
    # Dealloca le risorse occupate dalle socket, provocando anche
    for client in addresses:
        client.close()
    SERVER.close()
```

Questa procedura invia a tutti i client connessi il messaggio “{end_conn}”, avvisandoli che il server si sta disconnettendo, e successivamente chiude tutte le socket aperte, compresa quella principale.

La chiusura della socket SERVER provoca la terminazione del thread di gestione delle connessioni in entrata, e di conseguenza la fine dell’esecuzione dello script.

Client

L’applicazione client ha il compito di gestire la connessione al server lato utente. Questo software deve consentire all’utente di connettersi ad una chat gestita da uno specifico server ed inviare messaggi al suo interno.

Per rendere più agevole l’utilizzo della chat, il client utilizza un’apposita GUI che fornisce le funzionalità richieste. Per la realizzazione della GUI è stato utilizzato il modulo “tkinter”.

```python
import tkinter as tkt
```

Come per il Server, anche in questo caso è necessario gestire due flussi di esecuzione separati: la ricezione dei messaggi e la gestione della GUI che prevede anche l’invio dei messaggi al server. Anche in questo caso, quindi, viene utilizzato il modulo “threading”.

Avvio applicazione

Per prima cosa, l'applicazione chiede all'utente di inserire, tramite riga di comando, l'indirizzo IP e il numero di porta a cui connettersi. Se l'utente non inserisce nessun numero di porta, viene utilizzato un numero di porta predefinito, cioè il 53000. Una volta che l'utente ha digitato le informazioni richieste, viene stabilita la connessione con il server specificato, creando un'apposita socket.

```
# CONNESSIONE AL SERVER:

HOST = input('Inserire il Server host: ')
PORT = input('Inserire la porta del server host: ')
if not PORT:
    PORT = 53000
else:
    PORT = int(PORT)

BUFSIZ = 1024
ADDR = (HOST, PORT)

client_socket = socket(AF_INET, SOCK_STREAM)
client_socket.connect(ADDR)
```

In seguito, si ha la creazione di una semplice GUI costituita da un'unica finestra contenente un riquadro con la lista di messaggi nella parte superiore e un campo di inserimento dei caratteri nella parte inferiore. Insieme alla casella di inserimento vi è un bottone "Invio", che consente di inviare il messaggio digitato nella casella. L'invio dei messaggi è possibile anche premendo il tasto [Invio] sulla tastiera, e l'associazione della pressione del tasto alla funzione di invio del messaggio è effettuato dall'istruzione entry_field.bind().

Per la gestione dei messaggi digitati dall'utente viene creata una variabile stringa chiamata my_msg, nella quale è immagazzinato il contenuto della casella di input, e quindi il messaggio che l'utente intende inviare alla chat.

```
# CREAZIONE GUI:

window = tkt.Tk()
window.title("Chat")

# Crea il Frame per contenere i messaggi
messages_frame = tkt.Frame(window)
# I messaggi dell'utente vengono inseriti in una variabile stringa.
my_msg = tkt.StringVar()
my_msg.set("Scrivi qui i tuoi messaggi.")
# Crea una scrollbar per navigare tra i messaggi precedenti.
scrollbar = tkt.Scrollbar(messages_frame)

# La parte seguente contiene i messaggi.
msg_list = tkt.Listbox(messages_frame, height=30, width=100, yscrollcommand=scrollbar.set)
scrollbar.pack(side=tkt.RIGHT, fill=tkt.Y)
msg_list.pack(side=tkt.LEFT, fill=tkt.BOTH)
msg_list.pack()
messages_frame.pack()

# Crea il campo di input, associandolo alla variabile stringa che contiene il messaggio.
entry_field = tkt.Entry(window, textvariable=my_msg, width=50)
# Imposta il tasto Invio o Return come comando per l'invio dei messaggi.
entry_field.bind("<Return>", send)
entry_field.pack()

# Crea il tasto che consente l'invio di messaggi dalla GUI.
send_button = tkt.Button(window, text="Invio", command=send)
send_button.pack()

# Specifichiamo la procedura che dev'essere svolta alla chiusura della finestra.
window.protocol("WM_DELETE_WINDOW", on_closing)
```

L’ultima riga di creazione della GUI specifica la procedura che dev’essere svolta quando l’utente decide di chiudere la finestra. In particolare, la chiusura della finestra provoca l’esecuzione della funzione on_closing:

```
def on_closing(event=None):
    my_msg.set("{quit}")
    send()
```

Questa funzione invia al server il messaggio “{quit}”, comunicando l’intenzione di uscire dalla chat. In seguito, la funzione send(), responsabile dell’invio dei messaggi, si occuperà della chiusura vera e propria della finestra.

Infine, si ha l’avvio dell’applicazione vera e propria: viene avviato il thread che si occupa di gestire la ricezione dei messaggi e viene chiamata la funzione mainloop() di tkinter, che dà inizio al flusso di gestione della GUI.

```
# AVVIO DELL'APPLICAZIONE:

# Avvia il thread di gestione della ricezione dei messaggi.
receive_thread = Thread(target=receive)
receive_thread.start()
# Avvia l'esecuzione della Finestra Chat.
tkt.mainloop()
```

Ricezione dei messaggi

La funzione receive() gestisce la ricezione dei messaggi inviati dal server. Ogni messaggio ricevuto viene aggiunto alla lista presente nella GUI, in modo che l'utente possa visualizzare ogni messaggio inviato nella chat. La ricezione del messaggio "{end_conn}" da parte del server indica che quest'ultimo sta per chiudere la connessione. In tal caso, l'applicazione avvisa l'utente con un apposito messaggio sulla GUI e chiude l'applicazione dopo 5 secondi.

Il costrutto try serve a terminare l'esecuzione del thread nel momento in cui il client chiude la socket di comunicazione.

```
def receive():
    while True:
        try:
            msg = client_socket.recv(BUFSIZ).decode("utf8")
            # Se viene ricevuto il messaggio di fine connessione, chiude la socket ed esce dall'
            if msg == "{end_conn}":
                msg = SERVER_DISCONNECTED
                msg_list.insert(tkt.END, msg)
                msg = "La finestra si chiuderà tra 5 secondi."
                msg_list.insert(tkt.END, msg)
                time.sleep(5)
                close_app()
            # Aggiunge il messaggio ricevuto alla lista dei messaggi presenti a schermo.
            msg_list.insert(tkt.END, msg)
            # Se il client chiude il socket, uscendo dalla chat, viene generata un'eccezione
            # che provoca la chiusura di questo thread.
        except OSError:
            break
```

Invio dei messaggi

I messaggi vengono inviati dalla funzione send, che viene invocata ogni volta che l'utente digita il tasto "Invio" sulla GUI o sulla tastiera. Questa funzione legge la stringa contenente il testo digitato dall'utente e gestisce il messaggio da inviare. Se l'utente invia il messaggio "{quit}", il client viene disconnesso dalla chat, e l'applicazione viene chiusa. Se, invece, si verifica un errore di

connessione con il server, l'utente viene avvisato con un apposito messaggio sull'interfaccia grafica.

```
def send(event=None):
    #Legge il messaggio dalla casella di inserimento e la libera.
    msg = my_msg.get()
    my_msg.set("")
    # Prova ad inviare il messaggio al server.
    # Se si verifica un errore di connessione, avvisa l'utente con
    try:
        client_socket.send(bytes(msg, "utf8"))
    except OSError:
        msg_list.insert(tkt.END, SERVER_ERROR)
        msg_list.insert(tkt.END, RESTART_APPLICATION)
    if msg == "{quit}":
        close_app()
```

Chiusura dell'applicazione

Quando viene invocata la funzione close_app(), il client chiude la socket, terminando anche il thread di ricezione dei messaggi, e termina la GUI con il metodo destroy(), che interrompe il thread che controlla la finestra di tkinter.

```
def close_app():
    client_socket.close()
    window.destroy()
```

Guida d'uso

Per avviare il sistema, è innanzitutto necessario mettere in esecuzione lo script del server sul dispositivo che dovrà fungere da host. Per farlo, basta aprire un terminale di comandi con un ambiente python installato e digitare il comando:

"python chat_server.py"

In alternativa, è possibile avviare il server dall'applicazione eseguibile facendo doppio click su chat_server.exe.

L'esecuzione del server può essere interrotta in ogni momento digitando la sequenza CTRL+C. In questo modo, il server si occuperà di chiudere tutte le connessioni attive con eventuali client.

Per connettersi alla chat come client, è necessario avviare lo script chat_client.py in modo analogo a quanto illustrato per il server, oppure avviare l'esecuzione dell'eseguibile chat_client.exe.

All'avvio dell'applicazione, verrà chiesto l'inserimento dell'indirizzo IP del server a cui ci si vuole connettere. Se il server è in esecuzione sulla stessa macchina del client, è necessario inserire l'indirizzo di loopback, cioè "127.0.0.1". Altrimenti, se i due dispositivi sono sulla stessa LAN è possibile utilizzare il comando ipconfig (Windows) o ifconfig (Linux) da terminale per trovare l'IP del server.

Insieme all'IP verrà chiesto il numero di porta. Il numero utilizzato di default dal server è 53000. Se si preme "Invio" senza digitare alcun numero, l'applicazione imposterà in automatico il numero corretto.

Dopo aver inserito queste informazioni, si aprirà la finestra dell'applicazione. Innanzitutto, è necessario inserire il proprio nome all'interno della casella di input e premere "Invio" (sulla tastiera o sul pulsante nella GUI), dopodiché sarà possibile interagire all'interno della chat. È possibile uscire dalla chat in ogni momento chiudendo la finestra con la X in alto a destra oppure inviando il messaggio "{quit}". Il messaggio "{quit}" consente di uscire dall'applicazione anche durante la fase di inserimento del nome.

In caso di errore o disconnessione del server, verrà ricevuto un apposito messaggio di avviso.

Considerazioni aggiuntive

Se ci si prova a connettere ad un altro dispositivo, la connessione può venire bloccata da eventuali Firewall nel sistema operativo. In quel caso, è necessario abilitare l'applicazione alla comunicazione sulla rete all'interno delle impostazioni di sistema.