Elaborato di Reti

Programma client e server per la messaggistica. I client inviano messaggi al server, il server rimane in ascolto e appena riceve un messaggio da un client, invia quel messaggio agli altri client. si basa su protocollo di livello 4 (transport) TCP quindi Connection oriented, viene prima instaurata una connessione stabile tra il client e il server così da garantire una connessione sicura senza perdita di informazioni.

Codice Client

Inizializzazione

librerie utilizzate

```
from socket import AF_INET, socket, SOCK_STREAM
from threading import Thread
import tkinter as tkt
```

Componenti

- **HOST**: indico l'indirizzo del server, 127.0.0.1 (indirizzo id loopback o localhost)
- **PORTA**: indico la porta in cui è in ascolto il server, 53000
- **BUFSIZ**: dimensione del buffer size per l'invio e la ricezione dei messaggi
- **ADDR**: socket del server con cui vogliamo connetterci
- **mySocket**: creo il socket del client
- **receiveThread**: thread per la l'ascolto dei messaggi che arrivano dal server

funzionamento

Quando chiami **mySocket.connect(ADDR)**, l'oggetto socket avvia un processo di three ways handshake per stabilire la connessione con il server. Quando la connessione viene stabilita, viene fatto partire il thread per l'ascolto (receiveThread) e poi viene aperta la finestra della chat.

Gestione eccezioni

viene gestita l'eccezione **ConnectionRefusedError:**
Se il server non risponde, e quindi è offline, viene stampato un messaggio che dice che il server non è in linea a quell'indirizzo a quella porta (127.0.0.1:53000), quindi non è stato possibile instaurare una connessione, in seguito viene chiudo il programma.

codice

```
HOST = '127.0.0.1'
PORT = 53000

BUFSIZ = 1024
ADDR = (HOST, PORT)
```

```
mySocket = socket(AF_INET, SOCK_STREAM)
try:
    mySocket.connect(ADDR)
    receiveThread = Thread(target=receive)
    receiveThread.start()
    # Avvia l'esecuzione della Finestra Chat.
    tkt.mainloop()
except ConnectionRefusedError:
    print("Server non in linea, impossibile stabilire la connessione")
    mySocket.close()
```

Funzioni principali

Ci sono due funzioni che un client per la messaggistica deve saper fare ovvero:

- invio del messaggio
- ricezione dei messaggi che riceverà dal server gestito da un thread

Send()

funzionamento

Quando si preme invio viene preso il valore dentro la messagebox e messo in una variabile di tipo stringa msg poi viene rimesso il valore della message box vuoto. invio il messaggio tramite alla funzione send() della libreria socket. ci permette di inviare il messaggio al server con cui ci siamo connessi con il nostro socket nella fase di **inizializzazione** del client.

Gestione eccezioni

Viene gestita l'eccezione di **ConnectionResetError** ovvero quando il server si disconnette. Quando si disconnette il server, appena viene mandato un messaggio si accorge che il server non è più in ascolto (quindi online) e viene chiuso il socket e la finestra.

codice

```
def send(event=None):

    try:
        # prende il valore dalla message box
        msg = msgVar.get()
        # libera la message box
        msgVar.set("")
        # invia il messaggio sul socket
            mySocket.send(bytes(msg, "utf8"))           #se viene inviato il
messaggio che serve per uscire dalla chat {quit} chiudo il socket e la finestra
        if msg == "{quit}":
            mySocket.close()
            window.quit()
            #gestisco l'eccezione nel caso il server viene chiuso forzatamente
    except ConnectionResetError:
```

```
        print("in server si è chiuso forzatamente")
        mySocket.close()
        window.quit()
```

Receive()

funzionamento

essendo una funzione eseguita da un thread, viene fatto un loop con **while True**. Il thread rimane in ascolto dei messaggi che arrivano sul socket con la funzione **mySocket.recv(BUFSIZ).decode("utf8")**. Appena arriva un pacchetto dal server, viene decodificato in utf-8 (standard per la codifica dei caratteri utilizzato in questo progetto) e salvato nella variabile **msg**. la variabile poi viene inserita all'interno della lista visibile sulla finestra

Gestione eccezioni

in caso di errori dovuti dal sistema operativo o semplicemente perché il client ha abbandonato la chat, viene fatto un **break** e il thread viene chiuso

codice

```
def receive():
    while True:
        try:
            #quando viene chiamata la funzione receive, si mette in ascolto
dei messaggi che
            #arrivano sul socket
            msg = mySocket.recv(BUFSIZ).decode("utf8")
            #visualizziamo l'elenco dei messaggi sullo schermo
            #e facciamo in modo che il cursore sia visibile al termine degli
stessi
            msgList.insert(tkt.END, msg)
            # Nel caso di errore e' probabile che il client abbia abbandonato
la chat.
        except OSError:
            break
```

Funzioni secondarie

on_closing()

funzionamento

funzione utilizzata quando si vuole chiudere il programma. Viene chiamata quando si clicca la **x** sulla finestra o il pulsante “**Quit**”

codice

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

clearChat()

funzionamento

funzione utilizzata per pulire la chat. viene cancellato tutto quello che c’è all’interno della lista dei messaggi nella finestra. viene chiamata quando viene premuto il pulsante “**Pulisci**”

codice

```
def clearChat():
    msgList.delete(0,'end')
```

Creazione GUI

interfaccia utente grafica (**GUI**) semplice per una chat utilizzando la libreria Tkinter. Le funzionalità della GUI sono:

- Visualizzare i messaggi di chat
- Inviare nuovi messaggi
- Cancellare la chat
- Uscire dalla chat

Componenti

- **window**: La finestra principale della GUI, intitolata "CHAT ROOM".
- **msgFrame**: Un frame dedicato per contenere i messaggi di chat.
- **msgList**: Un'area di scorrimento per visualizzare la cronologia dei messaggi di chat.
- **msgBox**: Un campo di testo per inserire nuovi messaggi.
- **msgSendButton**: Un pulsante per inviare il messaggio inserito nel campo di input.
- **msgClearButton**: Un pulsante per cancellare tutti i messaggi di chat.
- **msgQuitButton**: Un pulsante per chiudere la GUI e terminare il programma.

Funzionamento

- *Invio messaggi:* scrivo il messaggio nella message box e invio il messaggio al server in due modi:

    - premendo **INVIO**
    - premendo il bottone “**Invio**”
- *Visualizzazione messaggi:* i messaggi vengono visualizzati nella lista di elementi. Vengono inseriti dal thread.
- *Cancellazione chat:* pulisco la lista di elementi così da eliminare messaggi vecchi. si effettua premendo il bottone “Pulisci”
- *Uscita dalla chat:* chiusura della connessione e della finestra. tre modi per chiudere il programma:
    - premendo la **X** della finestra
    - scrivendo **{quit}** nella chat
    - premendo il bottone “**Quit**”

codice

```
"""Creo la mia GUI utilizzando la libreria Tkinter"""
window = tkt.Tk()
window.title("CHAT ROOM")

#creiamo il Frame per contenere i messaggi
msgFrame = tkt.Frame(window)
#creiamo una variabile di tipo stringa per i messaggi da inviare.
msgVar = tkt.StringVar()
#creiamo una scrollbar per navigare tra i messaggi precedenti.
msgScrollbar = tkt.Scrollbar(msgFrame)

# La parte seguente contiene i messaggi.
msgList        =        tkt.Listbox(msgFrame,        height=15,        width=50,
yscrollcommand=msgScrollbar.set)
msgScrollbar.pack(side=tkt.RIGHT, fill=tkt.Y)
msgList.pack(side=tkt.LEFT, fill=tkt.BOTH)
msgList.pack()
msgFrame.pack()

#Creiamo il campo di input e lo associamo alla variabile stringa
msgBox = tkt.Entry(window, textvariable=msgVar)
# leghiamo la funzione send al tasto Return
msgBox.bind("<Return>", send)

msgBox.pack()
#creiamo il tasto invio e lo associamo alla funzione send
msgSendButton = tkt.Button(window, text="Invio", command=send)
#bottone per il clear
msgClearButton = tkt.Button(window, text="Pulisci", command=clearChat)
#bottone per uscire (alternativo allo scrivere {quit} o chiudere con la x)
msgQuitButton = tkt.Button(window, text="Quit", command=on_closing)
#integriamo il tasto nel pacchetto
msgSendButton.pack()
msgQuitButton.pack()
```

```
msgClearButton.pack()
window.protocol("WM_DELETE_WINDOW", on_closing)
```

Codice Server

Inizializzazione

librerie utilizzate

```
from socket import AF_INET, socket, SOCK_STREAM
from threading import Thread
from datetime import datetime
```

Componenti

- **clients**: dizionario di chiave socket Client e valore nome scelto dal client
- **adresses**: dizionario di chiave socket client e valore indirizzo IP e porta
- **HOST**: indirizzo ip del Server, 127.0.0.1
- **PORT**: porta del Server, 53000
- **BUFSIZ**: dimensione del buffer size, 1024
- **ADDR**: “socket” del server (tupla con indirizzo server e porta, servirà a creare la socket), (‘127.0.0.1’,53000)
- **SERVER**: il vero socket del server
- **acceptThread**: thread che gestisce l’accettazione delle connessioni

funzionamento

viene fatto **SERVER.bind(ADDR)** per associare quell’indirizzo **ADDR** al socket **SERVER**. viene messo poi il server viene messo in ascolto. viene poi fatto partire il thread che accetta le connessioni e fatto il join con il thread quindi rimane in esecuzione fino a quando il thread non smette di andare. Quando il thread finisce viene chiusa la socket del server e chiuso il programma

codice

```
clients = {}
addresses = {}

HOST = '127.0.0.1'
PORT = 53000
BUFSIZ = 1024
ADDR = (HOST, PORT)

SERVER = socket(AF_INET, SOCK_STREAM)
SERVER.bind(ADDR)

if __name__ == "__main__":
    SERVER.listen(5)
    print("In attesa di connessioni...")
    acceptThread = Thread(target=acceptConnection)
    acceptThread.start()
    acceptThread.join()
```

```
    SERVER.close()
```

Funzioni principali

Il server deve svolgere due particolari funzioni:

- accettare le connessioni e quindi stabilire una connessione TCP con i client
- inviare in broadcast i messaggi che riceve da un client, quindi gestire i singoli client quando inviano un messaggio

acceptConnection()

Componenti

- **clientSocket**: socket del client da gestire
- **clientAddress**: address del client da gestire

funzionamento

quando viene mandata una richiesta di connessione viene salvato il socket e l'indirizzo del client. viene aggiunto il clientAddress in addresses con chiave clientSocket. Viene poi fatto partire il thread che gestisce il client specifico da gestire passando come argomento il clientSocket.

gestione eccezioni

viene gestita un'eccezione generale che può essere causata da un errore del thread o dal mancato invio del messaggio al client. viene gestita l'eccezione chiudendo il server

codice

```
def acceptConnection():
    while True:

        try:
            clientSocket, clientAddress = SERVER.accept()
            print("%s:%s si è collegato." % clientAdress)
            #al client che si connette per la prima volta fornisce alcune
indicazioni di utilizzo
            clientSocket.send(bytes("Salve! Digita il tuo Nome seguito dal
tasto Invio!", "utf8"))
            # ci serviamo di un dizionario per registrare i client
            addresses[clientSocket] = clientAdress
            #diamo inizio all'attività del Thread - uno per ciascun client
            Thread(target=clientHandle, args=(clientSocket,)).start()
        except Exception:
            print("Exception, closing server...")
            break
    for x in clients.keys():
        x.close()
```

clientHandle()

Componenti

name: il nome del client che gli viene inviato da lui stesso
welcome: messaggio di benvenuto che viene inviato solo al client che viene gestito dal thread
msg: è il messaggio del client che viene preso dal server e inviato in broadcast ai client

Funzionamento

fase iniziale

nella fase iniziale c'è l'attesa del client per l'invio del nome. Viene inviato in broadcast il messaggio che dice che quel client si è unito alla chat e Viene poi salvato il nome in corrispondenza del socket del client nel dizionario **clients**.

fase normale

il thread del server rimane in attesa di un messaggio del client. Se il messaggio è diverso da **{quit}** vuoi dire che si tratta di un messaggio normale e quindi viene inviato in broadcast, altrimenti chiudo la connessione con quel client.

chiusura della connessione

viene chiuso il socket del client e rimosso dai clients. viene poi mandato un messaggio agli altri client che li informa che quel client si è disconnesso. viene poi chiuso il thread (*Viene gestita meglio dalla ConnectionResetError*)

Gestione eccezioni

viene gestita l'eccezione di **ConnectionResetError** ovvero quando il client da gestire chiude la connessione. la gestisce chiudendo anche lui la connessione con quel client e rimuovendolo, se possibile, dal addresses e clients. Manda poi un messaggio in bradcast

codice

```
def clientHandle(clientSocket):  # Prende il socket del client come argomento della funzione.
    try:
        name = clientSocket.recv(BUFSIZ).decode("utf8")
        #da il benvenuto al client e gli indica come fare per uscire dalla chat quando ha terminato
        welcome = 'Benvenuto %s! Se vuoi lasciare la Chat, scrivi {quit} per uscire.' % name
        clientSocket.send(bytes(welcome, "utf8"))
        msg = "%s si è unito all chat!" % name
        #messaggio in broadcast con cui vengono avvisati tutti i client connessi che l'utente x è entrato
        broadcast(bytes(msg, "utf8"))
        #aggiorna il dizionario clients creato all'inizio
        clients[clientSocket] = name
```

```
    #si mette in ascolto del thread del singolo client e ne gestisce l'invio
dei messaggi o l'uscita dalla Chat
while True:
            msg = clientSocket.recv(BUFSIZ)
            if msg != bytes("{quit}", "utf8"):
                broadcast(msg, name+": ")
            else:
                clientSocket.send(bytes("{quit}", "utf8"))
                clientSocket.close()
                del clients[clientSocket]
                broadcast(bytes("%s ha abbandonato la Chat." % name, "utf8"))
                break
            #gestisco l'eccezione che il client si disconnetta in maniera
forzata
    except ConnectionResetError:
        adr=addresses.get(clientSocket)
        clientSocket.close()
        #condizione nel caso il client si disconnettesse prima di avere
inserito il nome e dunque di non essere inserito nei clienti
        if clientSocket in addresses.keys():
            del addresses[clientSocket]
        if clientSocket in clients.keys():
            del clients[clientSocket]
            broadcast(bytes("%s ha abbandonato la Chat." % name, "utf8"))

        print(str(adr)+" disconnessio.")
```

Funzioni secondarie

broadcast()

Componenti

dateString: ottengo la data e l'ora precisa in cui viene mandato il messaggio dal server
msgWithDate: è il messaggio che deve essere inviato più la data e ora del messaggio fra quadre

funzionamento

viene fatto un ciclo for dove prendo il singolo client da clients così da poter mandare a ciascuno client il messaggio.

codice

```
def broadcast(msg, prefix=""): # il prefisso è usato per
l'identificazione del nome.
```

```
    #aggiungo la data in cui è stato inviato il messaggio dal server ai
client
    dateString = datetime.now().strftime("%d/%m/%Y %H:%M:%S")
    msgWithDate=msg+bytes(" ["+dateString+"]","utf8")
    for utente in clients:
        utente.send(bytes(prefix, "utf8")+msgWithDate)
```