Data Management Modulo B First Project:

Christian Riccio P37000002

Giacomo Matrone P37000011

Breve introduzione alle socket

Riferendosi alle reti di calcolatori nasce naturale e spontaneo il concetto di pila protocollare, facente fede al modello ISO/OSI (l'attuale modello alla base della odierna internet) che permette ad utenti della rete di comunicare tra di loro.

L'attuale modello protocollare di ISO/OSI si presenta con una struttura a clessidra, la quale presenta 7 livelli ben distinti: partendo dal livello fisico, il più basso e risalendo fino all'ultimo livello, quello applicativo. Ed è grazie a tale livello che tutti gli utenti della rete, o in generale appartenenti ad una rete di calcolatori possono comunicare tra di loro. Ma come avviene allora tale comunicazione?

Una tipica comunicazione di rete, in primo luogo, deve uniformarsi a dei ben definiti e necessari protocolli di comunicazione: TCP ed UDP. Instaurare una connessione tra due hosts della rete significa quindi instaurare una connessione tra una coppia di programmi detti client e server (i quali si parleranno mediante i suddetti protocolli). Quando un programma viene eseguito, si parla di processo in esecuzione e si parlerà rispettivamente di processo client e processo server. Arriva ora la risposta alla domanda: i due processi in esecuzione comunicano per mezzo delle socket, che vengono a presentarsi come interfacce tra il livello applicativo (qui inteso come livello comprendente i layer di sessione, presentazione e applicazione) e quello di trasporto, qui presente come protocollo di comunicazione. Viene naturale quindi capire che si parlerà in questo caso di:

- Socket client;
- Socket server.

Le socket rappresentano quindi la cosa fondamentale alla base di una qualsiasi connessione e comunicazione di rete. Per fare riferimento al testo "Reti di Calcolatori e Internet: un approccio top-down" le socket rappresentano la porta di una casa che a sua volta simboleggia un processo. Quindi l'applicazione si trova nella casa ed il protocollo di trasporto al di là della porta: la socket rappresenta il canale di comunicazione. Ad esempio, nel caso di una comunicazione in TCP, qualsiasi informazione nella rete viaggia sotto forma di pacchetti, ognuno dei quali deve essere corredato di un header (tutte le informazioni necessarie al suo instradamento) e di un body (il corpo vero e proprio del messaggio). L'header viene quindi fornito dal processo e la socket fa da interfaccia con la rete affinché il contenuto informativo raggiunga il destinatario.

PRIMA PARTE ASSIGNMENT_: Client/server one to one

Le socket sviluppate per entrambe le parti sono state programmate in python e facenti riferimento al protocollo TCP per scambiare l'informazione. Ricordiamo a tal proposito che tramite tale protocollo le due applicazioni (client e server) prima di iniziare lo scambio di dati devono effettuare l'operazione di handshake con la quale si apre quindi il canale di comunicazione. Ciò rende necessario il dover avviare il server prima del client.

Ogni utente della rete è rappresentato come un nodo della stessa, sia esso un client o un server. Due nodi distanti possono parlare grazie alla socket, inoltre ogni nodo sarà identificato da un nome univoco, che per

---

una macchina sappiamo essere il suo indirizzo IP. Ricordiamo, inoltre, che un nodo può avere a suo carico diversi processi in esecuzione, le porte rivelano quindi quale sia lo stesso. Le porte sono comprese tra 0 e 65536, che si distinguono nelle seguenti categorie:

- le prime 1024 port (0 - 1023) sono definite well known ports ed allocano servizi quali SSH (22), FTP data (20), FTP controllo (21), SMTP (25), HTTP (80), HTTPS (443), DNS (53), ed altre;
- Le porte da 1024 a 49151 sono dette registered (da utente o applicazioni specifiche);
- Le porte fino alla 65535 sono così dette dynamic ports.

Il ruolo della socket è quello di creare una coppia univoca indirizzo IP-porta. Questa combinazione permetterà al server di esporre un servizio, mediante il quale lo stesso si mette in ascolto. È chiaro ora (volendo metaforizzare il concetto) che: un processo rappresenta la casa, l'indirizzo ip il suo indirizzo, la socket la porta e la porta (o porto) la persona che vi vive all'interno.

La prima fase necessaria a priori al lavoro di coding delle socket, è stata quella di abilitare una porta necessaria a rendere il server esposto sulla rete: la porta configurata per l'esperienza è stata la 15102. L'operazione non si è rilevata particolarmente tediosa, tranne per un **improvviso** e **precoce crollo** dell'intera rete internet casalinga non appena configurata la suddetta porta, con conseguente ricorso a riti pagani e scarmantici, nonchè invocazioni di eventuali santi da parte degli autori.

Avendo definito tutto ciò appena detto tutti i tests di fattibilità e le prove di operatività sono state condotte con il collega Giacomo Matrone.

_FASE DI SVILUPPO DELLA SOCKET SERVER

Il protocollo TCP dopo la fase di handshake instaura la vera e propria connessione, ci sarà un canale di comunicazione in cui una parte di TCP è legata al client e l'altra al server; partiamo quindi con la creazione della socket server.

Questa applicazione (e quella lato client) è stata strutturata secondo l'"**error handling"** , facendo uso delle eccezioni python. La creazione di una socket è un processo delicato, se qualsiasi funzione socket fallisce python reagirà con una eccezione, che come è possibile osservare dal codice fornito con la presente documentazione, chiamata "**socket.error**".

Di seguito sono riportate le cose che il nostro server ideale, qui presentato, deve poter fare:

1. Aprire una socket;
2. Legare la socket (bind) ad una porta ed un indirizzo ip;
3. Stare in ascolto per eventuali connessioni in arrivo;
4. Accettare la/e connessioni;
5. Leggere/rispondere ai messaggi.

Dopo aver importato il modulo socket (built-in e standard in python), sì è proceduto con il primo punto, ovvero la creazione della socket:

"**s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)**"

- AF_INET sta per 'address family': indica la famiglia di indirizzi IP, nello specifico IPv4;
- SOCK_STREAM ci dice che il flusso di informazione risponde al protocollo TCP.

"s" è detto descrittore che verrà usato in altre funzione socket collegate ad esso; tale oggetto gode delle proprietà sopra elencate.

Avendo, poi definito, le variabili "ip" e "serverPort" (i parametri affinché il server possa porsi in ascolto), si è proceduto ad accoppiare il socket sopra creato alla tupla (ip, porta) mediante l'utilizzo della funzione **"bind()".**

N.B. ip è stato passato come una stringa vuota poiché la funzione bind gli assegna automaticamente l'ip della macchina. Per quanto riguarda serverPort, è stata fornita la porta menzionata nel precedente paragrafo (la 15102).

Dopo aver legato il server alla tupla gli si concede l'abilità di ascoltare le connessioni in arrivo, con "**s.listen()**", da parte di un eventuale client. Nel caso in esame, si è deciso di passare come parametro alla funzione listen() il numero uno, in modo tale che il server ascoltasse per ora una connessione.

Lo step successivo, allora, è quello di permettere al server di accettare la connessione in arrivo. Di seguito è riportato un snippet di codice di cui ne sarà spiegato il senso:

```
29  while 1:  # infinity loop for let the server accept every sent message from the client
30      k = 0
31      connectSocket, addr = serverSocket.accept()
32      if k == 0:
33          welcome = f'welcome {addr[0]} to Chris & Gia server !!!'
34      k += 1
35      connectSocket.send(welcome.encode())
36      sentence = connectSocket.recv(1024).decode()
37      print(sentence)
38      response = input('Write here: ')
39      connectSocket.send(response.encode())
```

Finché sussiste la connessione (i.e. il client non si disconnette), il server accetterà la connessione dal socket connesso ('connectSocket'), tramite la funzione "**accept()**" (**4**).

Quando il client si connette al server, lo stesso gli manderà un messaggio di benvenuto. Il loop qui definito dà la possibilità di lasciare la connessione aperta per leggere i messaggi in entrata e rispondere (**5**).

Qualsiasi informazione in entrata e in uscita, dovrà essere:

- Decodificata, se si tratta di messaggi in entrata;
- Codificata, se si tratta di messaggi in uscita.

_FASE DI SVILUPPO DELLA SOCKET CLIENT

Dal lato di una socket client, essa deve:

1. Essere creata;
2. Connettersi ad un server remoto;
3. Mandare messaggi;
4. Ricevere una risposta.

Data Management Modulo B First Project:

Christian Riccio P37000002

Giacomo Matrone P37000011

La creazione della socket lato client si ha allo stesso modo di quella server.

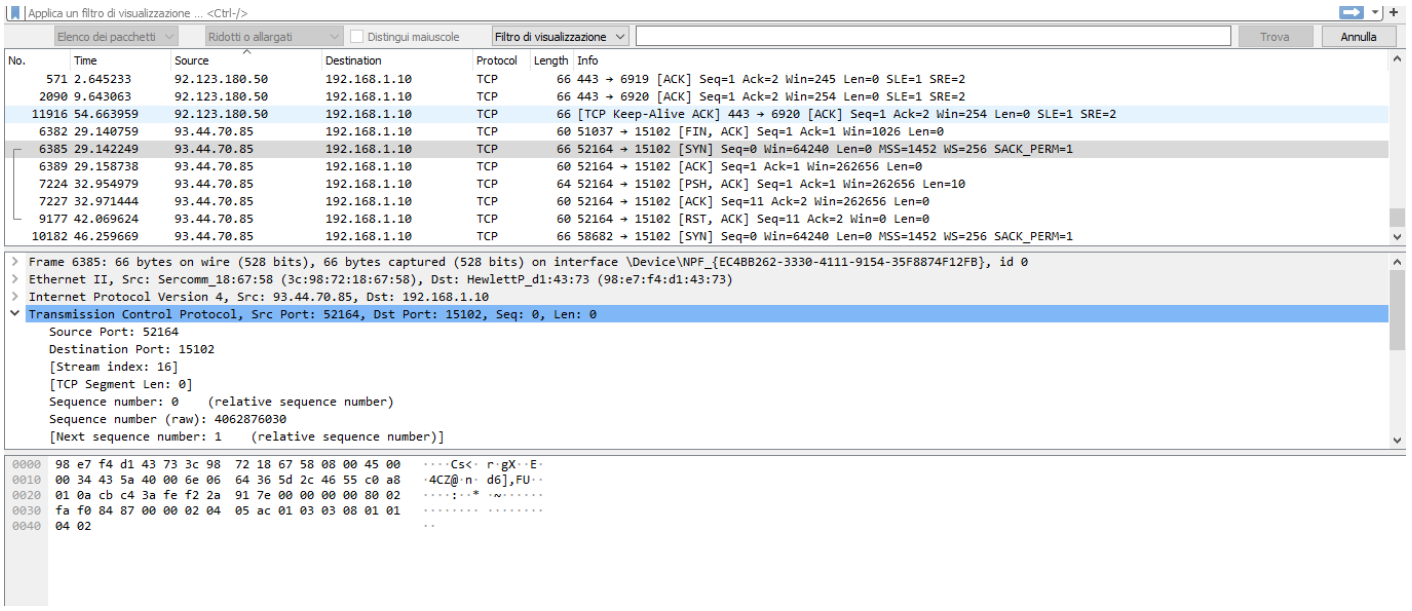
Dopo aver creato la socket, si sono definite due variabili che definiscono, rispettivamente, indirizzo e porta del server remoto. La nostra implementazione permette all'utente di inserire da terminale i campi richiesti. La connessione al server è stata fatta grazie al metodo "**socket.connect((host_ip, port))**" (**2**). La funzione connect(), come si può vedere, chiede come parametro di ingresso una tupla definita dalla coppia indirizzo ip dell'host e porta con il quale si vuole comunicare.

Avendo ora il socket connesso all'altro lato della rete è possibile scambiare messaggi con il server. Si chiede quindi all'utente di inserire il messaggio che vuole mandare in input e tramite l'uso del metodo "**socket.send()**" (**3**) il gioco è fatto, dopo aver codificato il messaggio stesso. Tale funzione chiede come parametro proprio il messaggio inserito, che sarà inoltre codificato per permettergli di viaggiare nella rete sotto forma di bit.

L'ultima fase riguarda la ricezione, da parte del client, di una risposta dal server. Questo è stato fatto grazie al metodo "**socket.recv()**" (**4**). Tutto il codice è stato inserito all'interno di un ciclo while, per permettere una comunicazione duratura e continuativa nel tempo con il server fin quando la connessione non viene interrotta. Grazie al ciclo while, ad ogni messaggio inviato con relativa risposta, si procede a ricreare la socket con il server in questione. A tal proposito, ogni risposta ricevuta viene salvata in una variabile "response" la quale viene data poi come parametro alla funzione recv(), avendola, preventivamente, decodificata, poiché come detto varie volte, ogni informazione nella rete viaggia cifrato come sequenza di 0 e 1.

Con quanto appena detto, si conclude la descrizione dell'elaborazione della prima parte del progetto. Sono state condotte varie prove, come già accennato.

Si mostra di seguito, l'immagine dell'analisi condotta con whireshark, per monitorare i pacchetti in entrata:



Data Management Modulo B First Project:

Christian Riccio P37000002

Giacomo Matrone P37000011

L'indirizzo del client è quello evidenziato in grigio, nel primo riquadro in alto. Nel riquadro centrale è possibile notare come una tipica comunicazione di rete ha luogo tra una porta alta ed una bassa: quella del client è la 52164.

Come conclusione si può dire che le prove si sono rivelate, dopo una prima fase di problemi legati al set-up del router, abbastanza semplici e veloci.

Per la parte di coding, e quindi per lo studio del modulo socket, ci siamo lasciati guidare dallo studio della documentazione reperibile al seguente link.

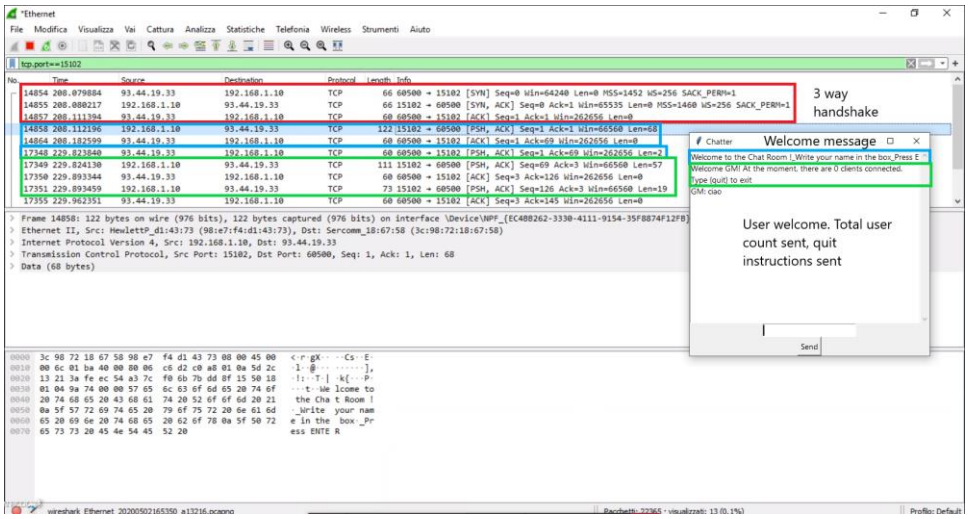
SECONDA PARTE ASSIGNMENT_: Many Clients Chat Room

Quanto detto fino ad ora vale, in buona misura, anche per la seconda parte del progetto. Infatti, questa modalità prevede ugualmente la presenza delle due socket, rispettivamente lato client e lato server ma con una leggera evoluzione. In particolare, il server ora ha la capacità di accettare e gestire molteplici connessioni in entrata da diversi utenti nonché affidare ad ogni connessione in entrata un proprio thread, con lo scopo di metterli in comunicazione. Quindi, ogni singolo messaggio da parte di un utente viene inoltrato a tutti gli altri partecipanti della "Chat Room". Il client, dal canto suo, è stato solo arricchito con qualche caratteristica in più per renderlo adatto alla tipologia di servizio con cui deve interfacciarsi, andando ad aggiungere una GUI e la possibilità di inoltrare un nome utente, nonché affidare ad una connessione in entrata ed una in uscita un proprio, rispettivo, thread.

_FASE DI SVILUPPO DELLA CHAT ROOM SERVER

L'idea di base su cui abbiamo sviluppato questo server è bene o male la stessa del single connection. La differenza riguarda la capacità del server di gestire molteplici connessioni in entrata ed in uscita da e verso i diversi client. Ogni connessone viene vista come un singolo processo. Il threading dà la possibilità non solo di far girare contemporaneamente diversi parti del programma, andando gestire in contemporanea molteplici connessioni. Si è richiesto l'utilizzo del modulo *Thread* (link per la documentazione).

La connessione iniziale sarà come mostrato nella seguente figura:



In questa fase il server, appunto, dopo aver accettato le connessioni in entrata dai vari client, deve:

Data Management Modulo B First Project:

Christian Riccio P37000002

Giacomo Matrone P37000011

- Effettuare il three way handshake (in rosso);
- Inviare un messaggio di benvenuto e recuperare il nome utente, inviato dal client (in blu);
- Inviare un messaggio di benvenuto con nome utente e conta delle persone attualmente connesse (verde);
- accettare ogni singolo messaggio e farne poi il broadcasting verso tutti gli hosts connessi, in modo che tutti i partecipanti lo leggano.

Sono state definite le seguenti funzioni:

- Una funzione per accettare le connessioni in entrata (**'accept_incoming_connections'**);
- Una funzione per gestire il client connesso, (**'handle_client'**);
- Una funzione di broadcasting dei messaggi di ogni partecipante (**'broadcast_MESSAGES'**).

La funzione che accetta le connessioni è stata strutturata con un ciclo while che permette di accettare sempre le connessioni. Ha il compito di:

- stampare a schermo alcuni dettagli della connessione;
- recapitare un messaggio di benvenuto al client connesso e di dirgli di scegliere il suo username per partecipare alla chat;
- memorizzare l'indirizzo del client connesso nel dizionario ***addresses.***

```
4   def accept_incoming_connections(): # accepting connections
5       while True:# the while loop will wait for ever for connections
6           client, client_address = SERVER.accept()
7           print("%s:%s has connected." % client_address)
8           client.send(bytes("Welcome to the Chat Room !\n Write your name in the box\n Press ENTER ", "utf8"))
9           addresses[client] = client_address # every client's address is stored into address dictionary
10          Thread(target=handle_client, args=(client,)).start()
```

La funzione appena descritta richiede quindi l'utilizzo del Thread per gestire in parallelo le connessioni da parte dei clients. Il target del Thread è stato appunto la funzione **'handle_client'** che ha il compito di gestire ed organizzare il client.

```
12  def handle_client(client):  # Takes client socket as argument, this function is for client mangement, here we want to handle a single connection
13      name = client.recv(buffer_sz).decode("utf8")
14      welcome = 'Welcome %s! If you want to quit, type {exit} to exit.' % name
15      client.send(bytes(welcome, "utf8"))
16      msg = "%s has joined the chat!" % name # every client will receive a message for every new user that will join the chat
17      broadcast_MESSAGES(bytes(msg, "utf8"))
18      clients[client] = name
19      while True:
20          msg = client.recv(buffer_sz)
21          if msg != bytes("{exit}", "utf8"):
22              broadcast_MESSAGES(msg, name + ": ")
23          else:
24              client.send(bytes("{exit}", "utf8")) # if the client type {quit} will automaticcaly be quitted from the chat by the server
25              client.close()
26              del clients[client]# naturraly the user that have quitted will be deleted from the client's list
27              broadcast_MESSAGES(bytes("%s has left the chat." % name, "utf8"))
28              break
```

Dopo aver mandato il messaggio di benvenuto al client, egli risponderà inserendo il suo username per essere riconoscibile nella chat room, per poi salvare il valore nella variabile 'name'. Tale funzione ha anche il compito di mandare ulteriori istruzioni al client, nel caso voglia abbandonare la chat. Il ciclo while ci permettere di garantire una continuità della comunicazione: in particolare se il messaggio inviato non corrisponde con la escape word ***{exit}*** il server, tramite l'ausilio della funzione di broadcasting, provvederà

Data Management Modulo B First Project:

Christian Riccio P37000002

Giacomo Matrone P37000011

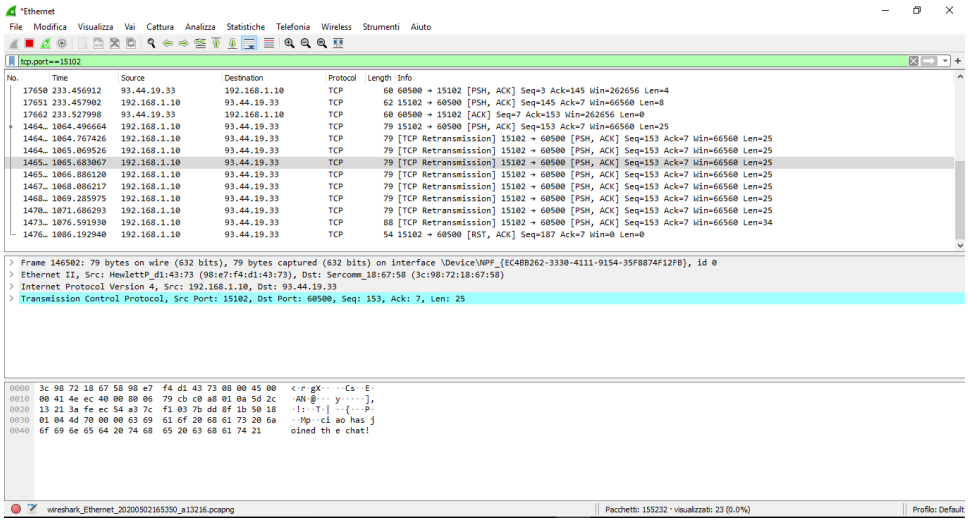
---

ad inoltrare ad ogni partecipante della chat i messaggi in entrata. Tale funzione è molto semplice, chiede come parametri la variabile 'msg', messaggio da inoltrare, e la variabile '*prefix*' che rappresenta lo username di chi ha scritto il messaggio in modo da rendere noto a tutti i partecipanti alla chat room da chi proviene lo stesso.

Si sono infine usati i seguenti metodi del modulo Thread, in ordine di esecuzione:

- **.start():** inizializza il server, in modo da gestire contemporaneamente più connessioni in entrata;
- .**join():** serve per far comunicare i Thread tra di loro. In particolare, lo snippet di codice all'interno del main attende prima di eseguire la riga che chiude la socket server.

Inoltre, in caso di necessità, il server procederà ad effettuare retransmission dei dati, come mostrato qui:



_FASE DI SVILUPPO DELLA SOCKET CLIENT

Abbiamo inoltre deciso di arricchire il lato client di una interfaccia utente grafica (GUI). Ciò è stato fatto con l'ausilio del modulo **'Tkinter'** (link alla documentazione). Lo sviluppo, la parte più onerosa ha riguardato l'implementazione della GUI.

La socket client lavora basandosi su 3 funzioni:

- Ricezione dei messaggi;
- Invio dei messaggi;
- Una funzione che gestisce la chiusura della GUI.

La funzione di ricezione del messaggio, **'receive_msg'**, implementa un ciclo while per far sì che la connessione resti sempre aperta. Per permettere ai messaggi di restare in maniera permanente all'interno della finestra di chat creata con Tkinter, ogni messaggio viene salvato all'interno della lista ***msg_list.***

Poiché abbiamo deciso di strutturare il tutto con una GUI, la funzione di invio dei messaggi (i.e. **'send_msg'**), presenta come argomento passato alla funzione un *event=None*, che è implicitamente passato da Tkinter quando si preme il pulsante di send del messaggio della GUI.

Data Management Modulo B First Project:

Christian Riccio P37000002

Giacomo Matrone P37000011

```
14  def send_msg(event=None):  # event is passed by binders.
15
16      msg = my_msg.get()
17      my_msg.set("")  # Clears input field.
18      client_socket.send(bytes(msg, "utf8"))
19      if msg == "{exit}":
20          client_socket.close()
21          top.quit()
```

Il messaggio che inviamo viene estratto dal I/O tramite il **.get()**.

Nel momento in cui si decide di chiudere la finestra della GUI, la funzione che presentiamo (**'on_closing'**) ha il compito di chiudere la socket prima che la GUI venga chiusa. Infatti essa pone, tramite metodo .set(), il campo di input con {exit}, che funge, come detto, da escape word.

```
23  def on_closing(event=None):
24
25      my_msg.set("{exit}")
26      send_msg()
27
```

Ci apprestiamo ora a descrivere la restante parte di codice che ci ha permessi di costruire la GUI.

Abbiamo come prima cosa istanziato il frame work di messaggistica della GUI:

```
29  top = tkinter.Tk()
30  top.title("CHAT ROOM")
```

Si riportano di seguito le caratteristiche della nostra GUI di messaggistica:

- il metodo .Frame() di Tkinter ci ha permesso di costruire il frame work per contenere i messaggi;
- abbiamo creato una variabile di tipo stringa e l'abbiamo impostata come "type your messages here";
- abbiamo creato uno scroll bar, per avere la possibilità di navigare tra i vari messaggi;
- Infine, un bottone permette di prendere l'input da parte dell'utente e spararlo verso il server.

Data Management Modulo B First Project:

Christian Riccio P37000002

Giacomo Matrone P37000011

```
30      messages_frame = tkinter.Frame(top)
31      my_msg = tkinter.StringVar()  # For the messages to be sent.
32      my_msg.set("Type your messages here.")
33      scrollbar = tkinter.Scrollbar(messages_frame)  # To navigate through past messages.
34      # Here we create the window for messaging.
35      msg_list = tkinter.Listbox(messages_frame, height=30, width=80, yscrollcommand=scrollbar.set)
36      scrollbar.pack(side=tkinter.RIGHT, fill=tkinter.Y)
37      msg_list.pack(side=tkinter.LEFT, fill=tkinter.BOTH)
38      msg_list.pack()
39      messages_frame.pack()
40      entry_field = tkinter.Entry(top, textvariable=my_msg)
41      entry_field.bind("<Return>", send_msg)
42      entry_field.pack()
43      send_button = tkinter.Button(top, text="Send", command=send_msg)
44      send_button.pack()
45      top.protocol("WM_DELETE_WINDOW", on_closing)
46      # socket management
47      HOST = input('Enter host: ')
48      PORT = int(input('Enter port: '))
49      buffer_sz = 1024
50      ADDR = (HOST, PORT)
51      client_socket = socket(AF_INET, SOCK_STREAM)
52      client_socket.connect(ADDR)
53      receive_thread = Thread(target=receive_msg)
54      receive_thread.start()
55      tkinter.mainloop()  # Starts GUI
```