



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA  
CAMPUS DI CESENA

# Programmazione di Reti

## Laboratorio #4

**Andrea Piroddi**

Dipartimento di Informatica, Scienza e Ingegneria

# **Trasferimento di file crittografati tramite socket in Python**



# CRITTOGRAFIA AES (Advanced Encryption Standard)

L'algoritmo di crittografia simmetrica più popolare e ampiamente adottato è l'Advanced Encryption Standard (AES).

Le caratteristiche di AES sono le seguenti:

- Cifrario a blocchi simmetrico a chiave simmetrica
- Dati a 128 bit, chiavi a 128/192/256 bit



# CRITTOGRAFIA AES (Advanced Encryption Standard)

AES è un cifrario iterativo. Si basa su "sostituzione-permutazione". Comprende una serie di operazioni collegate, alcune delle quali comportano la sostituzione di input con output specifici (sostituzioni) e altre implicano il mescolamento di bit (permutazioni).

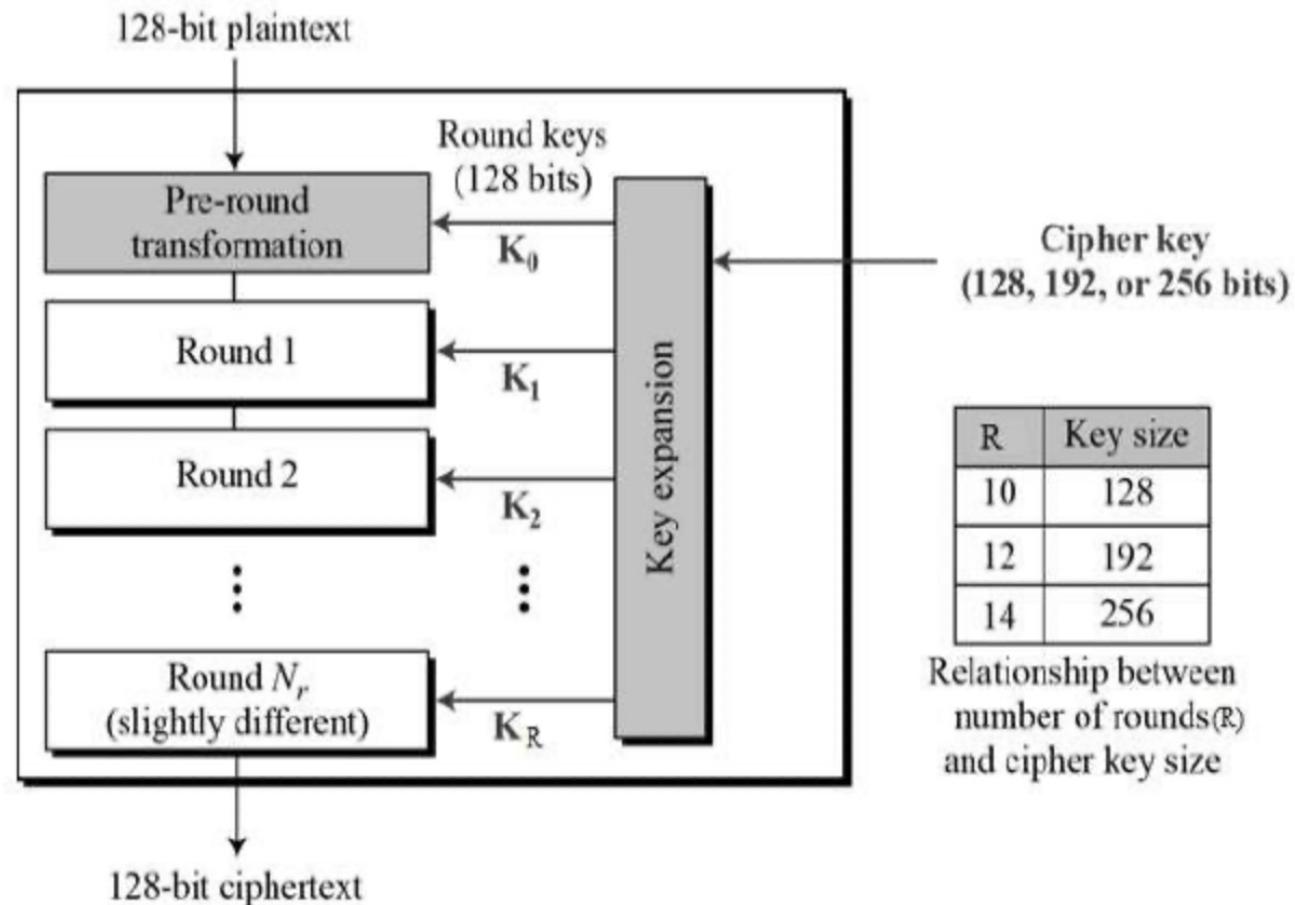
È interessante notare che AES esegue tutti i suoi calcoli su byte anziché su bit. Pertanto, AES tratta i 128 bit di un blocco di testo in chiaro come 16 byte. Questi 16 byte sono disposti in quattro colonne e quattro righe per l'elaborazione come matrice.

Il numero di cicli in AES è variabile e dipende dalla lunghezza della chiave. AES utilizza 10 round per chiavi a 128 bit, 12 round per chiavi a 192 bit e 14 round per chiavi a 256 bit. Ciascuno di questi cicli utilizza una diversa chiave a 128 bit, che viene calcolata dalla chiave AES originale.



# CRITTOGRAFIA AES (Advanced Encryption Standard)

Lo schema della struttura AES è riportato nell'illustrazione seguente:



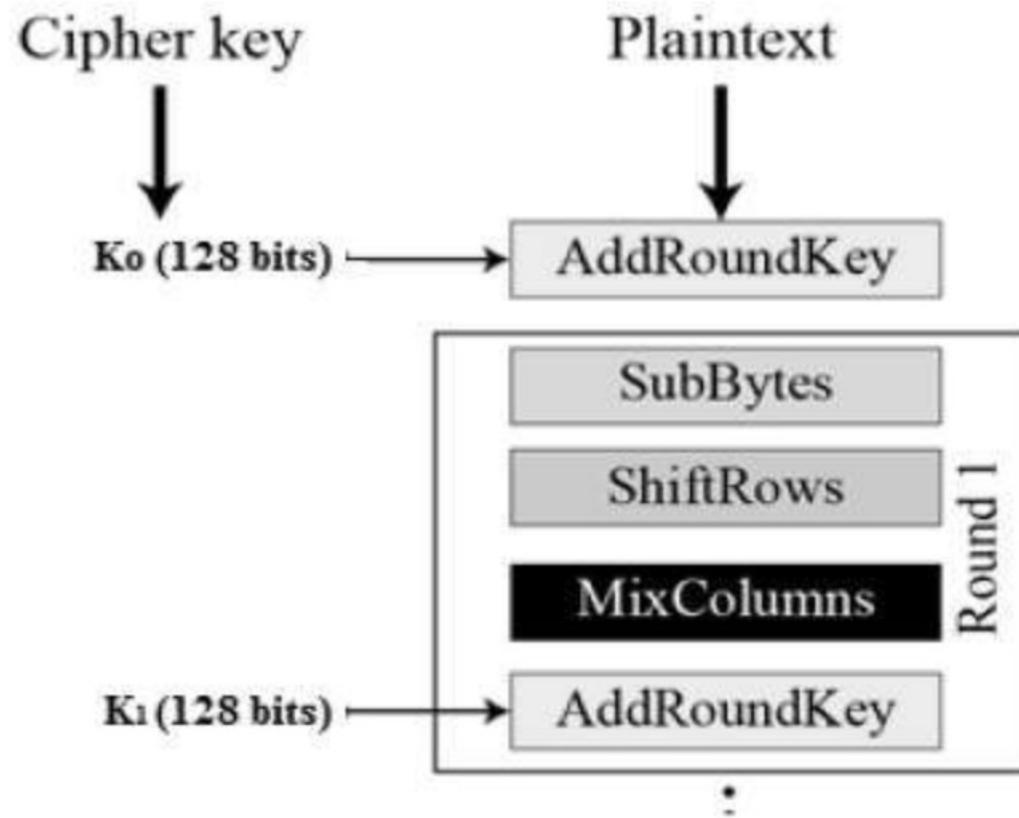
R	Key size
10	128
12	192
14	256

Relationship between  
number of rounds(R)  
and cipher key size



# CRITTOGRAFIA AES (Advanced Encryption Standard)

Ogni round comprende quattro sottoprocessi. Il processo del primo round è illustrato di seguito



# Trasferimento di file crittografati tramite socket in Python

Il trasferimento di file in modo sicuro è una funzionalità indispensabile in qualsiasi ambiente di sviluppo.

La crittografia end-to-end garantisce che nemmeno il server possa leggere il contenuto dei dati.

Il trasferimento di file crittografati su socket in Python è un metodo per inviare in modo sicuro file da un computer a un altro.

Implica la creazione di una connessione socket tra il client e il server e la crittografia del file sul lato client prima di inviarlo al server.

Il server quindi decriptografa il file e lo salva nella posizione desiderata.



# Trasferimento di file crittografati tramite socket in Python

Il processo può essere suddiviso in diverse fasi:

1. Il client stabilisce una connessione con il server utilizzando il modulo socket.
2. Il client legge il file da trasferire e lo crittografa utilizzando una libreria come ***pycrypto*** o ***cryptography***.
3. Il file crittografato viene inviato al server tramite la connessione socket stabilita.
4. Lato server, il file crittografato ricevuto viene decrittografato utilizzando la stessa libreria di crittografia e salvato nella posizione desiderata.



# Trasferimento di file crittografati tramite socket in Python

È importante notare che la chiave di crittografia deve essere scambiata in modo sicuro tra il client e il server prima del trasferimento.

Ciò può essere ottenuto tramite un protocollo di scambio di chiavi sicuro come Diffie-Hellman.

È anche importante utilizzare un algoritmo di crittografia robusto come AES per garantire che il file sia protetto durante la trasmissione.

Inoltre, è anche importante gestire correttamente gli errori, i trasferimenti di file di grandi dimensioni e la gestione delle chiavi.



# Trasferimento di file crittografati tramite socket in Python – Diffie-Hellmann

Si considera inizialmente un numero  $g$ , generatore del gruppo moltiplicativo degli interi modulo  $p$ , dove  $p$  è un numero primo.

Uno dei due interlocutori, ad esempio Alice, sceglie un numero casuale " $a$ " e calcola il valore  $A = g^a \text{ mod } p$  (dove *mod* indica l'operazione modulo, ovvero il resto della divisione intera) e lo invia attraverso il canale pubblico a Bob (l'altro interlocutore), assieme ai valori  $g$  e  $p$ .

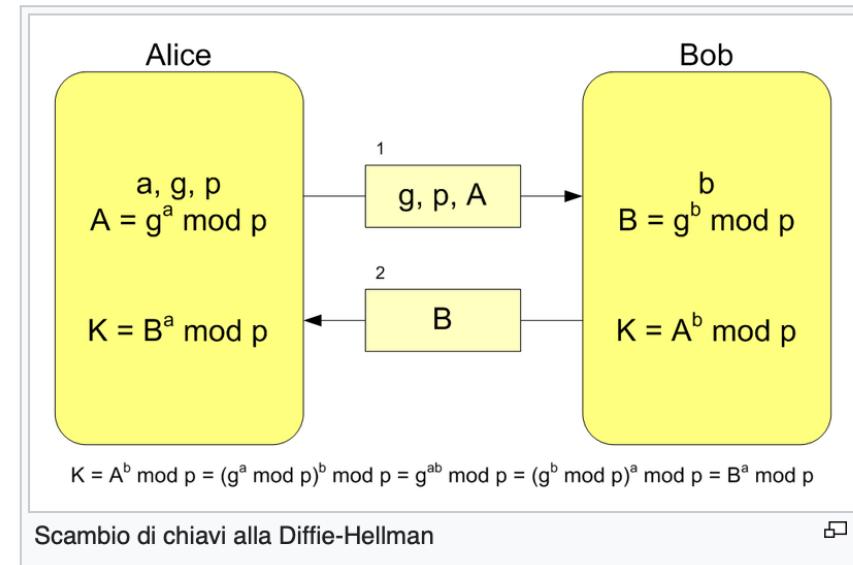
Bob da parte sua sceglie un numero casuale " $b$ ", calcola  $B = g^b \text{ mod } p$  e lo invia ad Alice.

A questo punto Alice calcola  $K_A = B^a \text{ mod } p$ , mentre Bob calcola  $K_B = A^b \text{ mod } p$ .

I valori calcolati sono gli stessi, in quanto  $B^a \text{ mod } p = A^b \text{ mod } p$ .

A questo punto i due interlocutori sono entrambi in possesso della chiave segreta e possono cominciare ad usarla per cifrare le comunicazioni successive.

Un attaccante può ascoltare tutto lo scambio, ma per calcolare i valori  $a$  e  $b$  avrebbe bisogno di risolvere l'operazione del logaritmo discreto, che è computazionalmente onerosa e richiede parecchio tempo, in quanto sub-esponenziale (sicuramente molto più del tempo di conversazione tra i 2 interlocutori).



# Trasferimento di file crittografati tramite socket in Python

Per questo trasferimento di file crittografato utilizzeremo la crittografia simmetrica, ciò significa sostanzialmente che utilizzeremo la stessa chiave per la crittografia e per la decrittografia.

Questo metodo è diverso dalla crittografia RSA in cui abbiamo chiavi pubbliche e private, dove la chiave pubblica è utilizzata per la crittografia mentre la chiave privata è utilizzata per la decrittografia

Prima di andare avanti dobbiamo installare due pacchetti python:



# Trasferimento di file crittografati tramite socket in Python

## Pycryptodome:

Pycryptodome è un pacchetto Python che supporta sia Python 2 che 3.

Pycryptodome fornisce una raccolta di moduli crittografici che supportano vari algoritmi come cifrari simmetrici, digest di messaggi e sistemi crittografici a chiave pubblica.

Può essere utilizzato per attività come la crittografia e la decrittografia dei dati, la generazione e la gestione di chiavi crittografiche e l'esecuzione di altre operazioni crittografiche.

Per installare pycrypto utilizzare il seguente comando nel prompt dei comandi o nel terminale:

```
1 pip3 install Pycryptodome
```



# Trasferimento di file crittografati tramite socket in Python

## TQDN:

TQDN è una libreria Python che fornisce una barra di avanzamento rapida ed estensibile per loop e tabelle. È progettato per fornire un feedback visivo sull'avanzamento di un ciclo o di un'iterazione e può essere utilizzato per tenere traccia dell'avanzamento di attività quali trasferimenti di file, elaborazione dati e altro.

Per installare TQDN utilizzare il seguente comando nel prompt dei comandi o nel terminale:

```
1 pip3 install tqdm
```



# Trasferimento di file crittografati tramite socket in Python

Ora andiamo a creare due script uno per il mittente e uno per il destinatario.

Step 1: importare la libreria richiesta

Step 2: creare la chiave

Step 3: crea un codice

Step 4: creare un socket TCP per la comunicazione Internet

Step 5: connettersi all'host locale in cui è ospitato il server ricevente

Step 6: Calcola la dimensione dei file da inviare

Step 7: caricare i dati sotto forma di byte

Step 8: crittografare i dati

Step 9: inviare al destinatario il nome del file, la dimensione del file, il file crittografato e un tag di chiusura per indicare che il file è stato ricevuto completamente.



# Trasferimento di file crittografati tramite socket in Python - SERVER

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Sat Apr 1 18:27:34 2023
5  LABORATORIO DI PROGRAMMAZIONE DI RETI
6  @author: apirodd
7  """
8
9  """
10 Server che riceve il file
11 """
12 import socket
13 import tqdm
14 import os
15 from Crypto.Cipher import AES
16
17 #definiamo le chiavi
18 key = b"TheBestRandomKey"
19 nonce = b"TheBestRandomNce"
20
21 cipher = AES.new(key, AES.MODE_EAX, nonce)
22
23 filename1="Cartel3.csv"
24 # IP address del server ricevente
25 SERVER_HOST = "127.0.0.1"
26 SERVER_PORT = 5001
27 # riceve 4096 bytes ogni tranne
28 BUFFER_SIZE = 1024 *4
29 SEPARATOR = "<SEPARATOR>"
30 # creiamo il socket del server
31 # TCP socket
32 s = socket.socket()
33 # facciamo il bind del socket all'ip address locale
34 s.bind((SERVER_HOST, SERVER_PORT))
35
36 # abilitiamo il nostro server ad accettare connessioni
37 # 5 è il numero di connessioni in attesa che il sistema
38 # processerà prima, le altre saranno rifiutate
39
40 s.listen(5)
41 print(f"[*] Listening as {SERVER_HOST}:{SERVER_PORT}")
```



# Trasferimento di file crittografati tramite socket in Python - SERVER

```
41  print(f"[*] Listening as {SERVER_HOST}:{SERVER_PORT}")
42  # accetta la connessione se presente
43
44  client_socket, address = s.accept()
45  # se il codice che segue viene eseguito significa che il mittente è connesso
46  print(f"[+] {address} is connected.")
47  # riceve le informazioni sul file
48  # riceve usandi il socket del client, non il socket del server
49
50  received = client_socket.recv(BUFFER_SIZE).decode("utf-8", "ignore")
51  pippo=[filename, filesize] = received.split(SEPARATOR)
52  # rimuove il percorso assoluto se presente
53
54  filename = os.path.basename(pippo[0])
55  # converte in un intero
56  filesize = int(pippo[1])
57  # comincia a ricevere il file dal socket
58  # e lo scrive sul file stream
59
60  progress = tqdm.tqdm(range(filesize), f"Receiving {filename}", unit="B", unit_scale=True, unit_divisor=1024)
61  with open(filename1, "wb") as f:
62      while True:
63          # legge 1024 bytes dal socket (receive)
64          bytes_read = client_socket.recv(BUFFER_SIZE)
65          if not bytes_read:
66              # non riceve nulla
67              # la trasmissione del file è completa
68              break
69
70          # scrive nel file i bytes che abbiamo ricevuto
71          f.write(bytes_read)
72          print(bytes_read)
73          # aggiorna la barra di avanzamento
74          progress.update(len(bytes_read))
75          if bytes_read[-5:] == b"<END>":
76              break
77      # chiude il client socket
78  client_socket.close()
79  # chiude il server socket
80  s.close()
```



# Trasferimento di file crittografati tramite socket in Python - CLIENT

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Sat Apr 1 18:26:35 2023
5  LABORATORIO DI PROGRAMMAZIONE DI RETI
6  @author: apirodd
7  """
8  """
9  """
10 Client that sends the file (uploads)
11 """
12 import socket
13 import tqdm
14 import os
15 import argparse
16 from Crypto.Cipher import AES
17 key = b"TheBestRandomKey"
18 nonce = b"TheBestRandomNce"
19 # creiamo la cifratura
20
21 cipher = AES.new(key, AES.MODE_EAX, nonce)
22
23 SEPARATOR = "<SEPARATOR>"
24 BUFFER_SIZE = 1024 * 4 #4KB
25
26 def send_file(filename, host, port):
27     # recuperiamo la dimensione del file
28     filesize = os.path.getsize(filename)
29     # creiamo il socket del client
30     s = socket.socket()
31     print(f"[+] Connecting to {host}:{port}")
32     s.connect((host, port))
33     print("[+] Connected.")
34
35     # inviamo il nome del file e la dimensione del file
36     s.send(f"{filename}{SEPARATOR}{filesize}".encode())
```



# Trasferimento di file crittografati tramite socket in Python - CLIENT

```
37      # cominciamo ad inviare il file
38      progress = tqdm.tqdm(range(filesize), f"Sending {filename}", unit="B", unit_scale=True, unit_divisor=1024)
39      with open(filename, "rb") as f:
40          while True:
41              # leggiamo i bytes dal file
42              bytes_read = f.read(BUFFER_SIZE)
43              if not bytes_read:
44                  # la trasmissione del file è completata
45                  break
46              # utilizziamo sendall per garantire la trasmissione
47              # in reti trafficate
48              # criptiamo i dati
49              encrypted = cipher.encrypt(bytes_read)
50              s.sendall(encrypted)
51              # aggiorniamo la barra di avanzamento
52              progress.update(len(bytes_read))
53              s.send(b"<END>")
54      # chiudiamo il socket
55      s.close()
56
57
58  if __name__ == "__main__":
59      import argparse
60      parser = argparse.ArgumentParser(description="Simple File Sender")
61      parser.add_argument("file", help="File name to send")
62      parser.add_argument("host", help="The host/IP address of the receiver")
63      parser.add_argument("-p", "--port", help="Port to use, default is 5001", default=5001)
64      args = parser.parse_args()
65      filename = args.file
66      host = args.host
67      port = args.port
68      send_file(filename, host, port)
69
```



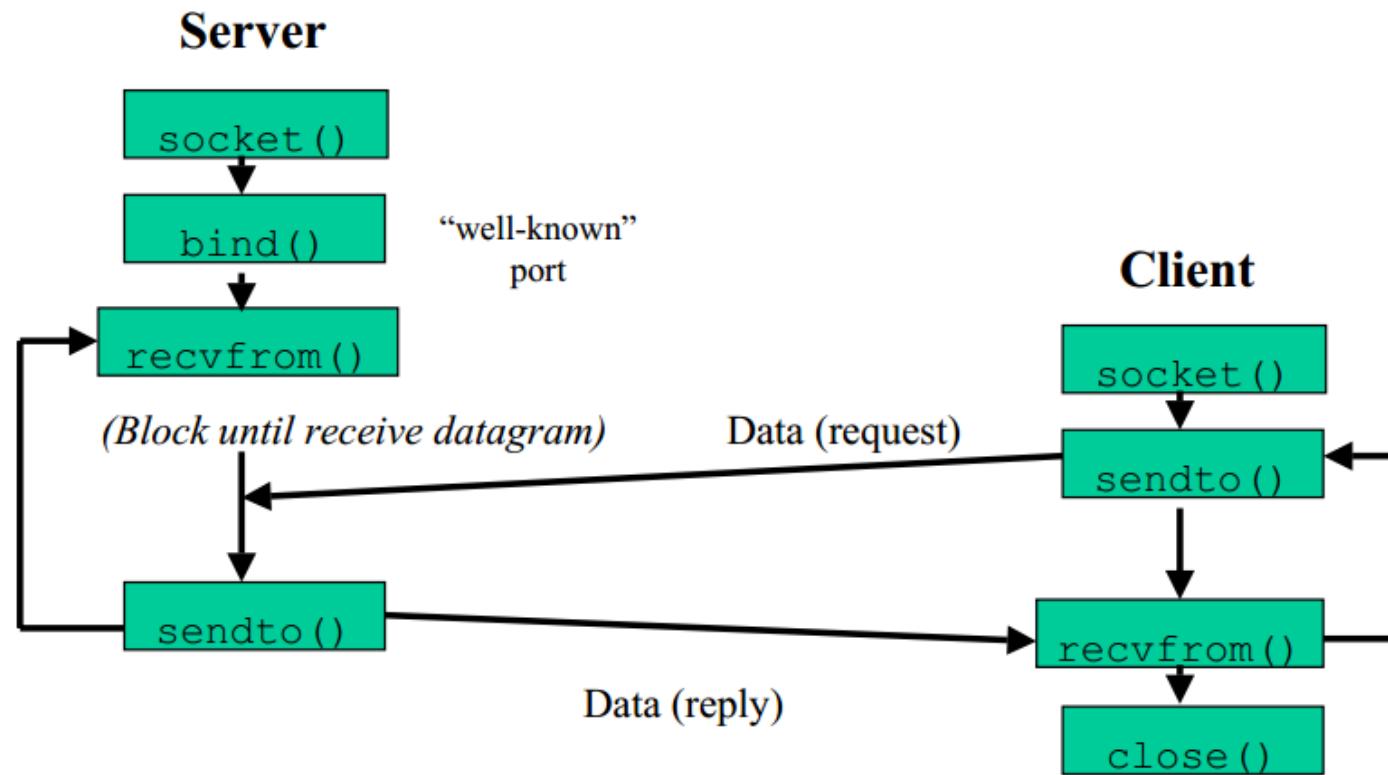
# **UDP**

# **SERVER SOCKET**

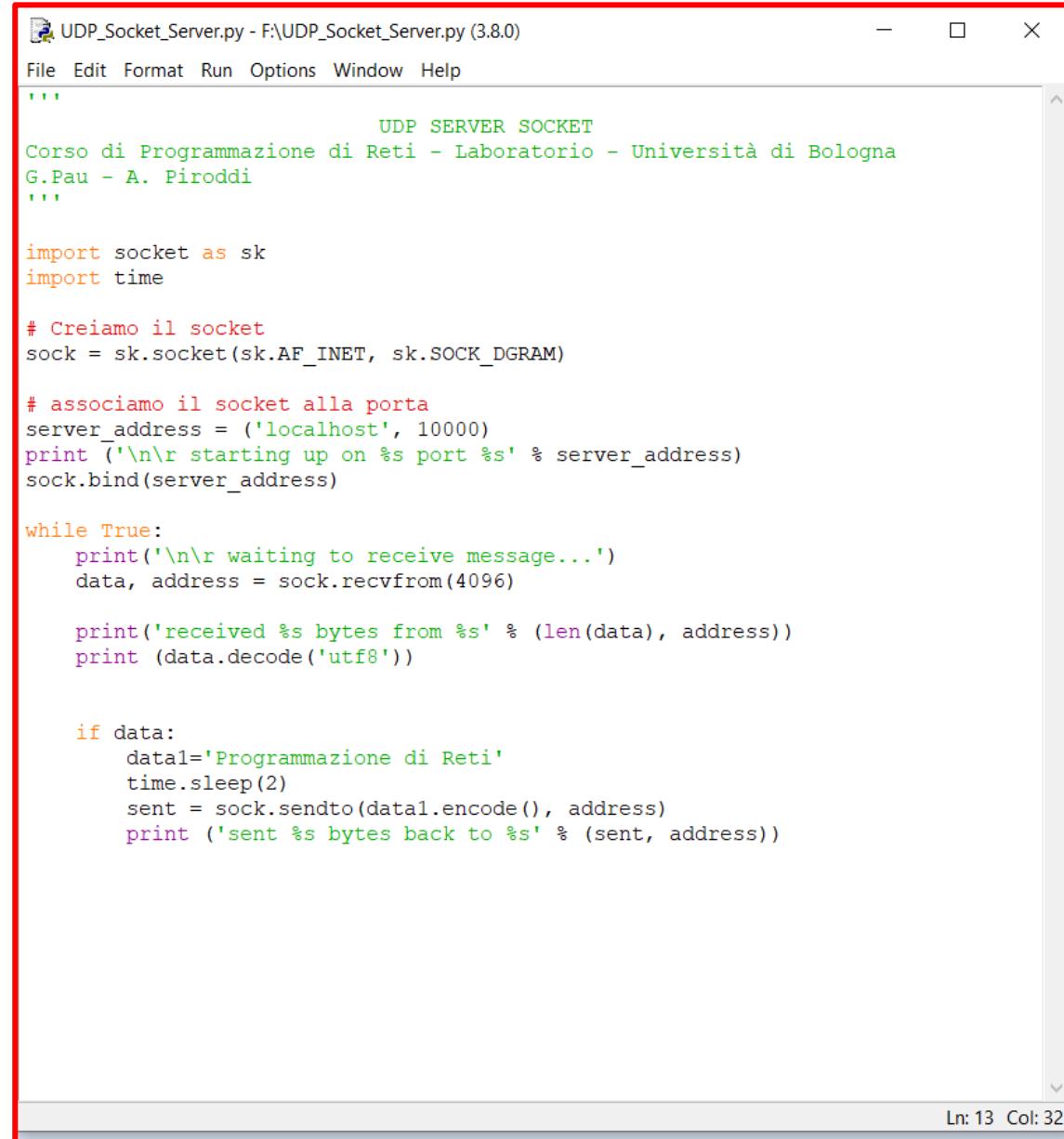


# UDP - SERVER SOCKET e CLIENT SOCKET

## UDP Client-Server



# UDP SERVER SOCKET



The screenshot shows a window titled "UDP\_Socket\_Server.py - F:\UDP\_Socket\_Server.py (3.8.0)". The code is a Python script for a UDP server. It starts with a header block containing the title "UDP SERVER SOCKET", the course information "Corso di Programmazione di Reti - Laboratorio - Università di Bologna", and the authors "G.Pau - A. Piroddi". The script then imports the "socket" module and "time" module. It creates a socket object "sock" using AF\_INET and SOCK\_DGRAM. It binds the socket to the address ('localhost', 10000). A loop begins, printing a message to the console and receiving data from the socket. It then prints the received data length and content. If data is received, it sends a response back to the client. The status bar at the bottom indicates "Ln: 13 Col: 32".

```
'''  
    UDP SERVER SOCKET  
    Corso di Programmazione di Reti - Laboratorio - Università di Bologna  
    G.Pau - A. Piroddi  
'''  
  
import socket as sk  
import time  
  
# Creiamo il socket  
sock = sk.socket(sk.AF_INET, sk.SOCK_DGRAM)  
  
# associamo il socket alla porta  
server_address = ('localhost', 10000)  
print ('\n\r starting up on %s port %s' % server_address)  
sock.bind(server_address)  
  
while True:  
    print ('\n\r waiting to receive message...')  
    data, address = sock.recvfrom(4096)  
  
    print('received %s bytes from %s' % (len(data), address))  
    print (data.decode('utf8'))  
  
    if data:  
        data1='Programmazione di Reti'  
        time.sleep(2)  
        sent = sock.sendto(data1.encode(), address)  
        print ('sent %s bytes back to %s' % (sent, address))
```



## UDP SERVER SOCKET

```
import socket as sk
```

Importiamo il modulo socket nella forma sintatticamente appropriata e lo associamo al nome abbreviato

**sk**

che utilizzeremo nel resto del codice.



# UDP SERVER SOCKET

```
import time
```



Importiamo il modulo time.

Il modulo time consente di gestire le attività legate al tempo.

Per esempio se voleste verificare quanto tempo dura l'esecuzione di un codice o di una parte di esso, potete inserire prima dell'inizio del codice l'istruzione

```
t0=time.time()
```

E

a valle del codice

```
t1=time.time()-t0
```

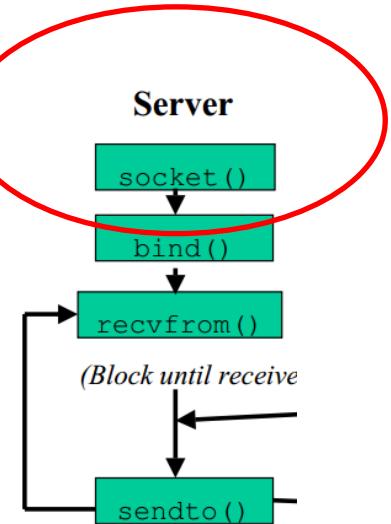
```
Print(t1)
```

in modo da calcolare l'intervallo.



# UDP SERVER SOCKET

```
# Creiamo il socket  
sock = sk.socket(sk.AF_INET, sk.SOCK_DGRAM)
```



Creiamo il socket UDP e lo associamo alla variabile `sock`



# UDP SERVER SOCKET

```
# associamo il socket alla porta
server_address = ('localhost', 10000)
print ('\n\r starting up on %s port %s' % server_address)
```



Definiamo l'indirizzo IP del nostro server e la porta su cui lo metteremo in ascolto.

NOTA: se inserite al posto di `localhost` un indirizzo che non è compatibile con la macchina (per esempio 1.2.3.4) vi verrà restituito un errore dal sistema operativo del tipo:

## WINDOWS

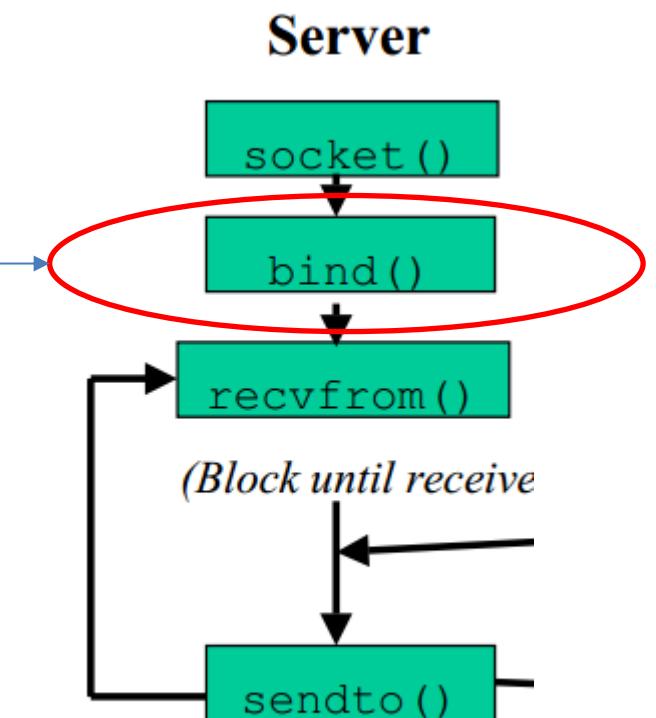
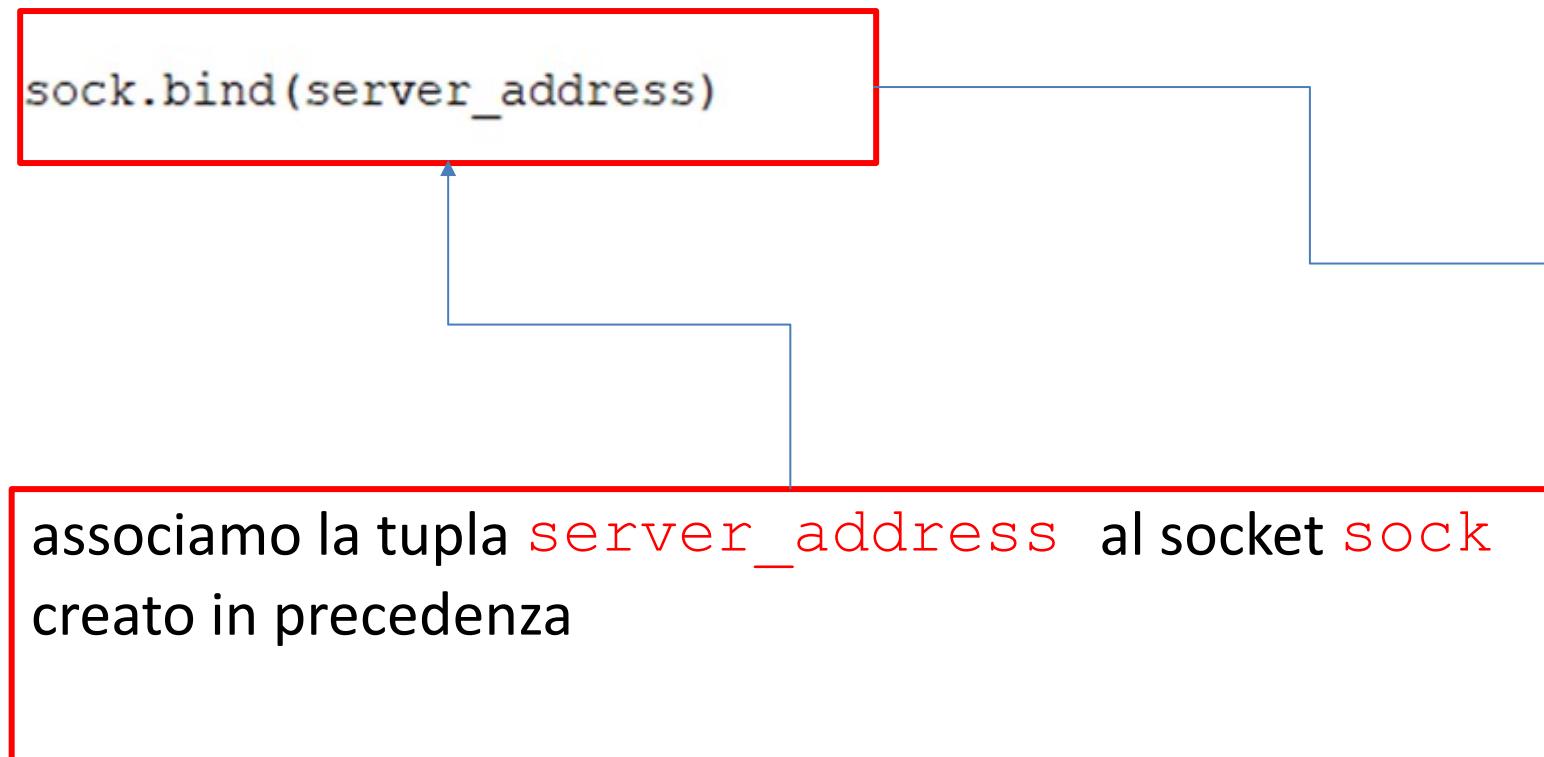
```
starting up on 1.2.3.4 port 10000
Traceback (most recent call last):
  File "F:\UDP_Socket_Server.py", line 16, in <module>
    sock.bind(server_address)
socket.error: [WinError 10049] Indirizzo richiesto non valido nel proprio contesto
>>>
```

## LINUX

```
apirodd@ubuntunet2008:~$ python3 UDP_Socket_Server.py
starting up on 1.2.3.4 port 10000
Traceback (most recent call last):
  File "UDP_Socket_Server.py", line 16, in <module>
    sock.bind(server_address)
socket.error: [Errno 99] Cannot assign requested address
apirodd@ubuntunet2008:~$
```



# UDP SERVER SOCKET



# UDP SERVER SOCKET

```
while True:  
    print('\n\r waiting to receive message...')  
    data, address = sock.recvfrom(4096)
```

Diamo inizio al ciclo while per creare il loop.

E attendiamo l'arrivo di qualche messaggio sul socket `sock` su cui abbiamo impostato la dimensione del buffer a 4096 (\*). Quando un messaggio si presenterà ne assegneremo il contenuto alle due variabili `data` e `address`

(\*)

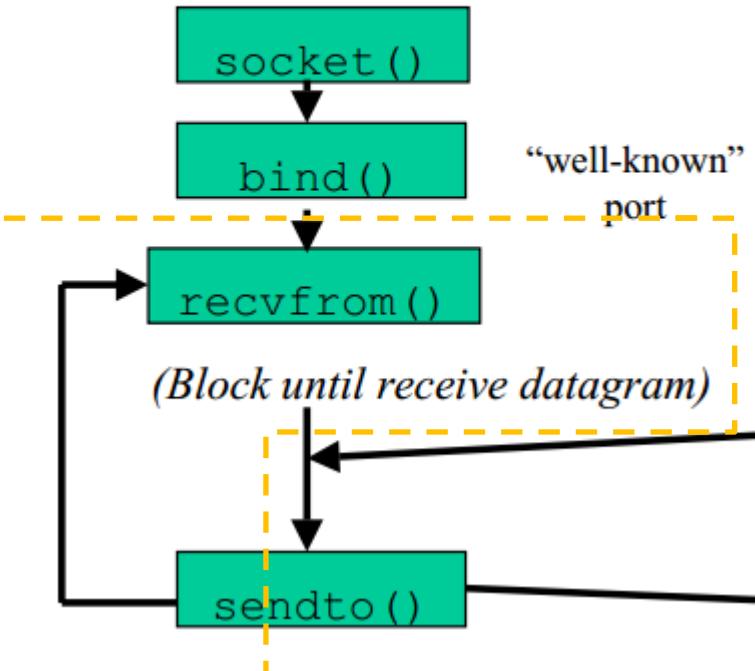
`socket.recv(bufsize[, flags])`

Riceve dati dal socket connesso, restituendo una stringa binaria. **bufsize** specifica la dimensione del buffer di ricezione.

Valori consigliati di bufsize sono potenze di 2 relativamente basse, come 1024, 4096.

**flags** sono delle flag platform-dependent che modificano il comportamento della funzione.

## Server



# UDP SERVER SOCKET

```
print('received %s bytes from %s' % (len(data), address))  
print (data.decode('utf8'))
```

Visualizziamo quindi la lunghezza dei dati che sono arrivati sul socket e l'indirizzo di provenienza in modo da sapere a chi rispondere.

## NOTA:

Python utilizza la formattazione di stringhe in stile C per creare nuove stringhe formattate.

L'operatore "%" viene utilizzato per formattare una serie di variabili racchiuse in una "tupla", insieme ad un formato stringa, che contiene testo normale, insieme a "identificatori di argomenti" e simboli speciali come "% s" e "% d".

Il metodo opposto di `bytes.decode ()` è `str.encode ()`, che restituisce una rappresentazione in byte della stringa Unicode, codificata nella codifica richiesta.

```
script.py  
1 # This prints out "John is 23 years old."  
2 name = "John"  
3 age = 23  
4 print("%s is %d years old." % (name, age))
```

```
IPython Shell  
John is 23 years old.  
In [1]: |
```

%s - String (or any object with a string representation, like numbers)  
%d - Integers  
%f - Floating point numbers  
. <number of digits> f - Floating point numbers with a fixed amount of digits to the right of the dot.  
%x/%X - Integers in hex representation (lowercase/uppercase)



## UDP SERVER SOCKET

```
if data:  
    data1='Programmazione di Reti'  
    time.sleep(2)
```

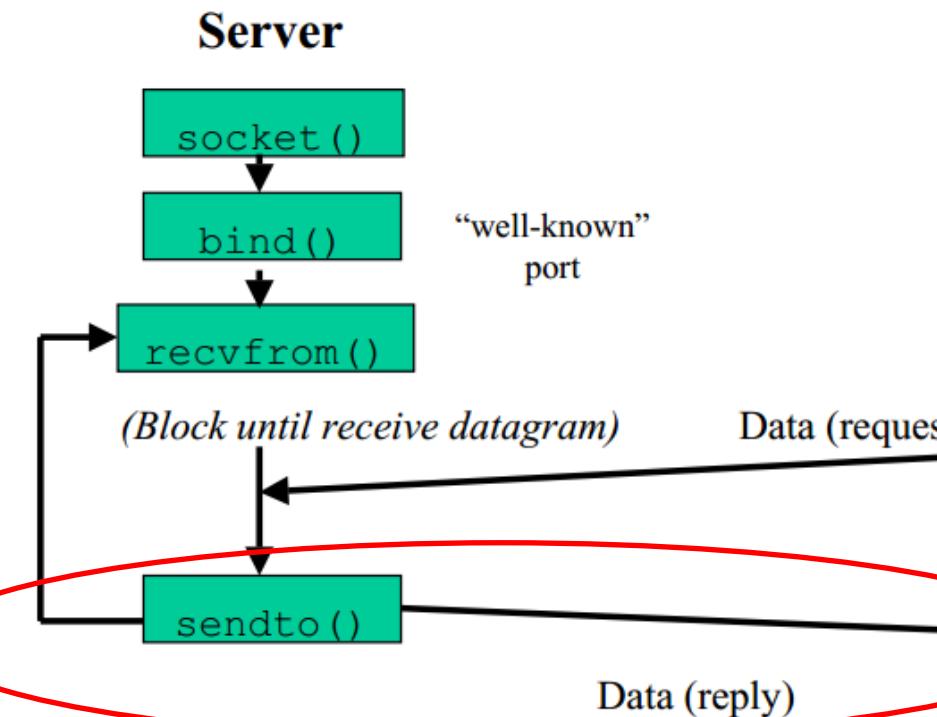
Controlla che i dati che sono arrivati sul socket non siano vuoti; in caso la variabile `data` non sia vuota, assegna alla variabile `data1` la stringa  
`'Programmazione di Reti'`  
e poi attende 2 secondi prima di inviarla verso il client.



# UDP SERVER SOCKET

```
sent = sock.sendto(data1.encode(), address)  
print ('sent %s bytes back to %s' % (sent, address))
```

Inviamo verso il client (address) il contenuto della variabile data opportunamente codificato.



# UDP SERVER SOCKET – Note sull'utilizzo di decode()/encode() in Python3

Cosa è cambiato in Python 3 che causa la restituzione di errori del tipo *UnicodeDecodeError* e *UnicodeEncodeError* che non si presentavano in Python 2?

La differenza fondamentale è che il comportamento di elaborazione del testo predefinito in Python 3 mira a rilevare i problemi di codifica del testo il più presto possibile - sia quando si legge un testo codificato in modo errato (indicato da *UnicodeDecodeError*) o quando viene richiesto di scrivere una sequenza di testo che non può essere rappresentata correttamente nella codifica di destinazione (indicata da *UnicodeEncodeError*).

Ciò è in contrasto con l'approccio Python 2 che consentiva la «corruzione» dei dati mentre i controlli di correttezza dovevano essere esplicitamente richiesti. Ciò potrebbe certamente essere utile quando i dati elaborati erano prevalentemente testo ASCII ed era improbabile che il bit corrotto occasionale venisse rilevato, ma difficilmente questo approccio è una base solida per la creazione di applicazioni multilingue.



# Confronto UDP e TCP Server

**UDP\_Socket\_Server.py - F:\UDP\_Socket\_Server.py (3.8.0)**

```
"""
    UDP SERVER SOCKET
Corso di Programmazione di Reti - Laboratorio - Università di Bologna
G.Pau - A. Piroddi
"""

import socket as sk
import time

# Creiamo il socket
sock = sk.socket(sk.AF_INET, sk.SOCK_DGRAM) 1

# associamo il socket alla porta
server_address = ('localhost', 10000)
print ('\n\n starting up on %s port %s' % server_address)
sock.bind(server_address)

while True:
    print ('\n\n waiting to receive message...')
    data, address = sock.recvfrom(4096) 4

    print('received %s bytes from %s' % (len(data), address))
    print (data.decode('utf8')) 5

    if data:
        data1='Programmazione di Reti'
        time.sleep(2)
        sent = sock.sendto(data1.encode(), address)
        print ("sent %s bytes back to %s" % (sent, address)) 6

Send: invia un messaggio attraverso un socket
Sendto: invia un messaggio attraverso un socket (non orientato alla connessione)
```

**TCP\_Socket\_Server.py - F:\TCP\_Socket\_Server.py (3.8.0)**

```
# Corso di Programmazione di Reti - Laboratorio - Università di Bologna
# Socket_Programming_Assignment - WebServer - G.Pau - A. Piroddi

import sys
from socket import *

serverSocket = socket(AF_INET, SOCK_STREAM) 1
server_address=('localhost',8080)
serverSocket.bind(server_address)

listen(1) Definisce la lunghezza della coda di backlog, ovvero il numero
#di connessioni in entrata che sono state completate dallo stack TCP / IP
#ma non ancora accettate dall'applicazione.
serverSocket.listen(1)
print ('the web server is up on port:',8080) 2

while True:
    print ("Ready to serve...")
    connectionSocket, addr = serverSocket.accept() 3
    print(connectionSocket,addr)

    message = "" + str(len(data)) + "\r\n"
    message += data
    connectionSocket.sendall(message) 4

    filename = message.split()[1]
    f = open(filename[1:], "r")
    outputdata = f.read()
    print (outputdata)

    connectionSocket.send("HTTP/1.1 200 OK\r\n\r\n".encode())
    connectionSocket.sendall(outputdata.encode())
    connectionSocket.send("\r\n".encode()) 5
    connectionSocket.close() 6

except IOError:
    #Invia messaggio di risposta per file non trovato
    connectionSocket.sendall(bytes("HTTP/1.1 404 Not Found\r\n\r\n","UTF-8"))
    connectionSocket.sendall(bytes("<html><head></head><body><h1>404 Not Found</h1></body></html>","UTF-8"))
    connectionSocket.close() 7
```

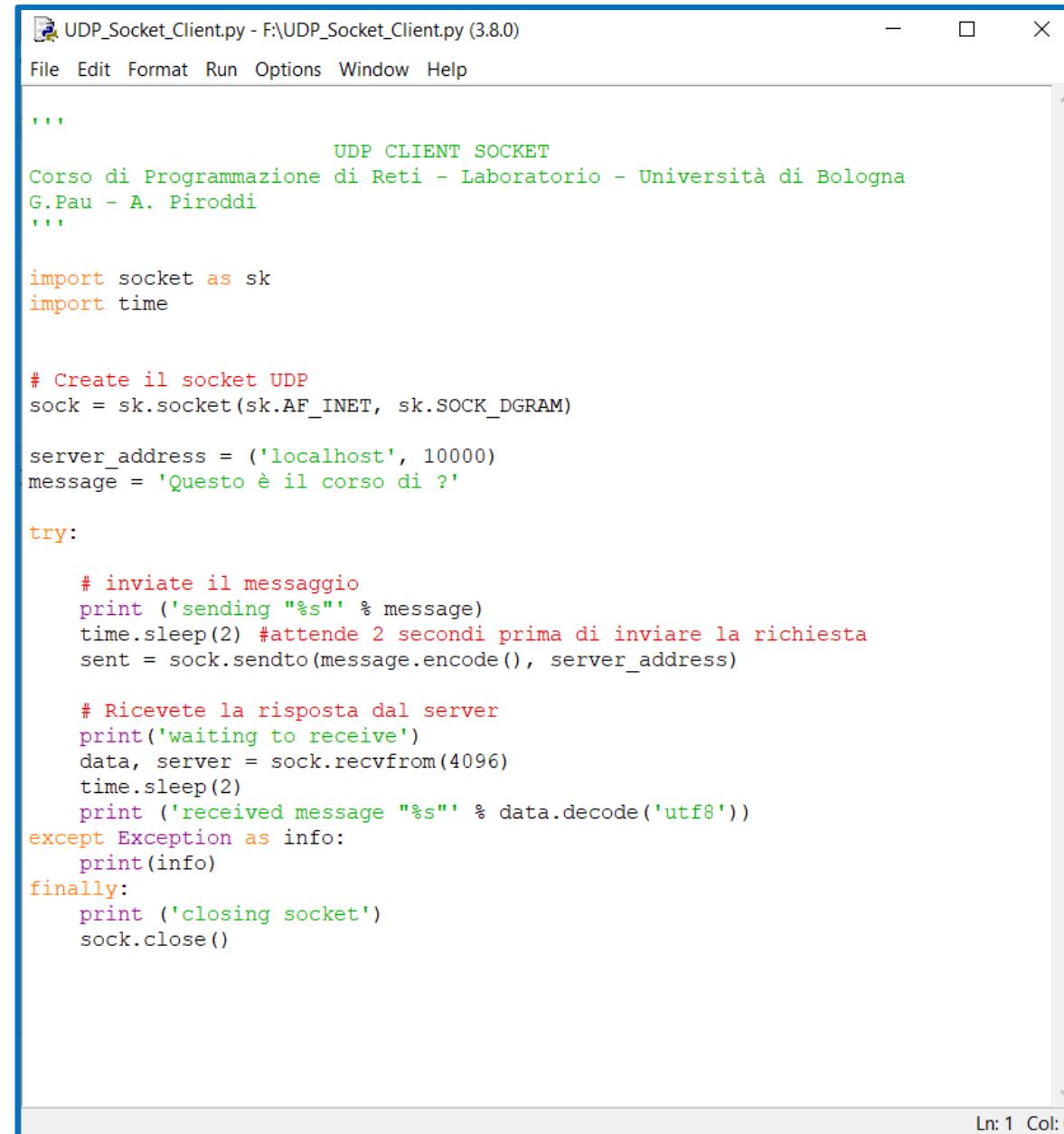


# **UDP**

# **CLIENT SOCKET**



# UDP CLIENT SOCKET



The screenshot shows a Windows-style code editor window titled "UDP\_Socket\_Client.py - F:\UDP\_Socket\_Client.py (3.8.0)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code itself is a Python script for a UDP client. It starts with a header block containing three dots, the title "UDP CLIENT SOCKET", the course information "Corso di Programmazione di Reti - Laboratorio - Università di Bologna", and the authors "G.Pau - A. Piroddi". Below this is another set of three dots. The script then imports the socket module and time module. It creates a UDP socket using sk.socket(sk.AF\_INET, sk.SOCK\_DGRAM). It defines a server address as ('localhost', 10000) and a message as 'Questo è il corso di ?'. It uses a try block to handle the socket operations. Inside the try block, it prints the message to be sent, waits 2 seconds, sends the message encoded to the server, and receives a response from the server. It then prints the received message decoded from utf8. If an exception occurs, it prints the info. Finally, it prints a closing message and closes the socket. The status bar at the bottom right shows "Ln: 1 Col: 0".

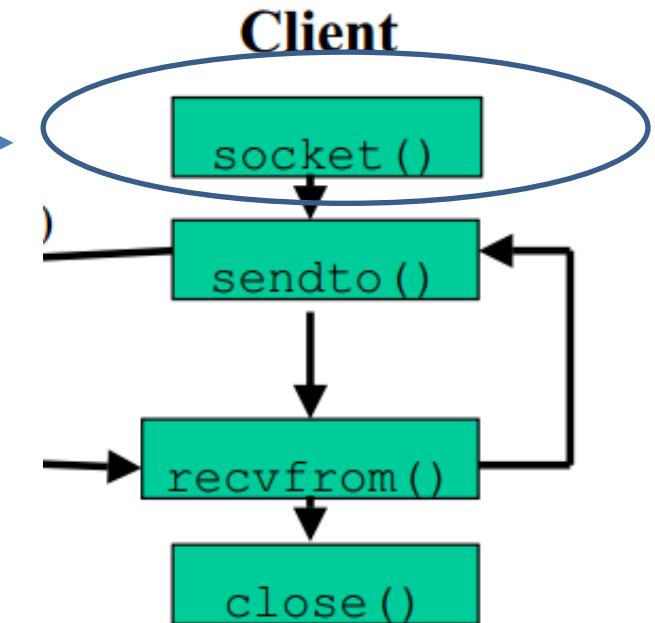
```
'''  
UDP CLIENT SOCKET  
Corso di Programmazione di Reti - Laboratorio - Università di Bologna  
G.Pau - A. Piroddi  
'''  
  
import socket as sk  
import time  
  
# Create il socket UDP  
sock = sk.socket(sk.AF_INET, sk.SOCK_DGRAM)  
  
server_address = ('localhost', 10000)  
message = 'Questo è il corso di ?'  
  
try:  
  
    # inviate il messaggio  
    print ('sending "%s"' % message)  
    time.sleep(2) #attende 2 secondi prima di inviare la richiesta  
    sent = sock.sendto(message.encode(), server_address)  
  
    # Ricevete la risposta dal server  
    print('waiting to receive')  
    data, server = sock.recvfrom(4096)  
    time.sleep(2)  
    print ('received message "%s"' % data.decode('utf8'))  
except Exception as info:  
    print(info)  
finally:  
    print ('closing socket')  
    sock.close()  
  
Ln: 1 Col: 0
```



# UDP CLIENT SOCKET

```
# Create il socket UDP  
sock = sk.socket(sk.AF_INET, sk.SOCK_DGRAM)
```

Creiamo il socket UDP per l'invio della richiesta e lo associamo alla variabile `sock`



## UDP CLIENT SOCKET

Associamo alla variabile *server\_address* la tupla che contiene l'indirizzo e la porta **udp** del server.

```
server_address = ('localhost', 10000)  
message = 'Questo è il corso di ?'
```

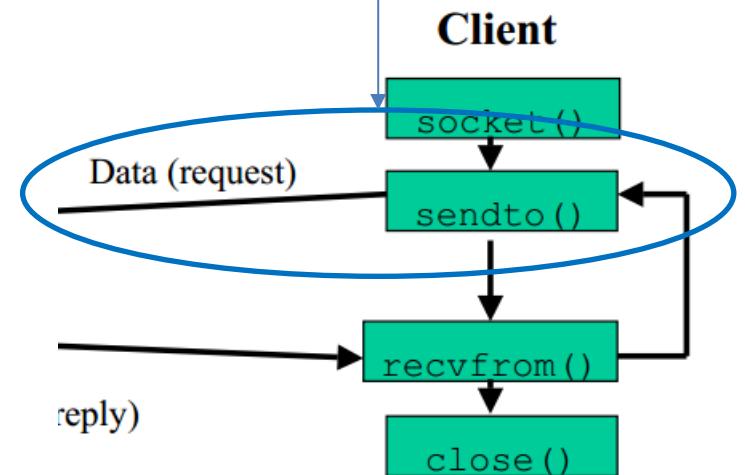
Prepariamo la variabile *message* assegnandole la stringa che vogliamo inviare al server



# UDP CLIENT SOCKET

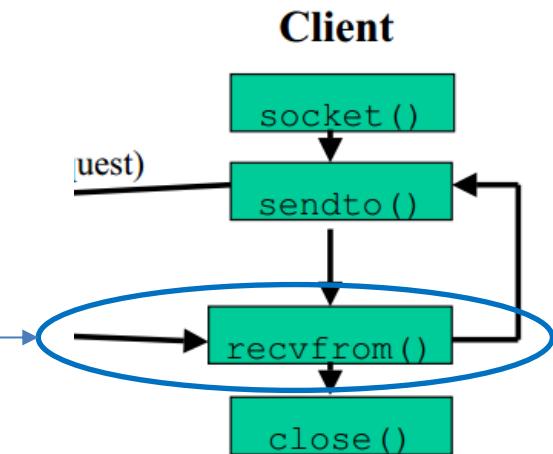
```
sent = sock.sendto(message.encode(), server_address)
```

Inviamo, tramite il modulo socket (*sock.sendto()*), il messaggio all'indirizzo del server (ipaddress e udp port) e associamo il contenuto di alla variabile *sent*



## UDP CLIENT SOCKET

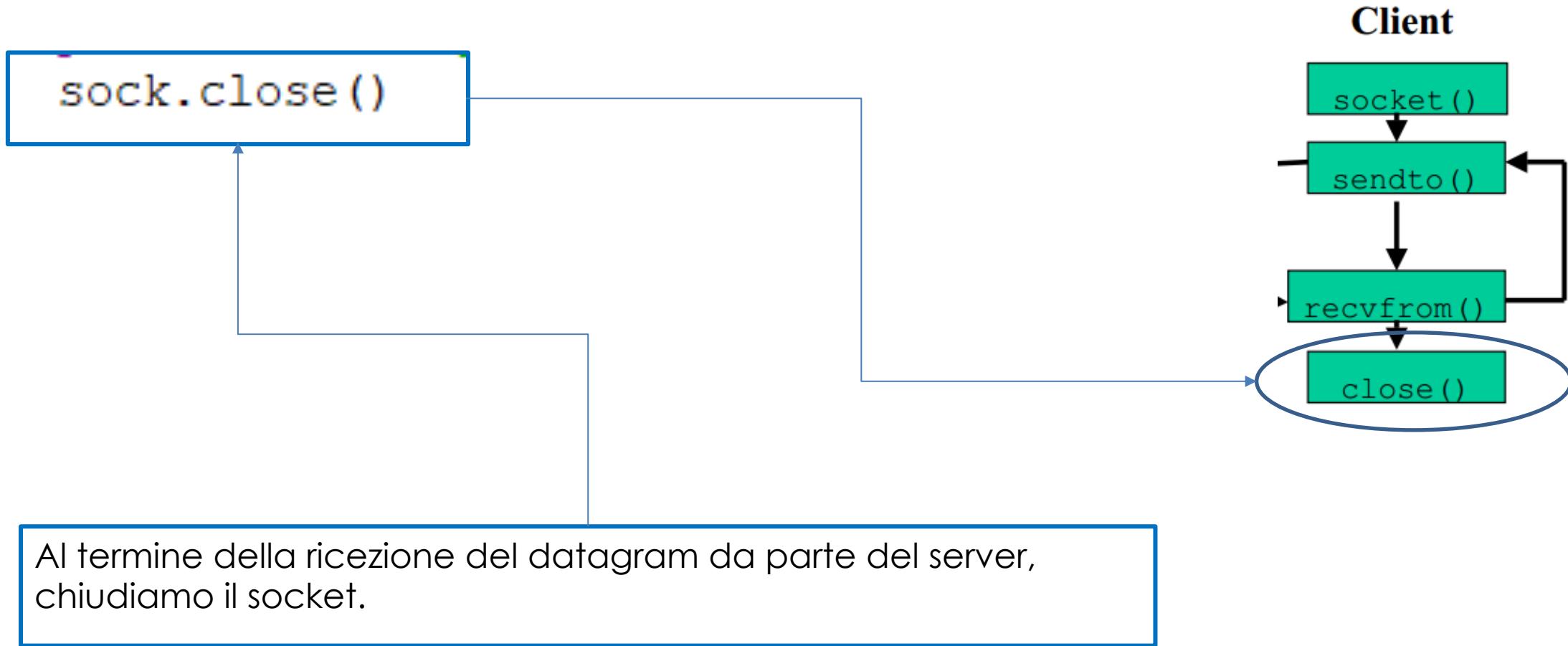
```
data, server = sock.recvfrom(4096)
```



Appena riceviamo il contenuto del messaggio proveniente dal server tramite il modulo socket `sock.recvfrom()` ne assegniamo il contenuto alle due variabili `data` e `server`, corrispondenti rispettivamente al PAYLOAD e all'indirizzo del server (ip address e porta) da cui esso proviene.



# UDP CLIENT SOCKET



# Confronto UDP e TCP Client

```
UDP_Socket_Client.py - F:\UDP_Socket_Client.py (3.8.0)
File Edit Format Run Options Window Help
"""
        UDP CLIENT SOCKET
Corso di Programmazione di Reti - Laboratorio - Università di Bologna
G.Pau - A. Piroddi
"""

import socket as sk
import time
# Create il socket UDP
1
sock = sk.socket(sk.AF_INET, sk.SOCK_DGRAM)

server_address = ('localhost', 10000)
message = 'Questo è il corso di ?'

try:
    # inviate il messaggio
    3
    print ('sending "%s"' % message)
    time.sleep(2) #attende 2 secondi prima di inviare la richiesta
    sent = sock.sendto(message.encode(), server_address)

    # Ricevete la risposta dal server
    4
    print('waiting to receive from')
    data, server = sock.recvfrom(4096)
    #print(server)
    time.sleep(2)
    print ('received message "%s"' % data.decode('utf8'))
except Exception as info:
    print(info)
finally:
    5
    print ('closing socket')
    sock.close()

TCP_Socket_Client.py - F:\TCP_Socket_Client.py (3.8.0)
File Edit Format Run Options Window Help
"""
Corso di Programmazione di Reti - Laboratorio - Università di Bologna
Socket_Programming_Assignment - WebServer - G.Pau - A. Piroddi

Per eseguire il presente codice è necessario utilizzare o una Command Prompt o d

"""

1
import socket as sk
import sys

clientsocket = sk.socket(sk.AF_INET, sk.SOCK_STREAM)

if len(sys.argv) != 4:
    print (len(sys.argv))
    print("Il comando non è corretto. Usa il seguente formato: client.py indiriz
sys.exit(0)

host = str(sys.argv[1])
port = int(sys.argv[2])
request = str(sys.argv[3])
2
request = "GET /" + request + " HTTP/1.1"
try:
    clientsocket.connect((host,port))
except Exception as data:
    print (Exception,":",data)
    print ("Ritenta sarai più fortunato.\r\n")
    sys.exit(0)
clientsocket.send(request.encode()) #se tutto è andato bene, rispondiamo al serv
print(request.encode())
4
response = clientsocket.recv(1024)

print (response)
5
clientsocket.close()

Ln: 36 Col: 0
Ln: 21 Col: 0
```



# WIRESHARK



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA  
CAMPUS DI CESENA

# ANALIZZATORE di PROTOCOLLO - WIRESHARK

Ai link seguenti potete trovare il manuale di Wireshark e i pacchetti di installazione.

Manuale Wireshark:

<https://www.wireshark.org/download/docs/user-guide.pdf>

Download Pacchetto installativo:

<https://www.wireshark.org/download.html>

## Stable Release (3.2.2)

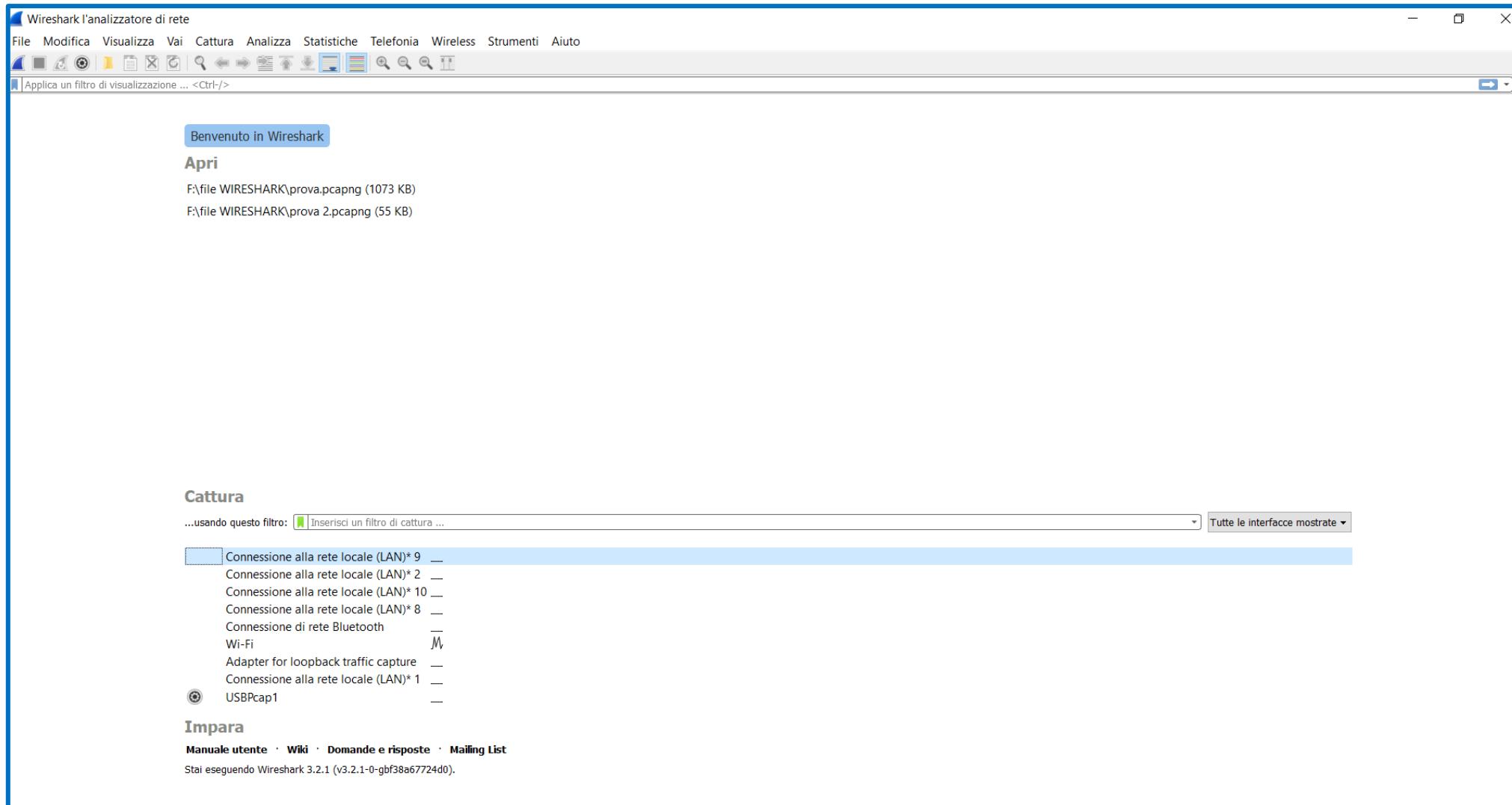
-  [Windows Installer \(64-bit\)](#)
- [Windows Installer \(32-bit\)](#)
- [Windows PortableApps® \(32-bit\)](#)
- [macOS Intel 64-bit .dmg](#)
- [Source Code](#)

VENDOR / PLATFORM	SOURCES
Alpine / Alpine Linux	<a href="#">Standard package</a>
Apple / macOS	<a href="#">Homebrew (Formula)</a> <a href="#">MacPorts</a> <a href="#">Fink</a>
Arch Linux / Arch Linux	<a href="#">Standard package</a>
Canonical / Ubuntu	<a href="#">Standard package</a> <a href="#">Latest stable PPA</a>
Debian / Debian GNU/Linux	<a href="#">Standard package</a>
The FreeBSD Project / FreeBSD	<a href="#">Standard package</a>
Gentoo Foundation / Gentoo Linux	<a href="#">Standard package</a>
HP / HP-UX	<a href="#">Porting And Archive Centre for HP-UX</a>
NetBSD Foundation / NetBSD	<a href="#">Standard package</a>
Novell / openSUSE, SUSE Linux	<a href="#">Standard package</a>
Offensive Security / Kali Linux	<a href="#">Standard package</a>
PCLinuxOS / PCLinuxOS	<a href="#">Standard package</a>
Red Hat / Fedora	<a href="#">Standard package</a>
Red Hat / Red Hat Enterprise Linux	<a href="#">Standard package</a>
Slackware Linux / Slackware	<a href="#">SlackBuilds.org</a>
Oracle / Solaris 11	<a href="#">Standard package</a>
* / *	<a href="#">The Written Word</a>



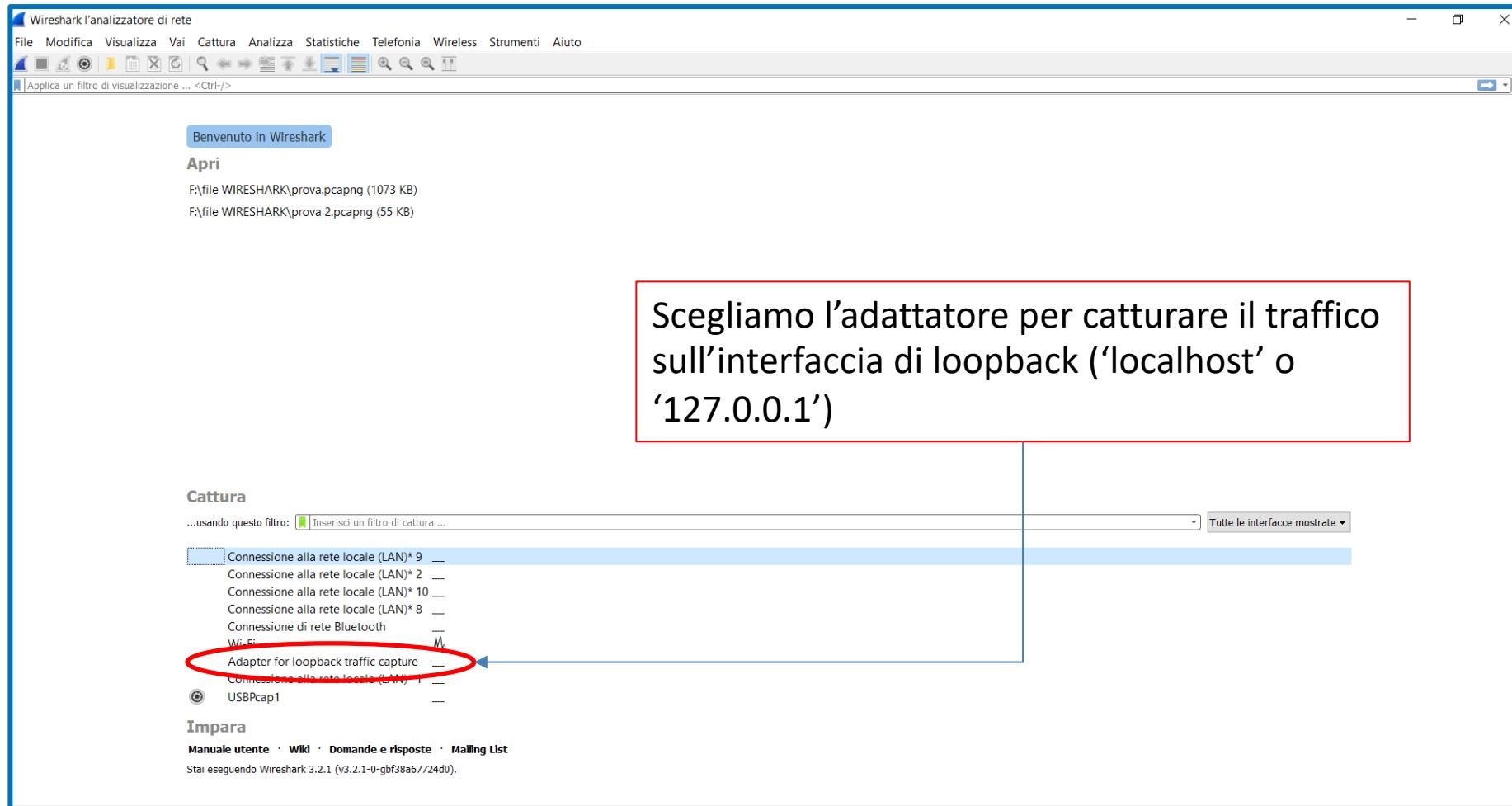
# ANALIZZATORE di PROTOCOLLO - WIRESHARK

L'interfaccia grafica si presenta così:

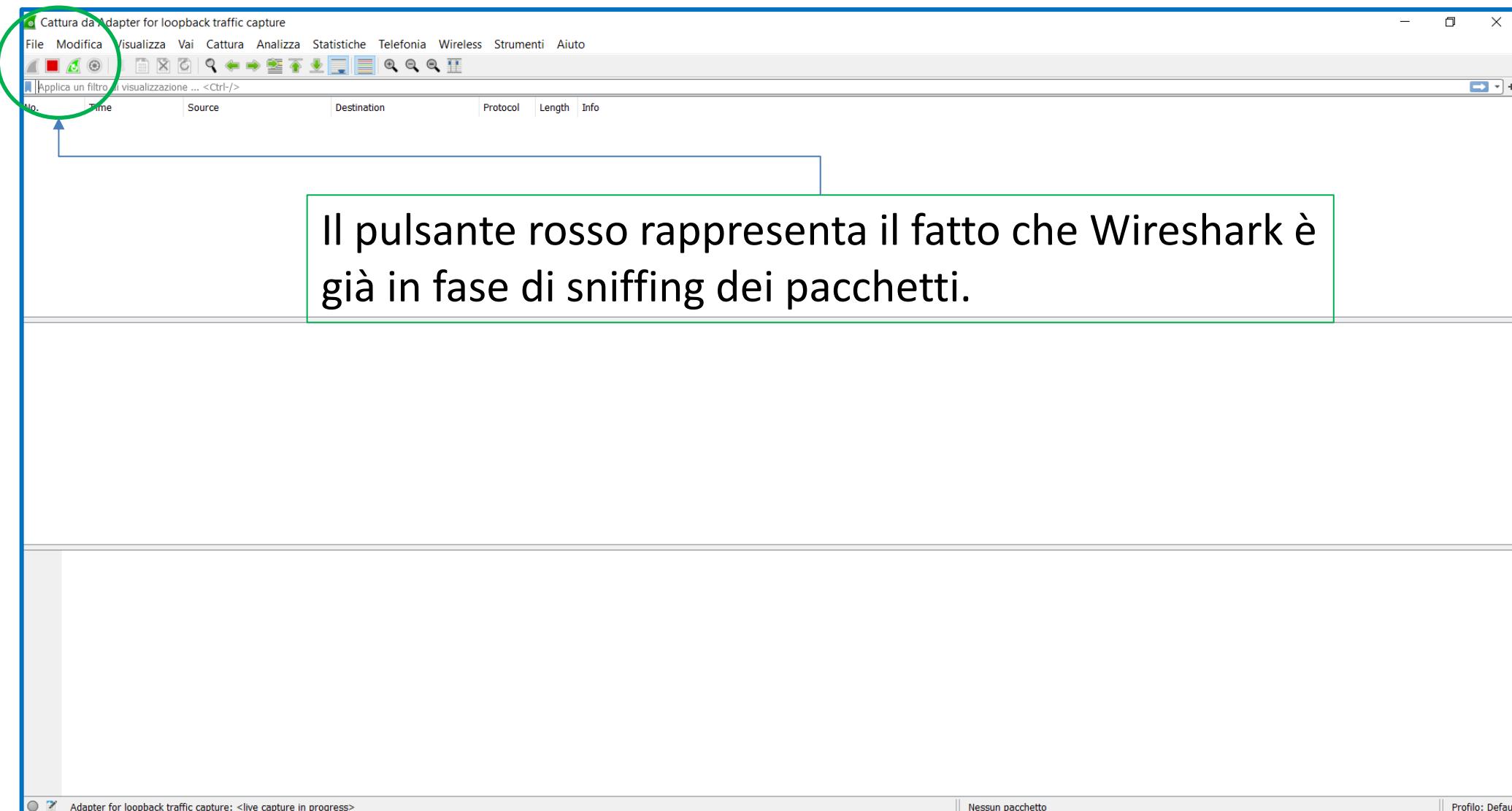


# ANALIZZATORE di PROTOCOLLO - WIRESHARK

Selezionare l'interfaccia di rete su cui si vuole effettuare lo sniffing dei pacchetti



# ANALIZZATORE di PROTOCOLLO - WIRESHARK



# ANALIZZATORE di PROTOCOLLO - WIRESHARK

Wireshark è attivo e sta sniffando il traffico che proviene ed è destinato alla interfaccia di loopback.

Lanciate quindi l'UDP\_Socket\_Server.py e successivamente l'UDP\_Socket\_Client.py

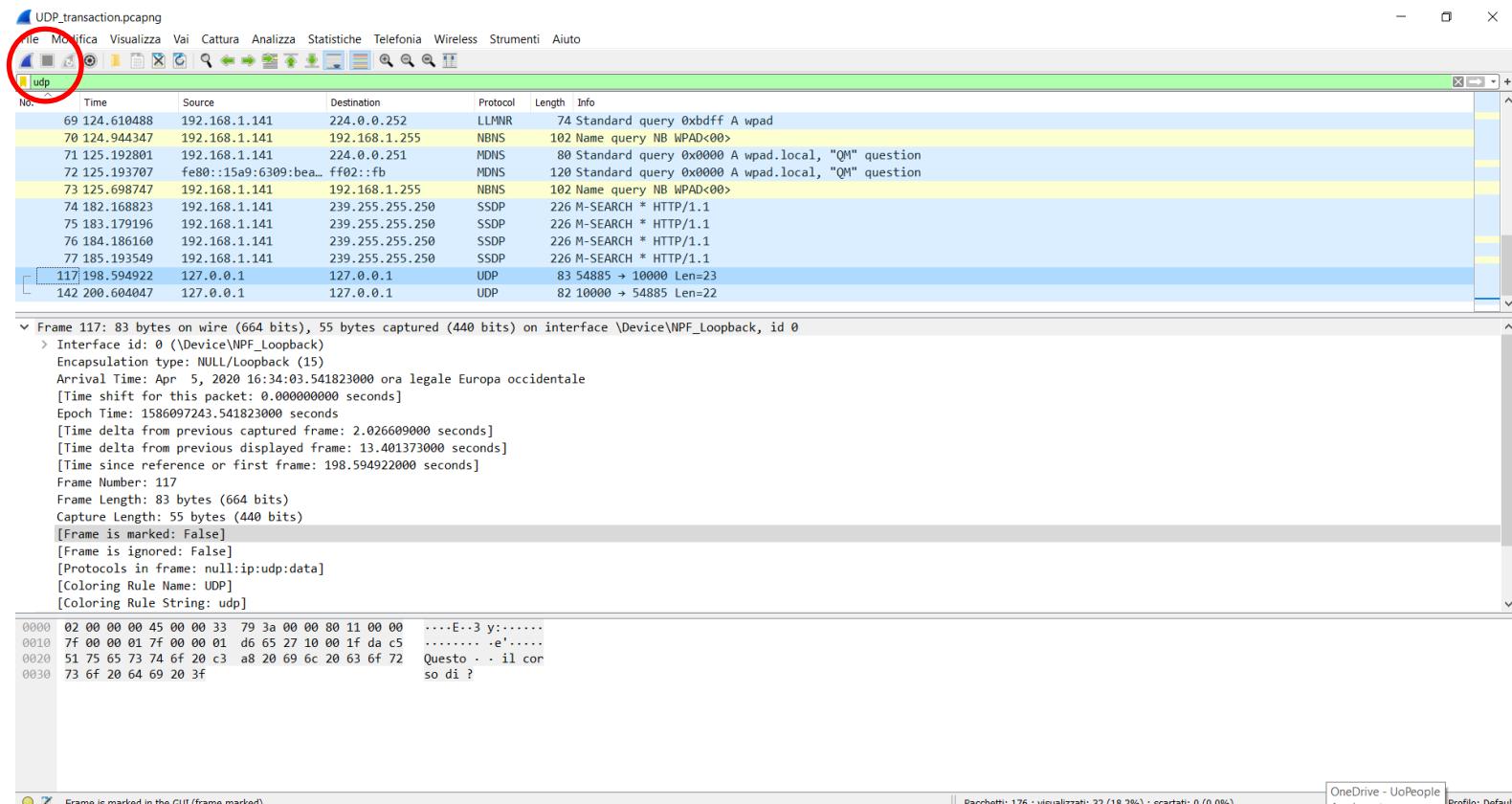
Comincerete a vedere nell'interfaccia di Wireshark alcune righe informative del tipo:

The screenshot shows the Wireshark interface with a file named "UDP\_transaction.pcapng". The packet list pane displays several TCP and UDP packets. A red box highlights a UDP packet at frame 117, which is a User Datagram Protocol (UDP) packet from source port 54885 to destination port 10000. The red box also covers the detailed and bytes panes below it, which show the raw hex and ASCII data of the UDP payload. The ASCII dump shows the string "Questo . . il cor so di ?". Other TCP packets are visible in the list, such as frames 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, and 120, all showing typical ACK and PSH sequences between hosts 196.55.0544 and 127.0.0.1.



# ANALIZZATORE di PROTOCOLLO - WIRESHARK

Stoppiamo lo sniffing,  
E salviamo il file in modo da averlo disponibile per successive analisi.



# ANALIZZATORE di PROTOCOLLO - WIRESHARK

Visualizziamo il solo traffico UDP.

NOTA: stiamo solo filtrando la visualizzazione non stiamo filtrando il traffico catturato, cosa che invece è possibile fare utilizzando i filtri di cattura.

Analizziamo la richiesta del client

No.	Time	Source	Destination	Protocol	Length	Info
69	124.610488	192.168.1.141	224.0.0.252	LLMNR	74	Standard query 0xbdff A wpad
70	124.944347	192.168.1.141	192.168.1.255	NBNS	102	Name query NB WPAD<00>
71	125.192801	192.168.1.141	224.0.0.251	MDNS	80	Standard query 0x0000 A wpad.local, "QM" question
72	125.193707	fe80::15a9:6309:bea... ff02::fb		MDNS	120	Standard query 0x0000 A wpad.local, "QM" question
73	125.698747	192.168.1.141	192.168.1.255	NBNS	102	Name query NB WPAD<00>
74	182.168823	192.168.1.141	239.255.255.250	SSDP	226	M-SEARCH * HTTP/1.1
75	183.179196	192.168.1.141	239.255.255.250	SSDP	226	M-SEARCH * HTTP/1.1
76	184.186160	192.168.1.141	239.255.255.250	SSDP	226	M-SEARCH * HTTP/1.1
77	185.193549	192.168.1.141	239.255.255.250	SSDP	226	M-SEARCH * HTTP/1.1
117	198.594922	127.0.0.1	127.0.0.1	UDP	83	54885 → 10000 Len=23
142	200.604047	127.0.0.1	127.0.0.1	UDP	82	10000 → 54885 Len=22

Frame 117: 83 bytes on wire (664 bits), 55 bytes captured (440 bits) on interface \Device\NPF\_Loopback, id 0

> Interface id: 0 (\Device\NPF\_Loopback)  
Encapsulation type: NULL/Loopback (15)  
Arrival Time: Apr 5, 2020 16:34:03.541823000 ora legale Europa occidentale  
[Time shift for this packet: 0.000000000 seconds]  
Epoch Time: 1586097243.541823000 seconds  
[Time delta from previous captured frame: 2.026609000 seconds]  
[Time delta from previous displayed frame: 13.401373000 seconds]  
[Time since reference or first frame: 198.594922000 seconds]  
Frame Number: 117  
Frame Length: 83 bytes (664 bits)  
Capture Length: 55 bytes (440 bits)  
[Frame is marked: False]  
[Frame is ignored: False]  
[Protocols in frame: null:ip:udp:data]  
[Coloring Rule Name: UDP]  
[Coloring Rule String: udp]

0000 02 00 00 00 45 00 00 33 79 3a 00 00 80 11 00 00 ....E..3 y:.....  
0010 7f 00 00 01 0f 00 00 01 d6 65 27 10 00 1f da c5 ..... e'.....  
0020 51 75 65 73 74 6f 20 c3 a8 20 69 6c 20 63 6f 72 Questo . . il cor  
0030 73 6f 20 64 69 20 3f so di ?

Frame is marked in the GUI (frame.marked)

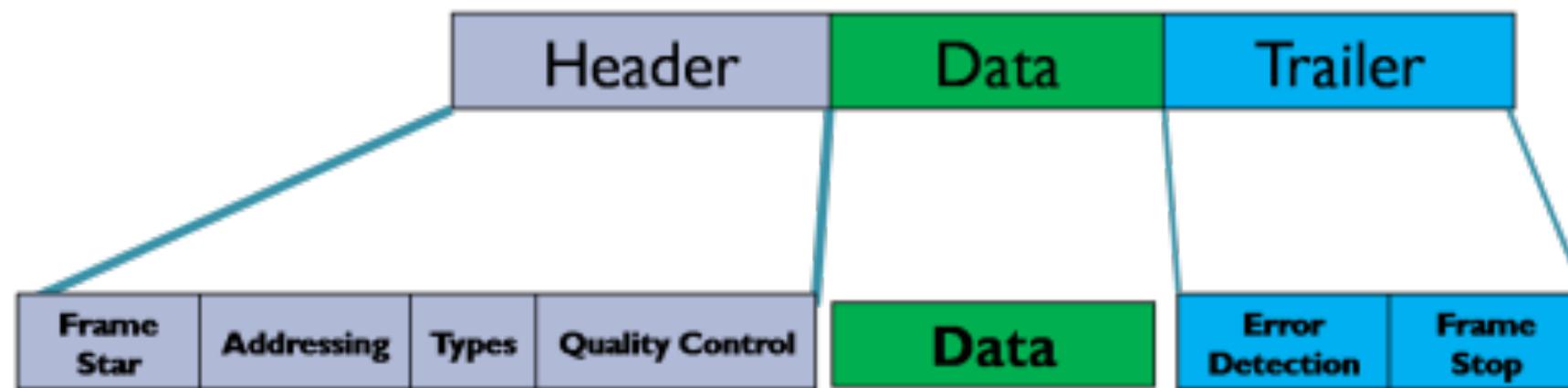
Pacchetti: 176 · visualizzati: 32 (18.2%) · scartati: 0 (0.0%)

OneDrive - UoPeople | Aggiornato | Profilo: Default



# ANALIZZATORE di PROTOCOLLO – WIRESHARK - FRAME

Il livello data link (Collegamento) si occupa di fornire ai livelli superiori una linea di comunicazione esente da errori di trasmissione non segnalati; per fare questo decompone i dati del mittente in pacchetti chiamati frame, composti da alcune centinaia o migliaia di byte, e li spedisce in sequenza attendendo eventualmente la conferma di avvenuta ricezione da parte del destinatario.



# ANALIZZATORE di PROTOCOLLO – WIRESHARK - FRAME

(15) È un valore interno di Wireshark che rappresenta il particolare tipo di intestazione del livello di collegamento per il pacchetto in questione e i valori numerici possono differire da una versione all'altra.

Frame 117: 83 bytes on wire (664 bits), 55 bytes captured (440 bits) on interface \Device\NPF\_Loopback, id 0

> Interface id: 0 (\Device\NPF\_Loopback)

Encapsulation type: NULL/Loopback (15)

Arrival Time: Apr 5, 2020 16:34:03.541823000 ora legale Europa occidentale

[Time shift for this packet: 0.000000000 seconds]

Epoch Time: 1586097243.541823000 seconds

[Time delta from previous captured frame: 2.026609000 seconds]

[Time delta from previous displayed frame: 2.026609000 seconds]

[Time since reference or first frame: 198.594922000 seconds]

Frame Number: 117

Frame Length: 83 bytes (664 bits)

Capture Length: 55 bytes (440 bits)

[Frame is marked: False]

[Frame is ignored: False]

[Protocols in frame: null:ip:udp:data]

[Coloring Rule Name: UDP]

[Coloring Rule String: udp]

Epoch Time (noto anche come tempo UNIX) è il numero di secondi dal 1 ° gennaio 1970. Questo è ciò che è effettivamente memorizzato nel file .pcap o .pcapng. Gli altri formati di tempo in Wireshark sono conversioni del Epoch Time a scopo di visualizzazione.

**Frame is Marked: False** - Wireshark ci permette di "contrassegnare" una frame; vedete «Marca / Deseleziona pacchetto» nel menu "Modifica". "Il frame è contrassegnato: False" significa che il frame non è stato "contrassegnato".

**Frame is Ignored: False** - Wireshark ci permette anche di "ignorare" un pacchetto; se «Ignora/Considera Pacchetto» nel menu "Modifica". "Frame ignorato: False" significa che il frame non è stato "ignorato".



# ANALIZZATORE di PROTOCOLLO – WIRESHARK - Networking



Il protocollo "null" è il protocollo a livello di collegamento utilizzato sull'interfaccia di loopback sulla maggior parte dei sistemi operativi BSD.

È chiamato impropriamente «null», in quanto l'intestazione del livello di collegamento non è «nulla»; l'intestazione del livello di collegamento è un numero intero di 4 byte, nell'ordine di byte nativo della macchina su cui viene acquisito il traffico, contenente un valore "famiglia di indirizzi" / "famiglia di protocollo" per il protocollo in esecuzione sul livello di collegamento, ad esempio AF\_INET per IPv4 e AF\_INET6 per IPv6. **AF\_INET** è 2 su tutti i sistemi operativi basati su BSD (Berkeley Sockets - <http://www.on-time.com/rtos-32-docs/rtip-32/programming-manual/programming-with/berkeley-socket-api.htm>)



# ANALIZZATORE di PROTOCOLLO – WIRESHARK - Networking

Il campo Identificazione è semplicemente un ID univoco applicato a ciascun pacchetto che un host invia su una determinata connessione. È generalmente utile solo se un pacchetto deve essere frammentato (diciamo da un router) - ogni frammento manterrà l'identificazione originale. Permette all'host ricevente di sapere come riassemblare i frammenti.

## Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

0100 .... = Version: 4

.... 0101 = Header Length: 20 bytes (5)

> Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)

Total Length: 51

Identification: 0x793a (31034)

> Flags: 0x0000

...0 0000 0000 0000 = Fragment offset: 0

Time to live: 128

Protocol: UDP (17)

Header checksum: 0x0000 [validation disabled]

[Header checksum status: Unverified]

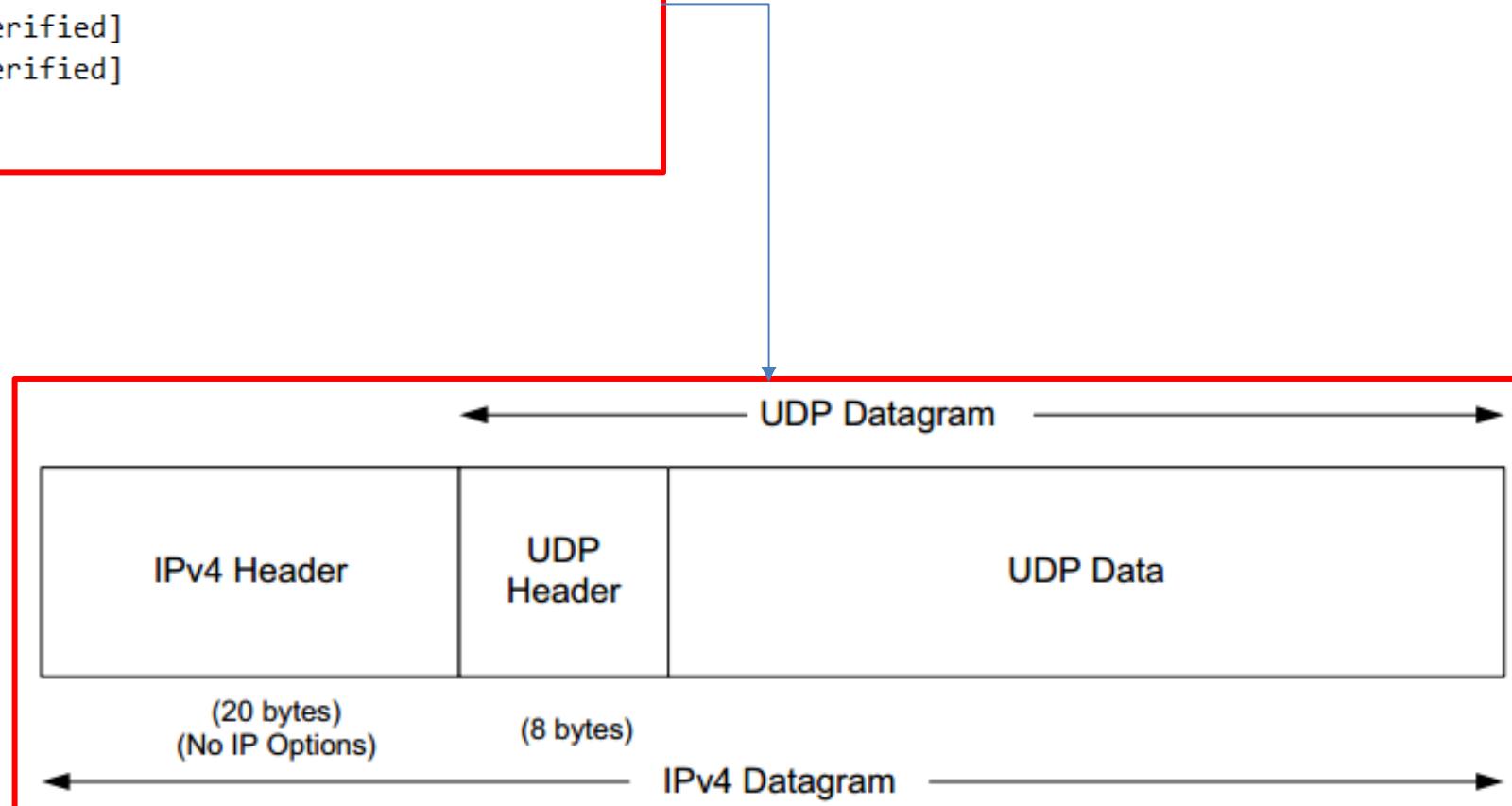
Source: 127.0.0.1

Destination: 127.0.0.1



# ANALIZZATORE di PROTOCOLLO – WIRESHARK - TRASPORTO

▼ User Datagram Protocol, Src Port: 54885, Dst Port: 10000  
Source Port: 54885  
Destination Port: 10000  
Length: 31  
Checksum: 0xdac5 [unverified]  
[Checksum Status: Unverified]  
[Stream index: 9]  
➤ [Timestamps]



# ANALIZZATORE di PROTOCOLLO – WIRESHARK - DATA

```
▼ Data (23 bytes)
Data: 51756573746f20c3a820696c20636f72736f206469203f
[Length: 23]

0000 02 00 00 00 45 00 00 33 79 3a 00 00 80 11 00 00  ....E..3 y:.....
0010 7f 00 00 01 7f 00 00 01 d6 65 27 10 00 1f da c5  ....e.....
0020 51 75 65 73 74 6f 20 c3 a8 20 69 6c 20 63 6f 72 Questo .. il cor
0030 73 6f 20 64 69 20 3f so di ?
```

Se provate ad accedere al seguente link

<https://onlineutf8tools.com/convert-bytes-to-utf8>



**bytes**

0000 51 75 65 73 74 6f 20 c3 a8 20 69 6c 20 63 6f 72  
0010 73 6f 20 64 69 20 3f

Import from file Save as... Copy to clipboard

**utf8**

! Output might be incorrect  
Base not set: assuming input of hexadecimal base.

Questo è il corso di ?

Chain with... Save as... Copy to clipboard



# ANALIZZATORE di PROTOCOLLO - WIRESHARK

UDP\_transaction.pcapng

File Modifica Visualizza Vai Cattura Analizza Statistiche Telefonia Wireless Strumenti Aiuto

udp

No.	Time	Source	Destination	Protocol	Length	Info
69	124.610488	192.168.1.141	224.0.0.252	LLMNR	74	Standard query 0xbdff A wpad
70	124.944347	192.168.1.141	192.168.1.255	NBNS	102	Name query NB WPAD<00>
71	125.192801	192.168.1.141	224.0.0.251	MDNS	80	Standard query 0x0000 A wpad.local, "QM" question
72	125.193707	fe80::15a9:6309:bea... ff02::fb		MDNS	120	Standard query 0x0000 A wpad.local, "QM" question
73	125.698747	192.168.1.141	192.168.1.255	NBNS	102	Name query NB WPAD<00>
74	182.168823	192.168.1.141	239.255.255.250	SSDP	226	M-SEARCH * HTTP/1.1
75	183.179196	192.168.1.141	239.255.255.250	SSDP	226	M-SEARCH * HTTP/1.1
76	184.186160	192.168.1.141	239.255.255.250	SSDP	226	M-SEARCH * HTTP/1.1
77	185.193549	192.168.1.141	239.255.255.250	SSDP	226	M-SEARCH * HTTP/1.1
117	198.504022	127.0.0.1	127.0.0.1	UDP	83	54885 > 10000 Len=23
142	200.604047	127.0.0.1	127.0.0.1	UDP	82	10000 > 54885 Len=22

> Frame 142: 82 bytes on wire (656 bits), 54 bytes captured (432 bits) on interface \Device\NPF\_Loopback, id 0

Null/Loopback

Family: IP (2)

> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

> User Datagram Protocol, Src Port: 10000, Dst Port: 54885

> Data (22 bytes)

0000 02 00 00 00 45 00 00 32 79 53 00 00 80 11 00 00 ...E..2 yS....

0010 7f 00 00 01 7f 00 00 01 27 10 d6 65 00 1e da ee .....'..e....

0020 50 72 6f 67 72 61 6d 6d 61 7a 69 6f 6e 65 20 64 Programm azione d

0030 69 20 52 65 74 69 i Reti

Data (data.data), 22 byte

Pacchetti: 176 · visualizzati: 32 (18.2%) · scartati: 0 (0.0%)

Profilo: Default

Analizziamo la risposta del server



# ANALIZZATORE di PROTOCOLLO - WIRESHARK

```
L 142 200.604047 127.0.0.1           127.0.0.1           UDP      82 10000 → 54885 Len=22
> Frame 142: 82 bytes on wire (656 bits), 54 bytes captured (432 bits) on interface \Device\NPF_Loopback, id 0
  ▼ Null/Loopback
    Family: IP (2)
      > Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
      ▼ User Datagram Protocol, Src Port: 10000, Dst Port: 54885
        Source Port: 10000
        Destination Port: 54885
        Length: 30
        Checksum: 0xdaee [unverified]
        [Checksum Status: Unverified]
        [Stream index: 9]
        > [Timestamps]
      ▼ Data (22 bytes)
        Data: 50726f6772616d6d617a696f6e652064692052657469
        [Length: 22]

0000  02 00 00 00 45 00 00 32  79 53 00 00 80 11 00 00  ....E..2 yS.....
0010  7f 00 00 01 7f 00 00 01  27 10 d6 65 00 1e da ee  ....'...e....
0020  50 72 6f 67 72 61 6d 6d  61 7a 69 6f 6e 65 20 64  Programm azione d
0030  69 20 52 65 74 69  .i Reti
```



# E-MAIL



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA  
CAMPUS DI CESENA

## E-Mail

La posta elettronica viene introdotta per la prima volta negli anni '60, tuttavia è diventata disponibile nella struttura attuale a partire dagli anni '70.

### Protocolli utilizzati nei sistemi di posta elettronica

La comunicazione e-mail viene effettuata tramite tre protocolli in generale:

- IMAP
- POP
- SMTP



## E-Mail - IMAP

**IMAP** è l'acronimo di **Internet Mail Access Protocol**.

Questo protocollo viene utilizzato durante la ricezione di un'e-mail.

Quando si utilizza IMAP:

- le e-mail saranno presenti nel server
- Le email non verranno scaricate nella casella di posta dell'utente
- Le email verranno successivamente eliminate dal server. Questo aiuta ad avere meno memoria utilizzata nel computer locale.



## E-Mail - POP

**POP** è l'acronimo di Post Office Protocol.

Anche questo protocollo viene utilizzato per le e-mail in arrivo.

La differenza principale tra i due protocolli (imap e pop) è che POP scarica l'intera e-mail nel computer locale ed elimina i dati sul server una volta scaricati.

Questo è utile quando il server ha poca memoria libera.

La versione corrente di POP è POP3.



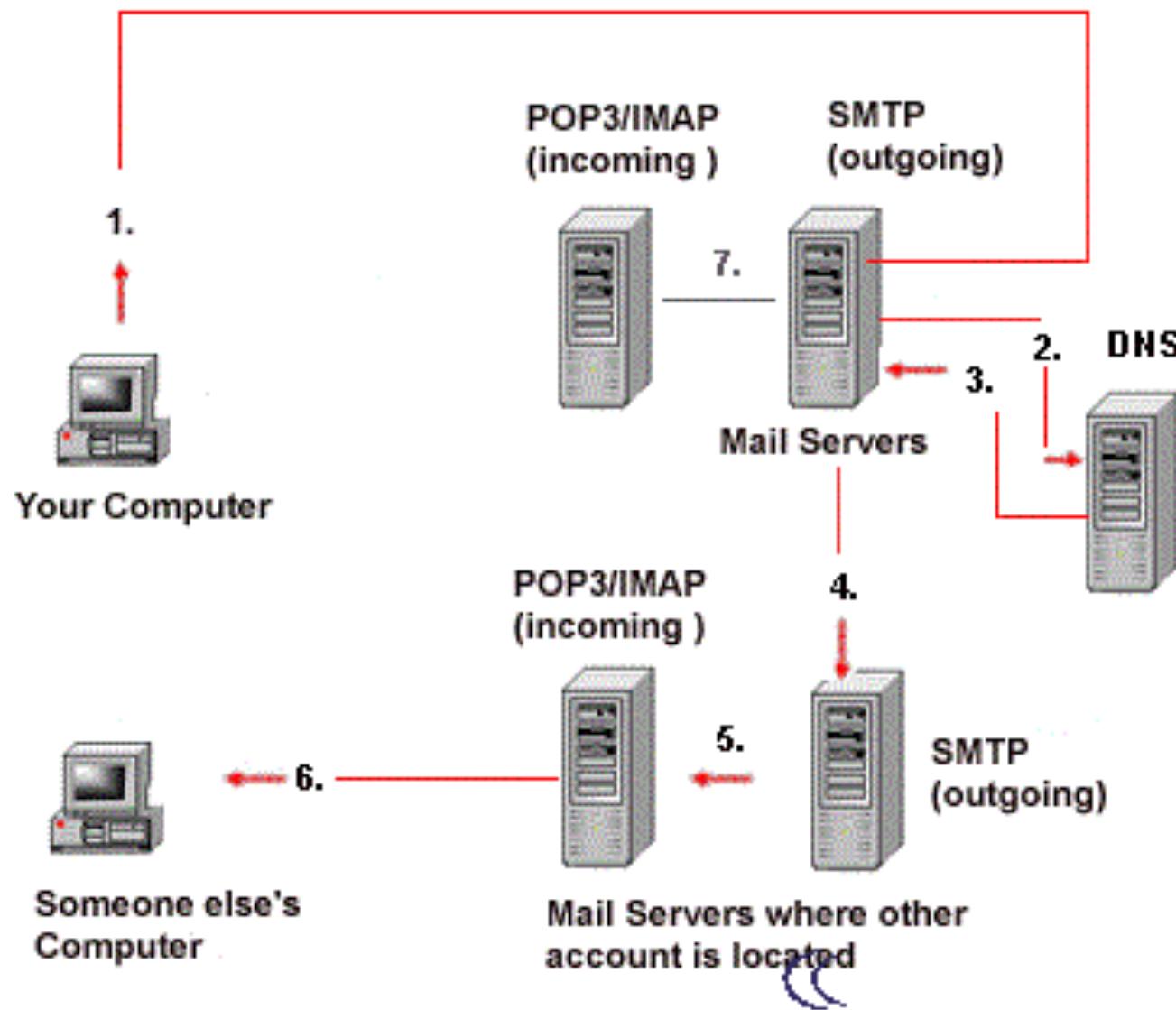
## E-Mail - SMTP

**SMTP** sta per Simple Mail Transfer Protocol.

L'e-mail vengono inviate utilizzando questo protocollo.



# E-Mail - architettura



## E-Mail - Server

Un server di posta è un software.

Il software gestisce la ricezione delle e-mail in arrivo dagli utenti locali (persone all'interno dello stesso dominio) e dai mittenti remoti e inoltra le e-mail in uscita per la consegna.

Un computer su cui è installata un'applicazione di questo tipo può anche essere chiamato server di posta.

Nella figura precedente si vedono due server di posta.

I due server di posta utilizzati per le e-mail in uscita sono chiamati **MTA**, Mail Transfer Agent.

Gli altri due server di posta utilizzati per la posta in arrivo, che utilizzano i protocolli POP3/IMAP, sono chiamati **MDA**, Mail Delivery Agent.



# E-Mail – HOW it Works

Il mittente inserisce l'indirizzo e-mail del destinatario insieme al messaggio utilizzando un'applicazione di posta elettronica (client SMTP). L'E-mail quindi verrà inviata all'MTA (Mail Transfer Agent). Questa comunicazione avviene tramite il protocollo SMTP.

Il passaggio successivo è la ricerca DNS. Il sistema invia una richiesta per conoscere l'MTA corrispondente del destinatario. Questo sarà fatto con l'aiuto del record MX. Nel DNS, in corrispondenza del dominio del destinatario, sarà presente un record MX (Mail Exchanger record). Questo specifica il server di posta di un dominio. Quindi, dopo la ricerca DNS, viene data una risposta al server di posta contenente l'indirizzo IP del server di posta del destinatario.

Il passaggio successivo è il trasferimento del messaggio tra i server di posta (server SMTP). Per questa comunicazione viene utilizzato il protocollo SMTP.

Ora, questo messaggio viene trasferito al MDA di destinazione e quindi viene trasferito al computer locale del destinatario. Qui possono essere utilizzati due protocolli. Se utilizziamo POP3, l'intera e-mail verrà scaricata sul computer locale e la copia sul server verrà eliminata. Se il protocollo utilizzato è IMAP, il messaggio e-mail viene archiviato nel server di posta stesso.

Se si è verificato un errore nell'invio dell'e-mail, le e-mail subiranno un ritardo. C'è una coda di posta in ogni server di posta. Queste mail saranno in sospeso nella coda della posta. Il server di posta continuerà a tentare di inviare nuovamente l'e-mail. Una volta che l'invio dell'e-mail fallisce in modo permanente, il server di posta può inviare un messaggio e-mail di ritorno all'indirizzo e-mail del mittente.



# E-Mail – SMTP comandi e significato

Command	Action taken
HELO	Accepts the greeting from the client and stores it in <code>seen_greeting</code> . Sets server to base command mode.
EHLO	Accepts the greeting from the client and stores it in <code>seen_greeting</code> . Sets server to extended command mode.
NOOP	Takes no action.
QUIT	Closes the connection cleanly.
MAIL	Accepts the “MAIL FROM:” syntax and stores the supplied address as <code>mailfrom</code> . In extended command mode, accepts the <a href="#">RFC 1870</a> SIZE attribute and responds appropriately based on the value of <code>data_size_limit</code> .
RCPT	Accepts the “RCPT TO:” syntax and stores the supplied addresses in the <code>rcpttos</code> list.
RSET	Resets the <code>mailfrom</code> , <code>rcpttos</code> , and <code>received_data</code> , but not the greeting.
DATA	Sets the internal state to DATA and stores remaining lines from the client in <code>received_data</code> until the terminator <code>\r\n.\r\n</code> is received.
HELP	Returns minimal information on command syntax
VRFY	Returns code 252 (the server doesn’t know if the address is valid)
EXPN	Reports that the command is not implemented.



# E-Mail – Esempio client SMTP

```
import smtplib
import email.utils
from email.mime.text import MIMEText

# Creiamo il messaggio mail
msg = MIMEText('Questo è il corpo del messaggio.')
msg['To'] = email.utils.formataddr(( 'DESTINATARIO', 'destinatario@example.com'))
msg['From'] = email.utils.formataddr(( 'MITTENTE', 'mittente@example.com'))
msg['Subject'] = 'Prova Invio Mail'

server = smtplib.SMTP('127.0.0.1', 1027)

server.set_debuglevel(True) # mostriamo le comunicazioni con il server
try:
    server.sendmail('mittente@example.com', [ 'destinatario@example.com'], msg.as_string())
finally:
    server.quit()
```



# E-Mail – server SMTP

Documentazione la trovate: <https://docs.python.org/3/library/smtpd.html>

*peer* è l'indirizzo dell'host remoto

*mailfrom* è l'originatore della mail

*rcpttos* sono i destinatari della mail

```
from datetime import datetime
import asyncore
from smtpd import SMTPServer

class EmlServer(SMTPServer):
    no = 0
    def process_message(self, peer, mailfrom, rcpttos, data, mail_options=None, rcpt_options=None):
        filename = '%s-%d.eml' % (datetime.now().strftime('%Y-%m-%d-%H-%M-%S'),
                                   self.no)
        f = open(filename, 'wb')
        f.write(data)
        f.close
        print ('%s saved.' % filename)
        self.no += 1

def run():
    EmlServer(('127.0.0.1', 1027), None)
    try:
        asyncore.loop()
    except KeyboardInterrupt:
        pass

if __name__ == '__main__':
    run()
```

*data* è una stringa contenente il contenuto dell'e-mail



## E-Mail – esempio

- Lanciate il codice smtp\_server.py
- Lanciate il codice smtp\_client.py

Nella console del client vedremo il seguente scambio di pacchetti

```
send: 'ehlo 87.43.168.192.in-addr.arpa\r\n'
reply: b'250-87.43.168.192.in-addr.arpa\r\n'
reply: b'250-SIZE 33554432\r\n'
reply: b'250-8BITMIME\r\n'
reply: b'250 HELP\r\n'
reply: retcode (250); Msg: b'87.43.168.192.in-addr.arpa\nSIZE 33554432\n8BITMIME\nHELP'
send: 'mail FROM:<mittente@example.com> size=256\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
send: 'rcpt TO:<destinatario@example.com>\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
send: 'data\r\n'
reply: b'354 End data with <CR><LF>.<CR><LF>\r\n'
reply: retcode (354); Msg: b'End data with <CR><LF>.<CR><LF>'
data: (354, b'End data with <CR><LF>.<CR><LF>')
send: b'Content-Type: text/plain; charset="utf-8"\r\nMIME-Version: 1.0\r\nContent-
Transfer-Encoding: base64\r\nTo: DESTINATARIO <destinatario@example.com>\r\nFrom: MITTENTE
<mittente@example.com>\r\nSubject: Prova Invio Mail\r\n\r\n
\nUXVlc3RvIMOoIGlsIGNvcnBvIGRlbCBtZXNzYWdnaw8u\r\n.\r\n.\r\n'
reply: b'250 OK\r\n'
reply: retcode (250); Msg: b'OK'
data: (250, b'OK')
send: 'quit\r\n'
reply: b'221 Bye\r\n'
reply: retcode (221); Msg: b'Bye'
```



# Ritardi di Trasferimento



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA  
CAMPUS DI CESENA

## Esercizio 7 – Ritardi di Trasferimento

---

Un sistema trasmissivo della velocità di 100 kb/s presenta una lunghezza di 600 km tra Tx ed Rx. Fra i due elementi di testa sono presenti due router, ciascuno dei quali presenta una latenza (latency), ossia un tempo di accodamento in uscita, pari al tempo di trasmissione, ed una capacità di 100kb/s. Si consideri trascurabile il tempo di elaborazione.

**Quesiti:**

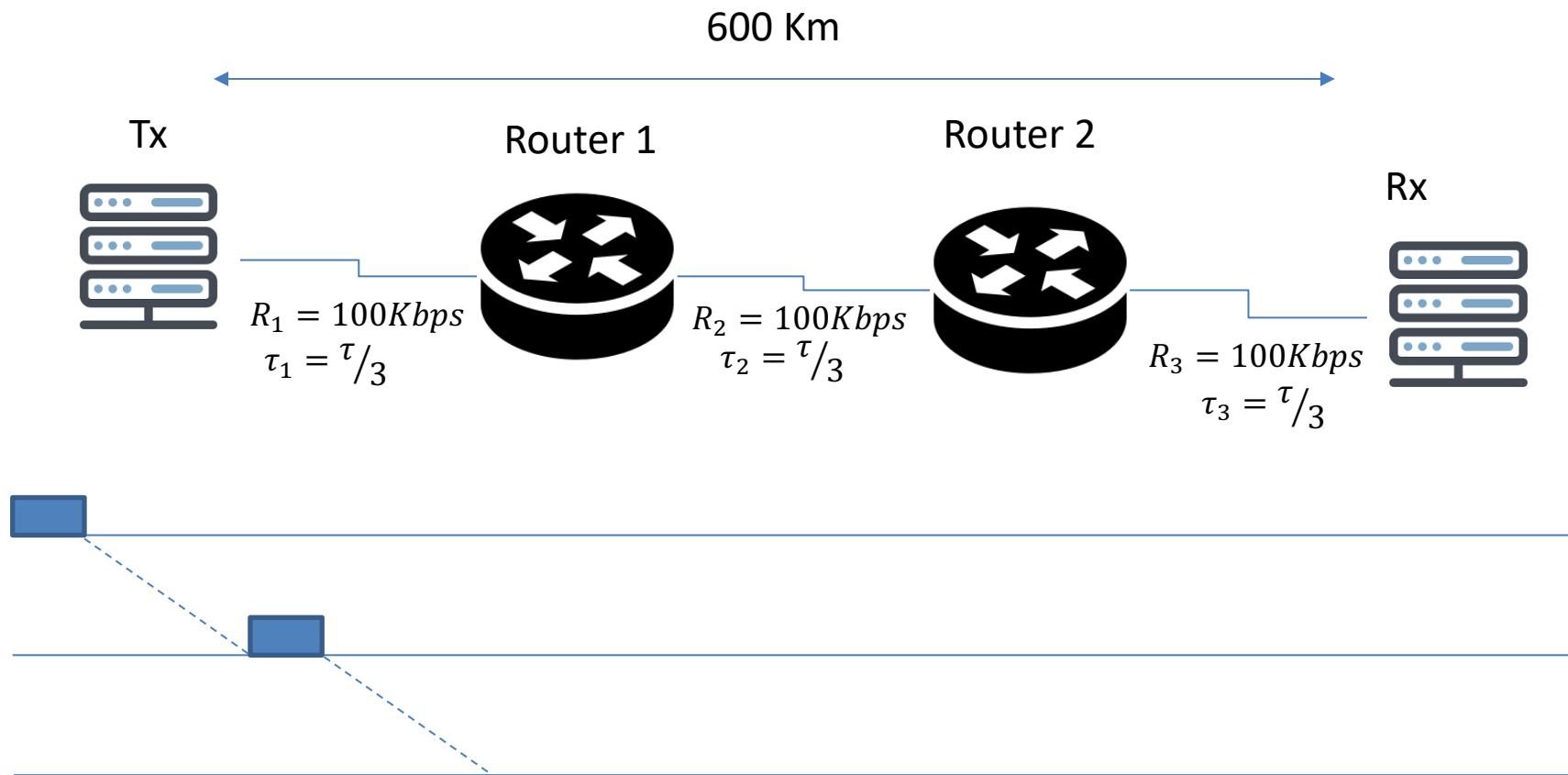
Si chiede di calcolare il ritardo totale nella trasmissione di un pacchetto di 3000 bit, assumendo un ritardo di propagazione di  $5\mu s$ : nei due casi in cui nei router si applichi:

- la modalità *store and forward*
- la modalità *cut-through*, considerando che la lunghezza dell'*header* sia di 200 bit (a parità di dimensione totale del pacchetto).



## Esercizio 7 – Ritardi di Trasferimento – Soluzione 1/5

Scenario: **STORE & FORWARD** → il pacchetto deve essere completamente ricevuto prima di essere ritrasmesso



## Esercizio 7 – Ritardi di Trasferimento – Soluzione 2/5

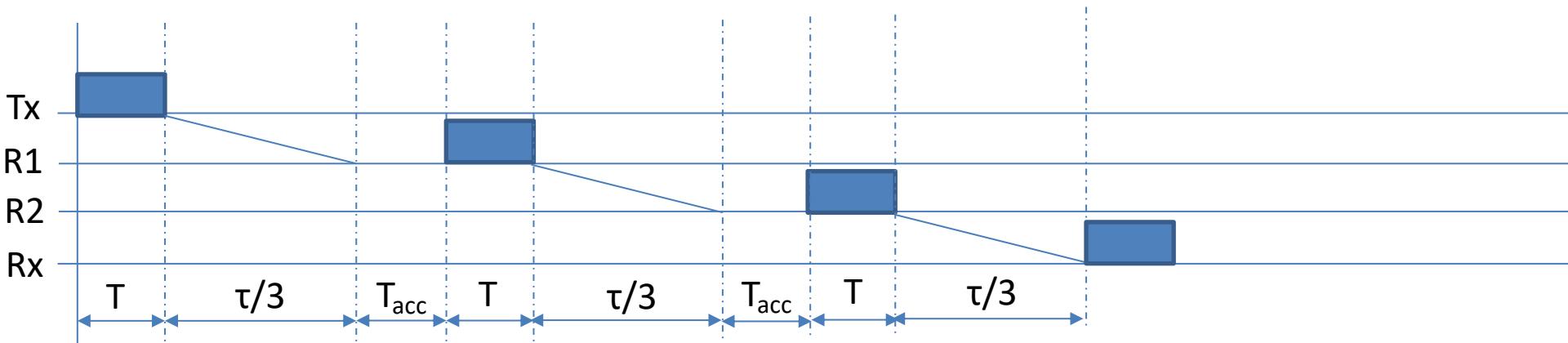
Scenario: STORE & FORWARD → il pacchetto deve essere completamente ricevuto prima di essere ritrasmesso

Osservazioni preliminari:

- Abbiamo 3 elementi trasmissivi quindi 3 tempi di trasmissione →  $3 \cdot T = 3 \cdot \frac{L}{R} = 3 \cdot \frac{3000 \text{ bit}}{100 \cdot 10^3 \text{ bps}} = 3 \cdot 30 \cdot 10^{-3} \text{ s} = 90 \text{ ms}$
- Abbiamo 2 elementi che introducono latenza per accodamento e ritrasmissione →  $2 \cdot T_{acc} = 2 \cdot T = 60 \text{ ms}$
- Abbiamo il tempo di propagazione fisico →  $\tau = \frac{5 \mu\text{s}}{\text{Km}} \cdot 600 \text{ Km} = 3000 \mu\text{s} = 3.0 \text{ ms}$

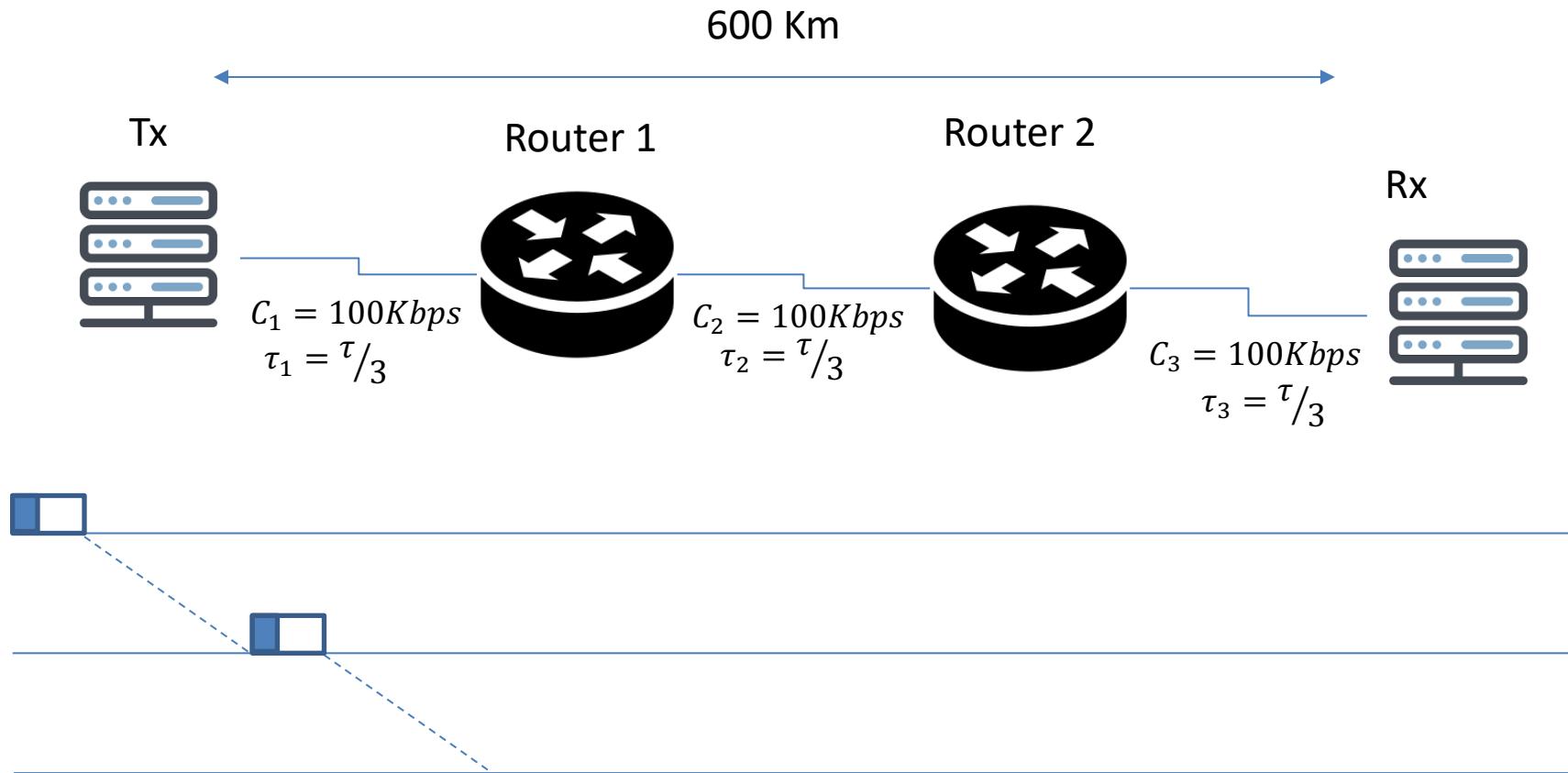
→ Sappiamo che il tempo di accodamento  $L = T$ , quindi facendo la somma otteniamo che:

$$T_{tot} = 3 \cdot T + 2 \cdot T + \tau = 5 \cdot T + \tau = 5 \cdot 30 \text{ ms} + 3 \text{ ms} = 153 \text{ ms}$$



## Esercizio 7 – Ritardi di Trasferimento – Soluzione 3/5

Scenario: Cut & Through → il pacchetto viene ritrasmesso alla completa ricezione dell'header

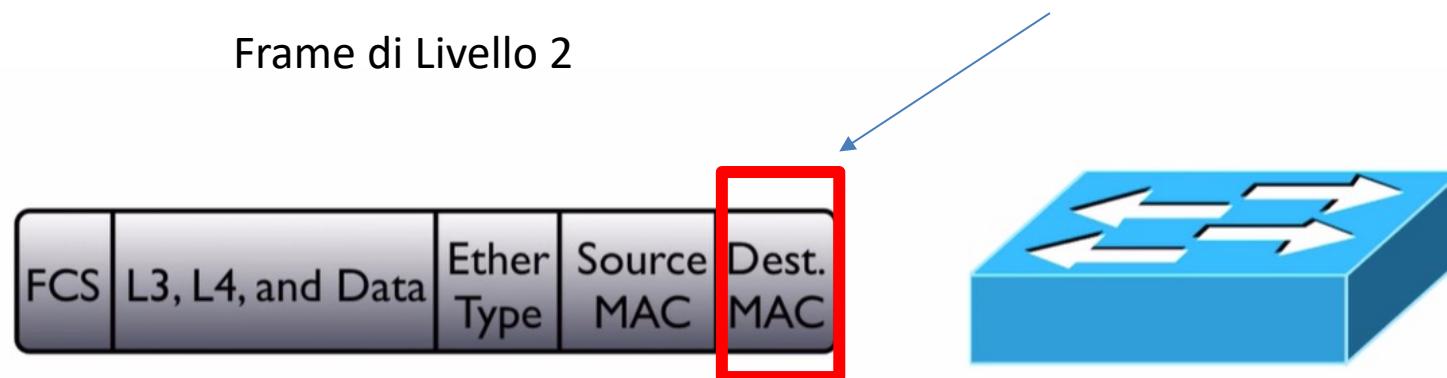


## Esercizio 7 – Ritardi di Trasferimento – Soluzione 4/5

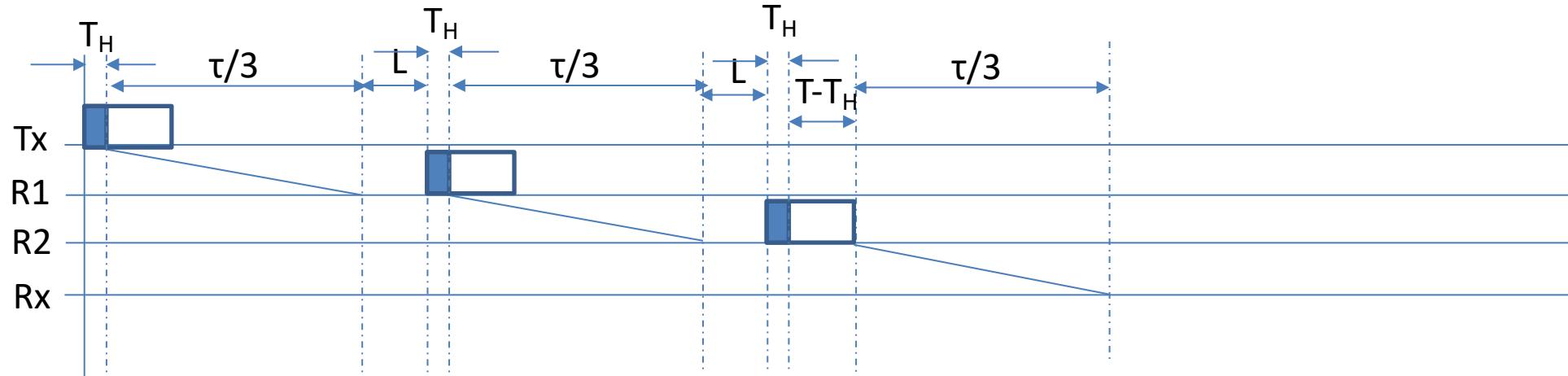
**Scenario: Cut & Through** → il pacchetto viene ritrasmesso alla completa ricezione dell'header

**Osservazioni preliminari:**

Nel caso *cut-through* il tempo di trasmissione dell'*header* si paga tre volte (l'*header* è sempre trasmesso con modalità *store and forward*) mentre il tempo di trasmissione del resto si paga una sola volta. Il tempo totale è dunque



## Esercizio 7 – Ritardi di Trasferimento – Soluzione 5/5



$$T_{tot} = T_H + (T - T_H) + T_H + \tau$$

Annotations below the equation identify the components of the total transmission time:

- Tempo di trasmissione dell'Header (Header transmission time) at the first  $T_H$
- Tempo di trasmissione del resto del pacchetto (Remaining packet transmission time) at the first  $L$
- Tempo di accodamento dell'Header (Header queuing time) at the second  $T_H$
- Tempo di ritrasmissione del resto del pacchetto (Remaining packet retransmission time) at the second  $L$
- Tempo di ritrasmissione dell'Header (Header retransmission time) at the third  $T_H$
- Tempo di accodamento dell'Header (Header queuing time) at the fourth  $T_H$
- Tempo di ritrasmissione del resto del pacchetto (Remaining packet retransmission time) at the fourth  $L$
- Tempo di ritrasmissione dell'Header (Header retransmission time) at the fifth  $T_H$

Labels below the equation indicate the nodes:

- Server 1 (covering the first three terms)
- router 1 (covering the next three terms)
- router 2 (covering the last three terms)



## Esercizio 7 – Ritardi di Trasferimento – Soluzione 5/5

$$\cancel{T_H} + \boxed{(T - T_H)} + \cancel{T_H} + \boxed{(T - T_H)} + \circled{T_H} + \cancel{T_H} + \boxed{(T - T_H)} + \circled{T_H} + \tau$$

$$T = \frac{L}{R} = \frac{3000 \text{ bit}}{100 \cdot 10^3 \text{ bps}} = 30 \cdot 10^{-3} \text{ s} = 30 \text{ ms}$$

$$\tau = \frac{5 \mu\text{s}}{\text{Km}} \cdot 600 \text{ Km} = 3000 \mu\text{s} = 3.0 \text{ ms}$$

$$T_{tot} = \boxed{3 \cdot T} + 2 \cdot T_H + \tau = 90 \text{ ms} + 4 \text{ ms} + 3 \text{ ms} = 97 \text{ ms}$$

$$T_H = \frac{200 \text{ bit}}{100 \cdot 10^3 \text{ bit/sec}} = 2 \cdot 10^{-3} \text{ sec} = 2 \text{ ms}$$

