



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Programmazione di Reti

Laboratorio #3

Andrea Piroddi

Dipartimento di Informatica, Scienza e Ingegneria

SOCKET TCP



Programmazione di un Socket - Web Server



Socket

Un **socket** è un ***oggetto software*** che **permette l'invio e la ricezione di dati, tra host remoti** (tramite una rete) **o tra processi locali**.

Più precisamente, il concetto di **socket** si basa sul modello Input/Output su file di Unix, quindi sulle operazioni di ***open, read, write e close***; l'utilizzo, infatti, avviene secondo le stesse modalità, aggiungendo i parametri utili alla comunicazione, quali **indirizzi ip, numeri di porta e protocolli**.

Socket locali e remoti in comunicazione formano una **coppia** (pair), composta da **indirizzo e porta** di client e server; tra di loro c'è una connessione logica. Solitamente i sistemi operativi forniscono delle API per permettere alle applicazioni di controllare e utilizzare i socket di rete.



Socket

1. Creazione dei socket

Client e server creano i loro rispettivi **socket**, e il **server** lo pone in **ascolto** su una **porta**. Dato che il server può creare più connessioni con client diversi (ma anche con lo stesso), ha bisogno di una **coda** per gestire le varie richieste.

2. Richiesta di connessione

Il **client** effettua una **richiesta di connessione** verso il server.

Da notare che possiamo avere due numeri di porta diversi, perchè una potrebbe essere dedicata solo al traffico in uscita, l'altra solo in entrata; questo dipende dalla configurazione dell'host.

Il **server** riceve la richiesta e, nel caso in cui sia accettata, viene creata una **nuova connessione**.

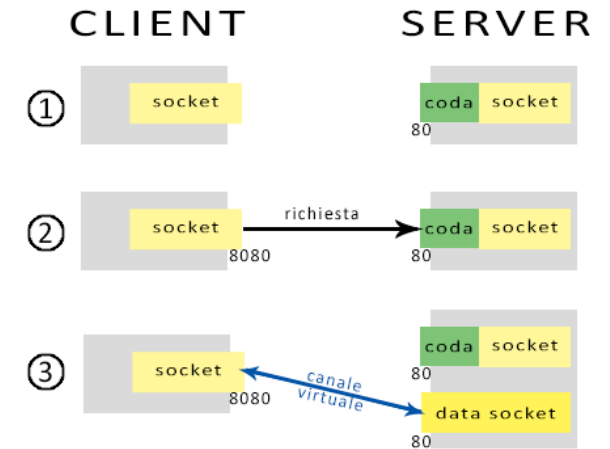
3. Comunicazione

Ora client e server comunicano attraverso un **canale virtuale**, tra il socket del primo, ed uno nuovo del server, creato appositamente per il flusso dei dati di questa connessione: **data socket**.

Coerentemente a quanto avete visto nella teoria, il server crea il data socket perchè il primo serve esclusivamente alla gestione delle richieste. È possibile, quindi, che ci siano molti client a comunicare con il server, ciascuno verso il **data socket** creato dal server per loro.

4. Chiusura della connessione

Essendo il TCP un protocollo orientato alla connessione, quando non si ha più la necessità di comunicare, il client lo comunica al server, che ne **deistanzia il data socket**. La connessione viene così chiusa.



Famiglie di socket

I tipi di **protocolli** utilizzati dal **socket**, ne definiscono la **famiglia** (o dominio).

Possiamo distinguere, ad esempio, due importanti famiglie:

- ***AF_INET***: comunicazione tra host remoti, tramite Internet;
- ***AF_UNIX***: comunicazione tra processi locali, su macchine Unix. Questa famiglia è anche chiamata *Unix Domain Socket*.



Esercizio 1: Programmazione di un Socket - Web Server

In questo laboratorio vediamo le basi della programmazione dei socket per le connessioni TCP in Python:

- come creare un socket
- associarlo a un indirizzo e una porta specifici,
- nonché inviare e ricevere un pacchetto HTTP.

Vedremo anche alcune nozioni di base sul formato dell'intestazione HTTP. Svilupperemo un server web che gestisce una richiesta HTTP alla volta.

- Il server Web deve accettare e analizzare la richiesta HTTP
- ottenere il file richiesto dal file system del server
- creare un messaggio di risposta HTTP costituito dal file richiesto preceduto da righe di intestazione e quindi inviare la risposta direttamente al client.
- Se il file richiesto non è presente nel server, il server deve inviare un messaggio **HTTP "404 non trovato"** al client.



Esercizio 1: Programmazione di un Socket - Web Server

Su IOL trovate nel Laboratorio il codice Python:
«**TCP_Socket_Server.py**» e un file «**index.html**».

Scaricate i due file (***index.html*** e ***TCP_Socket_Server.py***) sul vostro PC e metteteli nella stessa directory.



Esercizio 1: Programmazione di un Socket - Web Server

Aprirete con SPYDER il file Python e modificate il valore di porta all'interno del file python (esempio **serverPort=8080**)

```
1 # Corso di Programmazione di Reti - Laboratorio - Università di Bologna
2 # Socket Programming Assignment - WebServer - F. Callegati - G.Pau - A. Piroddi
3
4 import sys
5 from socket import *
6 serverPort=8080
7
8 serverSocket = socket(AF_INET, SOCK_STREAM)
9 server_address=('localhost',serverPort)
10 serverSocket.bind(server_address)
11
12 #listen(1) Definisce la lunghezza della coda di backlog, ovvero il numero
13 #di connessioni in entrata che sono state completate dallo stack TCP / IP
14 #ma non ancora accettate dall'applicazione.
15 serverSocket.listen(1)
16 print ('the web server is up on port:',serverPort)
17
18 while True:
19     print ('Ready to serve...')
20     connectionSocket, addr = serverSocket.accept()
21     print(connectionSocket,addr)
22
23     try:
24
25         message = connectionSocket.recv(1024)
26         if len(message.split())>0:
27             print (message,'::',message.split()[0],'::',message.split()[1])
28             filename = message.split()[1]
29             print (filename,'/',filename[1:])
30             f = open(filename[1:], 'r+')
31             outputdata = f.read()
32             print (outputdata)
33
34             #Invia la riga di intestazione HTTP nel socket con il messaggio OK
35
36             connectionSocket.send("HTTP/1.1 200 OK\r\n\r\n".encode())
37             connectionSocket.send(outputdata.encode())
38             connectionSocket.send("\r\n".encode())
39             connectionSocket.close()
40
41         except IOError:
42             #Invia messaggio di risposta per file non trovato
43             connectionSocket.send(bytes("HTTP/1.1 404 Not Found\r\n\r\n","UTF-8"))
44             connectionSocket.send(bytes("<html><head></head><body><h1>404 Not Found</h1></body></html>\r\n","UTF-8"))
45             connectionSocket.close()
```

ed eseguite il codice

```
In [2]: runfile('/Users/apirodd/OneDrive - Alma Mater Studiorum Università di
Bologna/programmazione di reti/Lezioni/Laboratorio 4/codice python Laboratorio 4/
Socket Programina Assignment 1.py', wdir='/Users/apirodd/OneDrive - Alma Mater
Studiorum Università di Bologna/programmazione di reti/Lezioni/Laboratorio 4/
codice python Laboratorio 4')
the web server is up on port: 8080
Ready to serve...
```



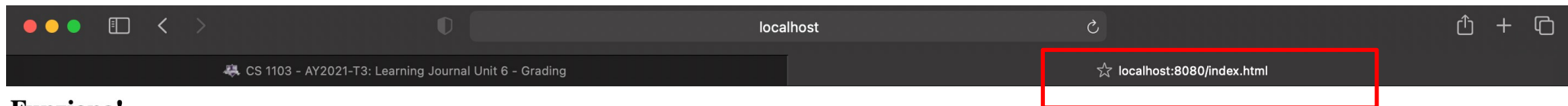
Esercizio 1: Programmazione di un Socket - Web Server

Determinare l'indirizzo IP dell'host che esegue il server (ad es. se è il vostro PC potete considerare come IP «localhost» o 127.0.0.1 indirizzo ip della loopback).

Aprire un browser sulla vostra macchina e digitare nella URL

http://localhost:8080/index.html

Dovrebbe aprirsi la pagina sotto



Funziona!

Corso di PROGRAMMAZIONE di RETI.

Laboratorio 4.



Esercizio 1: Programmazione di un Socket - Web Server

oppure determinate l'IP address associato alla vostra scheda di rete, e da un altro host, aprite un browser e inserite l'URL corrispondente ossia:

http:// «ipaddress-del-vostro-pc»:8080/index.html

«index.html» è il nome del file che avete inserito nella directory del server.

Notate anche l'uso del numero di porta dopo i due punti.

È necessario sostituire questo numero di porta con qualsiasi porta utilizzata nel codice del server. Nell'esempio sopra, abbiamo usato il numero di porta 8080. Il browser dovrebbe quindi visualizzare i contenuti di index.html. Se si omette ": 8080", il browser assumerà la porta 80 e si otterrà la pagina Web dal server solo se il server è in ascolto sulla porta 80.

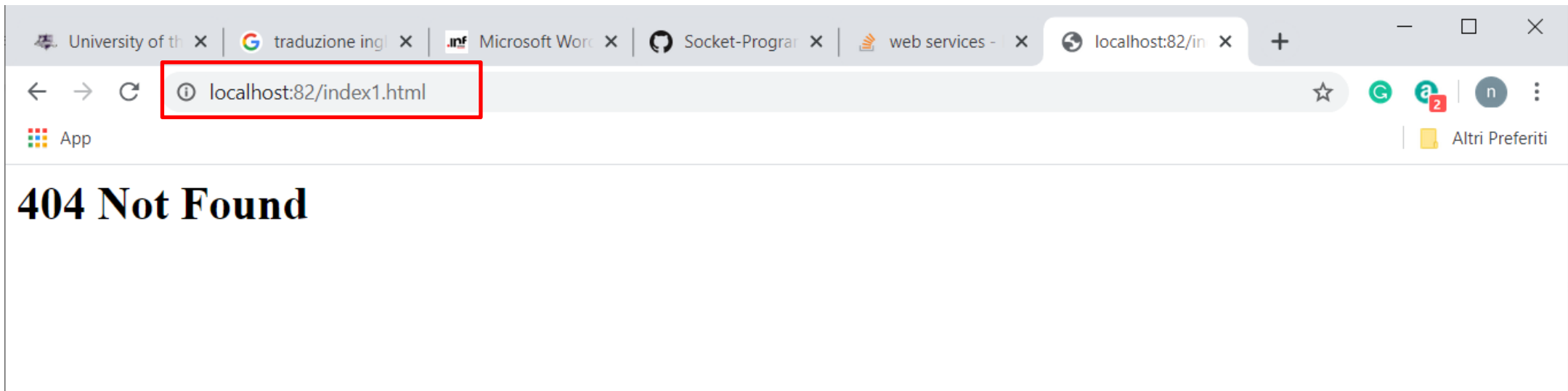


Esercizio 1: Programmazione di un Socket - Web Server

Quindi provate a ottenere un file che non è presente sul server.

Ossia provate a scrivere nella URL al posto di *index.html* il nome *index1.html*.

Dovreste ricevere un messaggio "404 Not Found".



Esercizio 1: Programmazione di un Socket - Web Server

Nel file **TCP_Socket_Server.py** ogni step è opportunamente commentato in modo da spiegare ogni singolo passaggio logico.

Analizzate il file e poi provate a fare il seguente esercizio:

Invece di utilizzare un browser, scrivete il vostro client HTTP per testare il server. Il client si connetterà al server utilizzando una connessione TCP, invierà una richiesta HTTP al server e visualizzerà la risposta del server come output. Si può presumere che la richiesta HTTP inviata sia un metodo GET.

Il client deve accettare gli argomenti della riga di comando specificando l'indirizzo IP o il nome host del server, la porta su cui è in ascolto il server e il percorso in cui l'oggetto richiesto è archiviato sul server. Di seguito è riportato un formato del comando di input per eseguire il client:

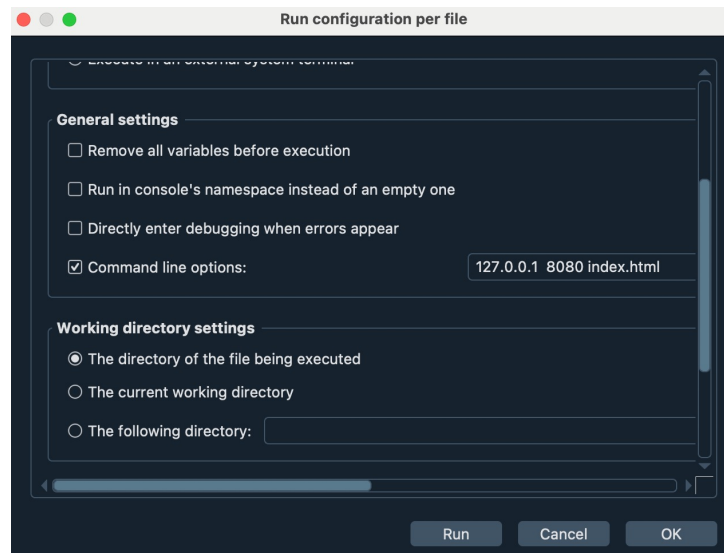
```
>>>client.py server_host server_port filename nome_file
```



Cosa dovreste vedere

Lato server

Lato client



```
Desktop/TCP Socket')
the web server is up on port: 8080
Ready to serve...
<socket.socket fd=124, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0,
laddr=('127.0.0.1', 8080), raddr=('127.0.0.1', 51922)> ('127.0.0.1', 51922)
b'GET /index.html HTTP/1.1 \r\n\r\n' :: b'GET' : b'/index.html'
b'/index.html' || b'index.html'
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <meta http-equiv="Content-Style-Type" content="text/css">
  <title></title>
  <meta name="Generator" content="Cocoa HTML Writer">
  <meta name="CocoaVersion" content="2487.4">
  <style type="text/css">
    p.p2 {margin: 0.0px 0.0px 12.0px 0.0px; font: 12.0px Times; -webkit-text-stroke: #000000}
    span.s1 {font-kerning: none}
  </style>
</head>
```

```
In [2]: runfile('/Users/apirodd/Desktop/TCP Socket/TCP_Socket_Client.py', args='127.0.0.1 8080
index.html', wdir='/Users/apirodd/Desktop/TCP Socket')
b'GET /index.html HTTP/1.1 \r\n\r\n'
```

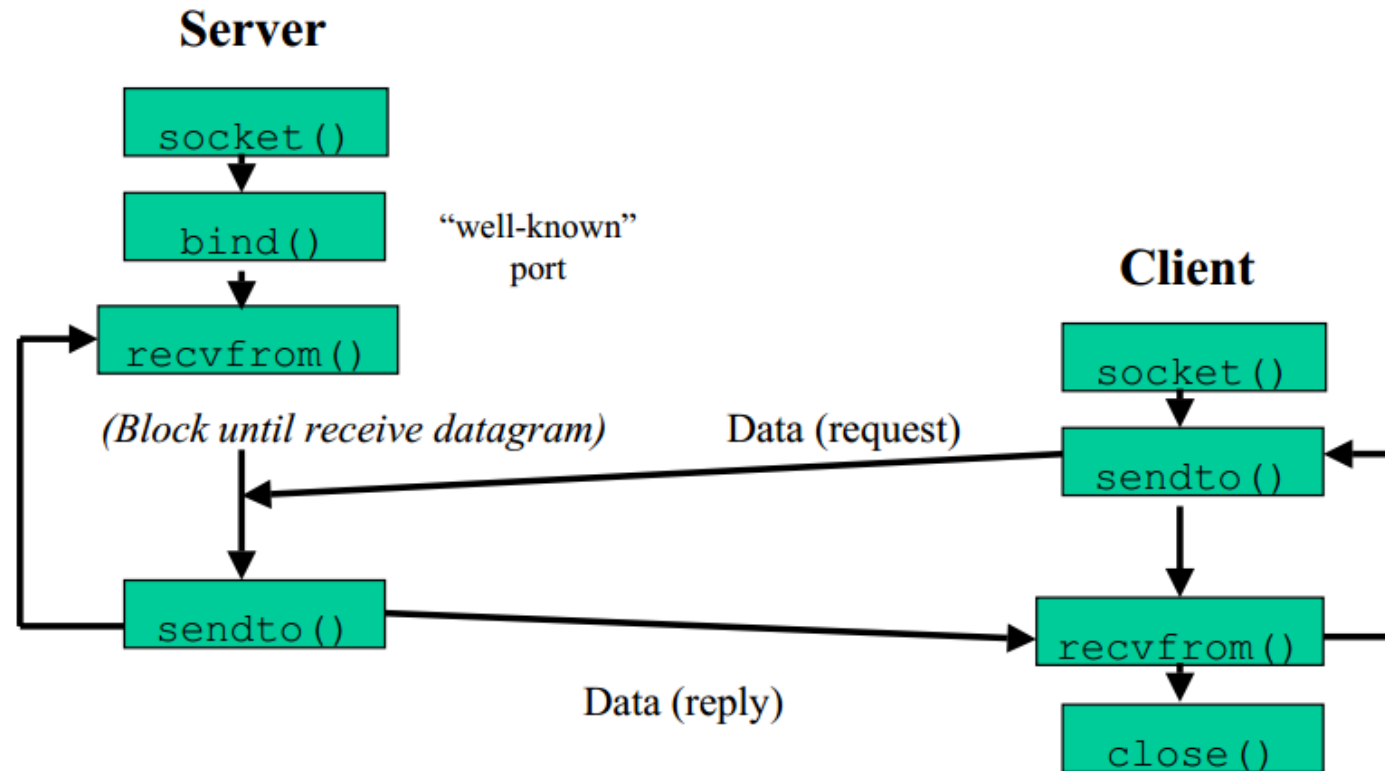


UDP SERVER SOCKET



UDP - SERVER SOCKET e CLIENT SOCKET

UDP Client-Server



UDP SERVER SOCKET

```
UDP_Socket_Server.py - F:\UDP_Socket_Server.py (3.8.0)
File Edit Format Run Options Window Help
'''
                                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))

Ln: 13 Col: 32
```



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`

Server

socket()

bind()

recvfrom()

(Block until receive)

sendto()



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)
OSError: [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:~$
```

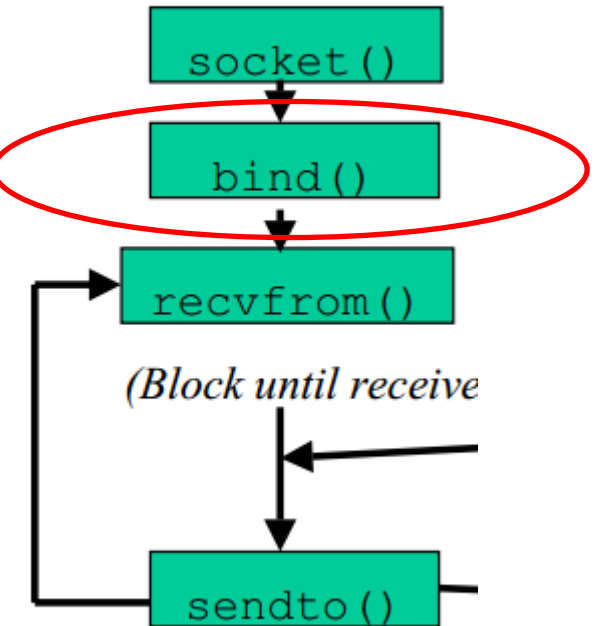


UDP SERVER SOCKET

```
sock.bind(server_address)
```

associamo la tupla `server_address` al socket `sock` creato in precedenza

Server



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`

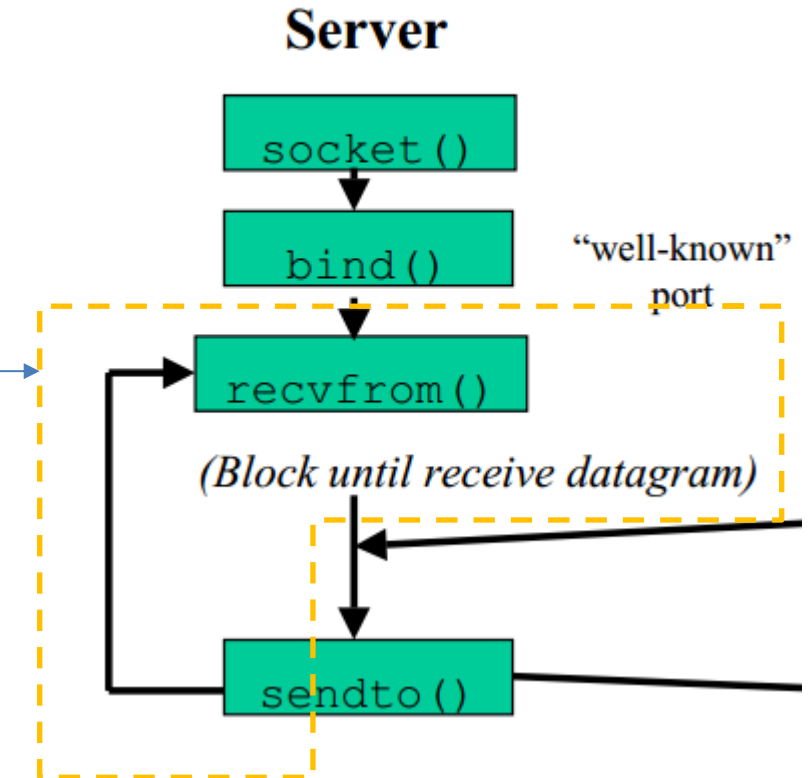
(*)

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.



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)
```

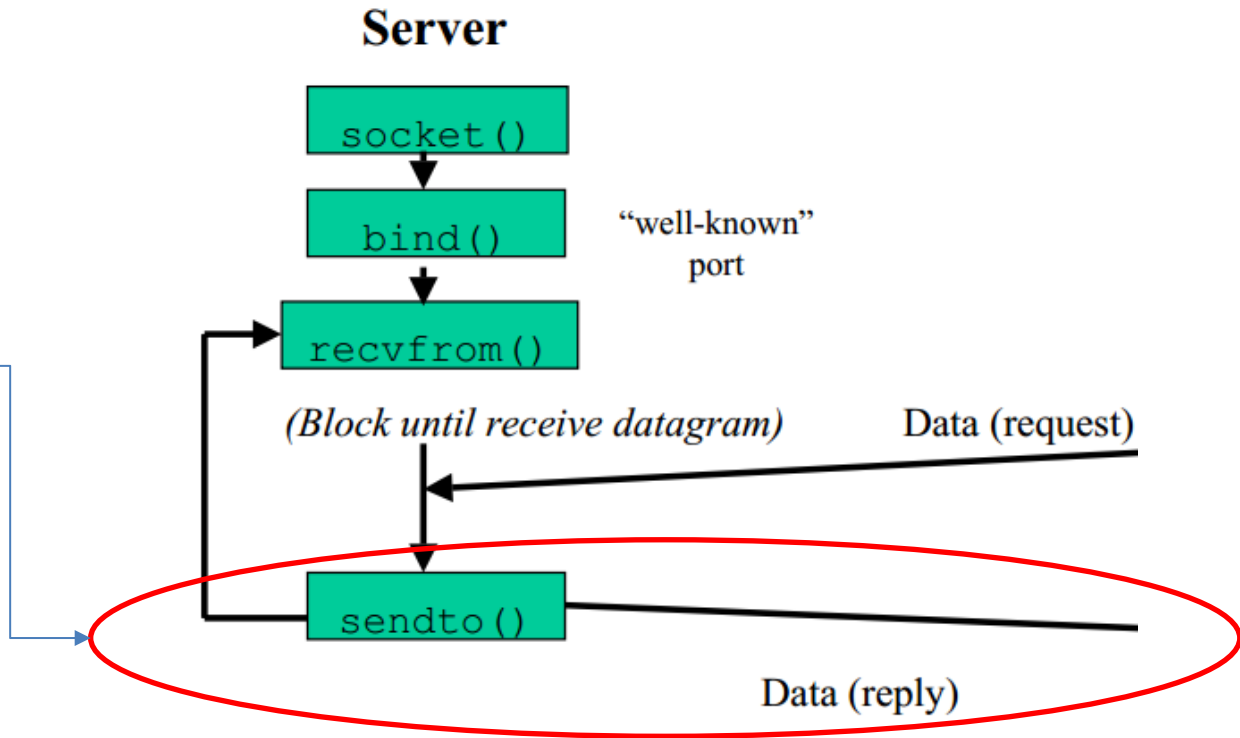
Controlla che i dati che sono arrivati sul socket non siano vuoti; in caso la variabile `data` non sia vuota, assegna alla variabile `data1` la stringa `'Programmazione di Reti'` e poi attende 2 secondi prima di inviarla verso il client.



UDP SERVER SOCKET

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

Inviemo 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_Server.py - F:\UDP_Server.py (3.8.0)
File Edit Format Run Options Window Help

'''
        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' %
          (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))
```

1

4

6

Recv: riceve un messaggio su di un certo socket

Recvfrom: riceve un messaggio su di un certo socket (non orientato alla connessione)

Send: invia un messaggio attraverso un socket

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

Lnc 28 Col: 0

```
TCP_Server.py - F:\TCP_Server.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

import sys
from socket import *

serverSocket = socket(AF_INET, SOCK_STREAM)
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)

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

    #riceve il messaggio
    message = connectionSocket.recv(1024)
    message = message.strip()
    if len(message) > 0:
        print (message, '::', message.split()[0], '::', message.split()[1])
        filename = message.split()[1]
        print (filename, '|', filename[1:])
        f = open(filename[1:], 'r+')
        outputdata = f.read()
        print (outputdata)

        #Invia la riga di intestazione HTTP nel socket con il messaggio OK
        connectionSocket.send('HTTP/1.1 200 OK\r\n\r\n'.encode())
        connectionSocket.send(outputdata.encode())
        connectionSocket.send("\r\n".encode())
        connectionSocket.close()

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

Lnc 15 Col: 0
```

1

2

3

4

5

6

7



UDP

CLIENT SOCKET



UDP CLIENT SOCKET

```
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
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()
```

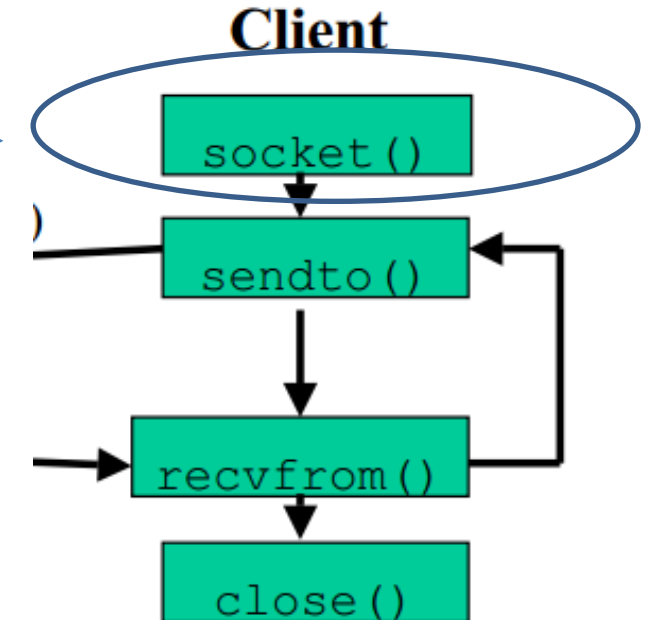
Ln: 1 Col: 0



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 ?'
```

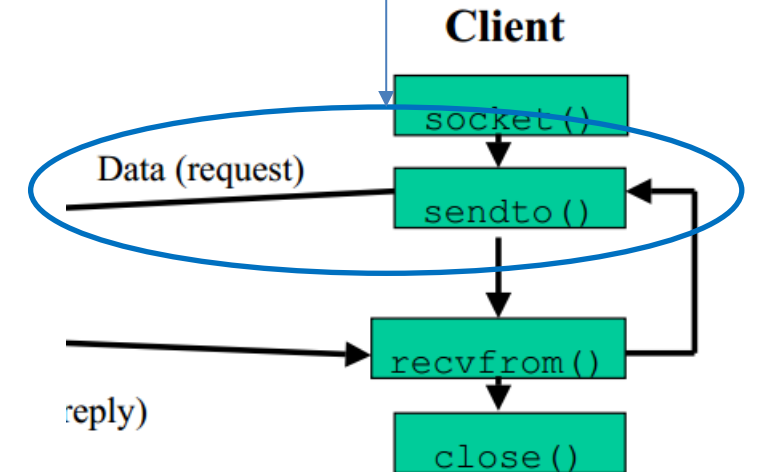
Prepariamo la variabile *message* assegnandole la stringa che vogliamo inviare al server



UDP CLIENT SOCKET

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

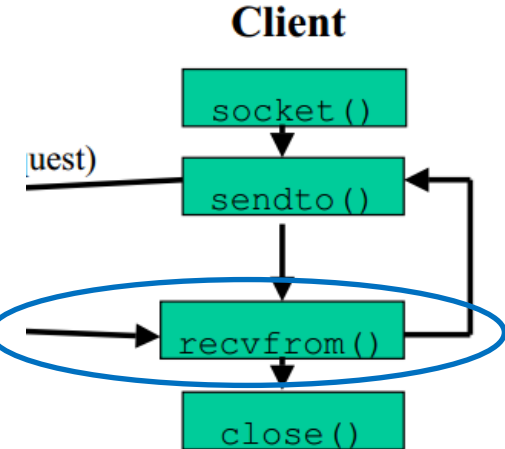
Inviando, tramite il modulo socket (*sock.sendto()*), il messaggio all'indirizzo del server (ipaddress e udp port) e associamo il contenuto di alla variabile *sent*



UDP CLIENT SOCKET

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

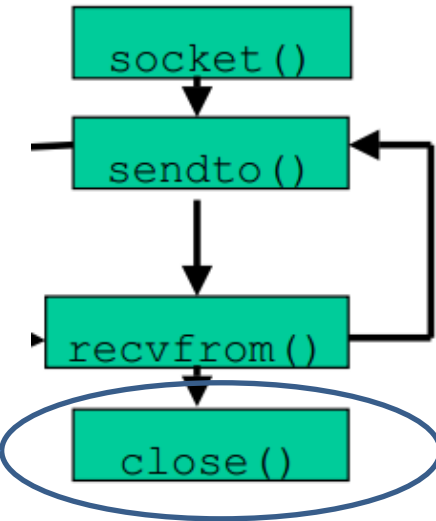
Appena riceviamo il contenuto del messaggio proveniente dal server tramite il modulo socket *sock.recvfrom()* ne assegniamo il contenuto alle due variabili *data* e *server*, corrispondenti rispettivamente al PAYLOAD e all'indirizzo del server (ip address e porta) da cui esso proviene.



UDP CLIENT SOCKET

```
sock.close()
```

Client



Al termine della ricezione del datagram da parte del server, chiudiamo il 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
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 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:
    print ('closing socket')
    sock.close()
```

Ln: 36 Col: 0

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

'''

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])
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())
response = clientsocket.recv(1024)

print (response)

clientsocket.close()
```

Ln: 21 Col: 0



WIRESHARK



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)

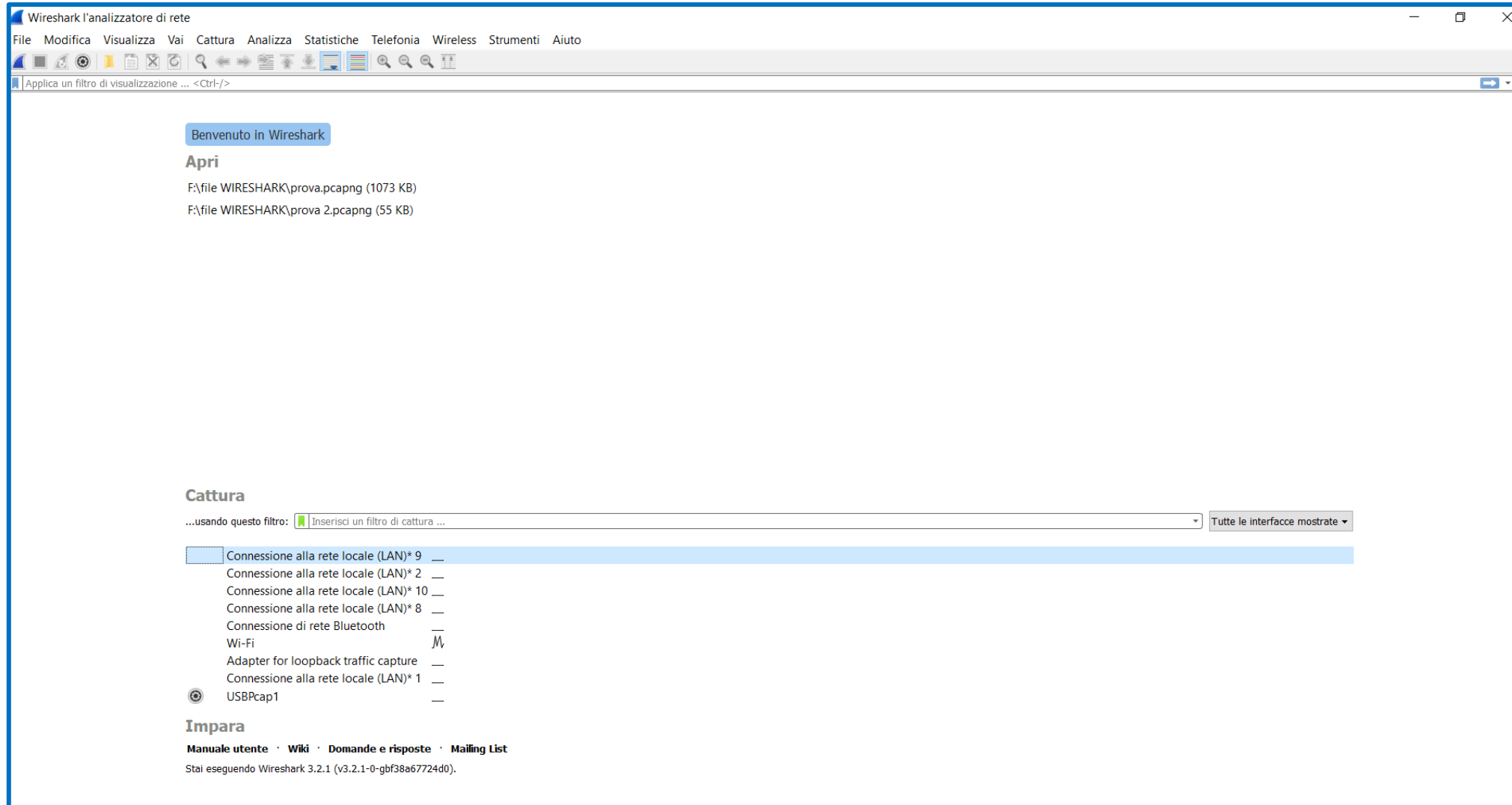
 Windows Installer (64-bit)
Windows Installer (32-bit)
Windows PortableApps® (32-bit)
macOS Intel 64-bit .dmg
Source Code

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



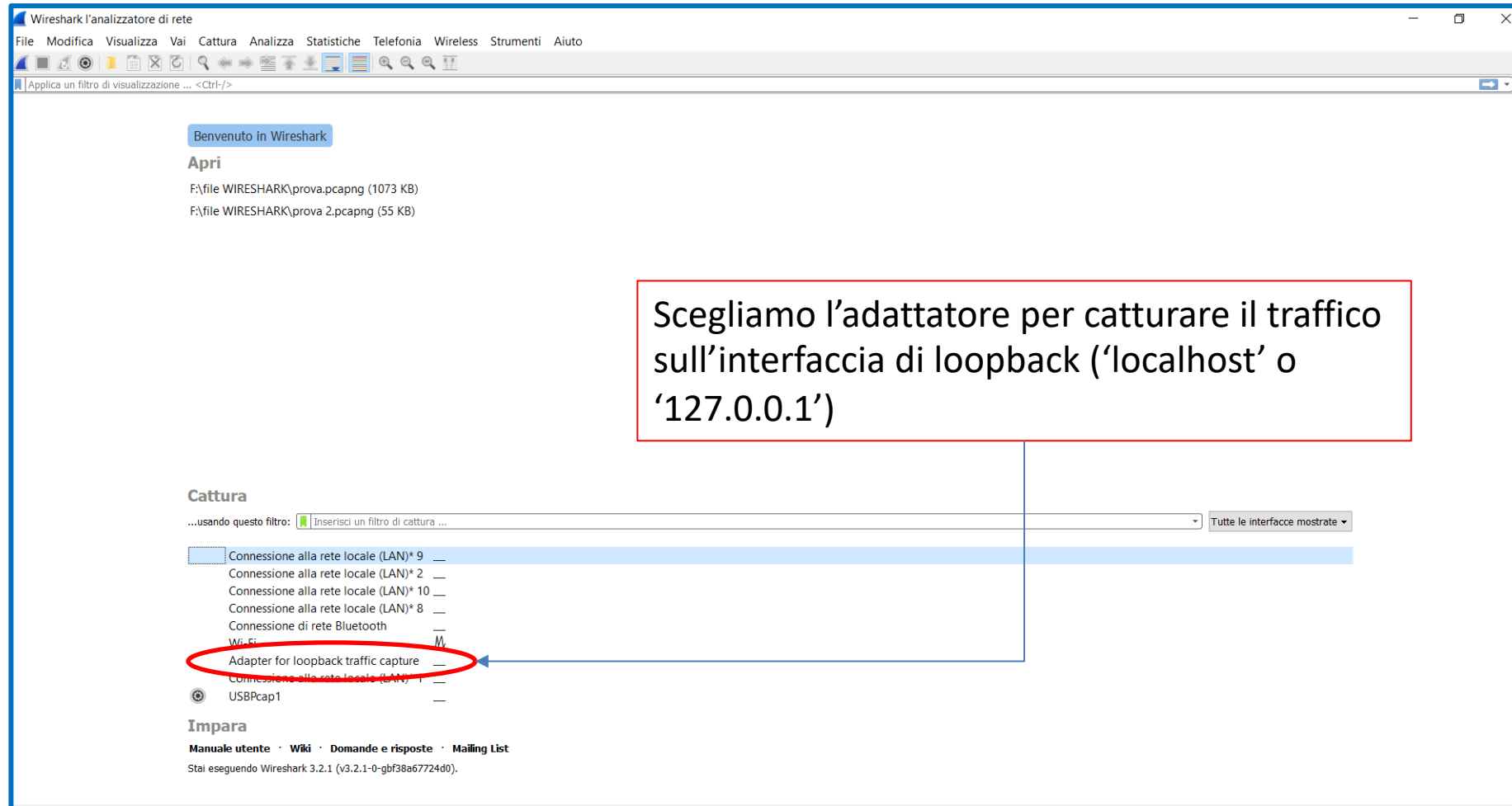
ANALIZZATORE di PROTOCOLLO - WIRESHARK

L'interfaccia grafica si presenta così:

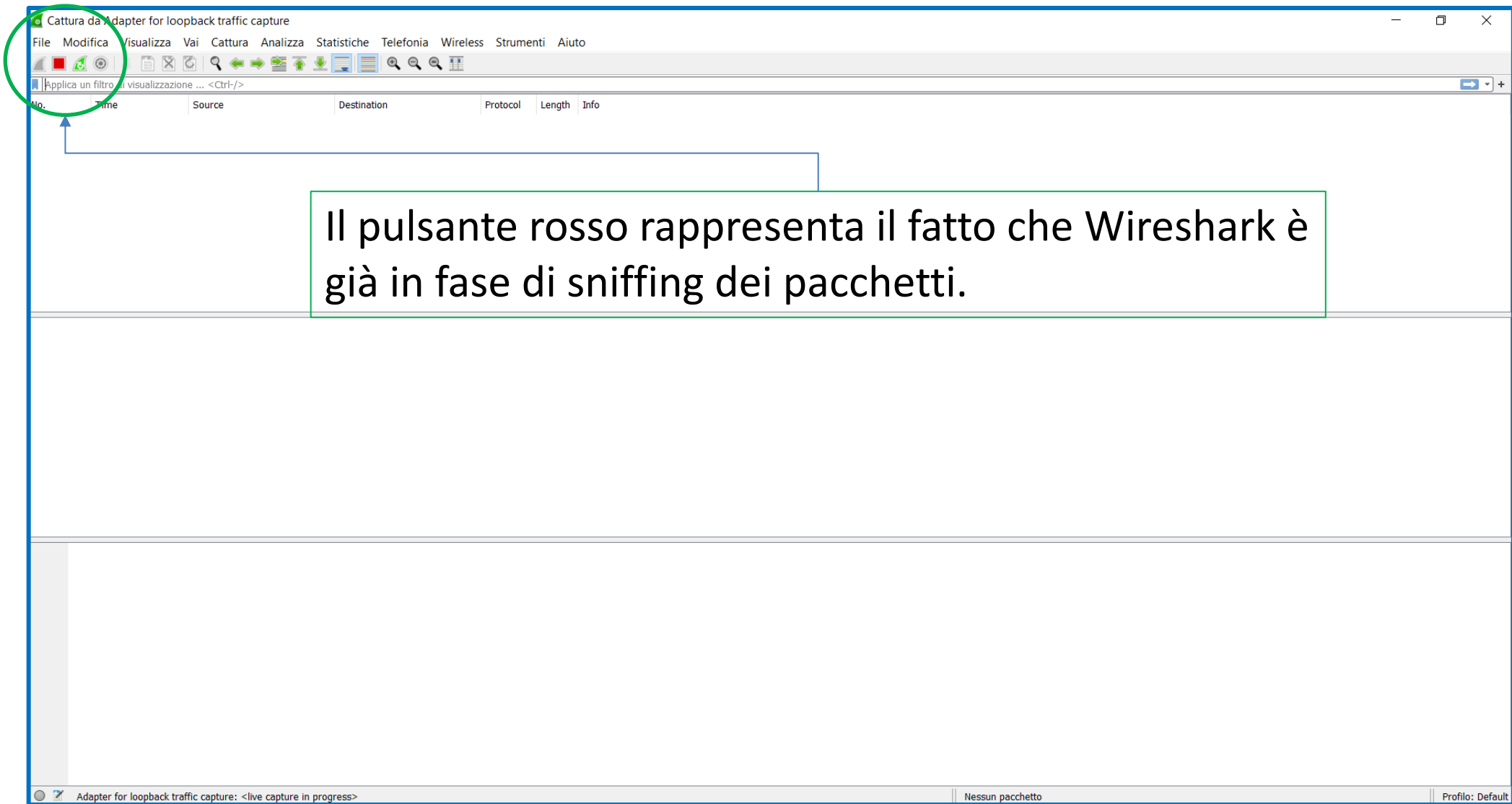


ANALIZZATORE di PROTOCOLLO - WIRESHARK

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



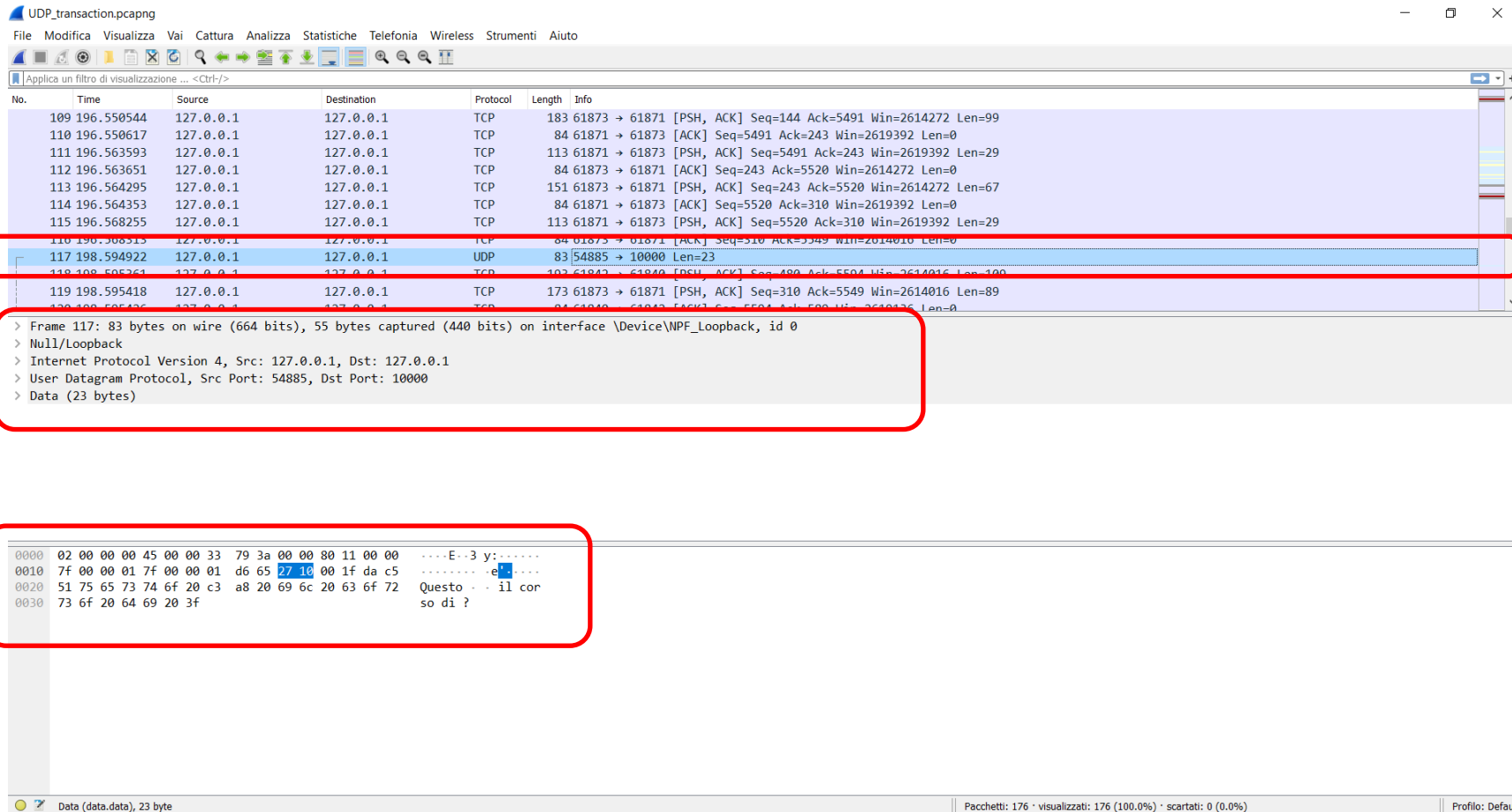
ANALIZZATORE di PROTOCOLLO - WIRESHARK



ANALIZZATORE di PROTOCOLLO - WIRESHARK

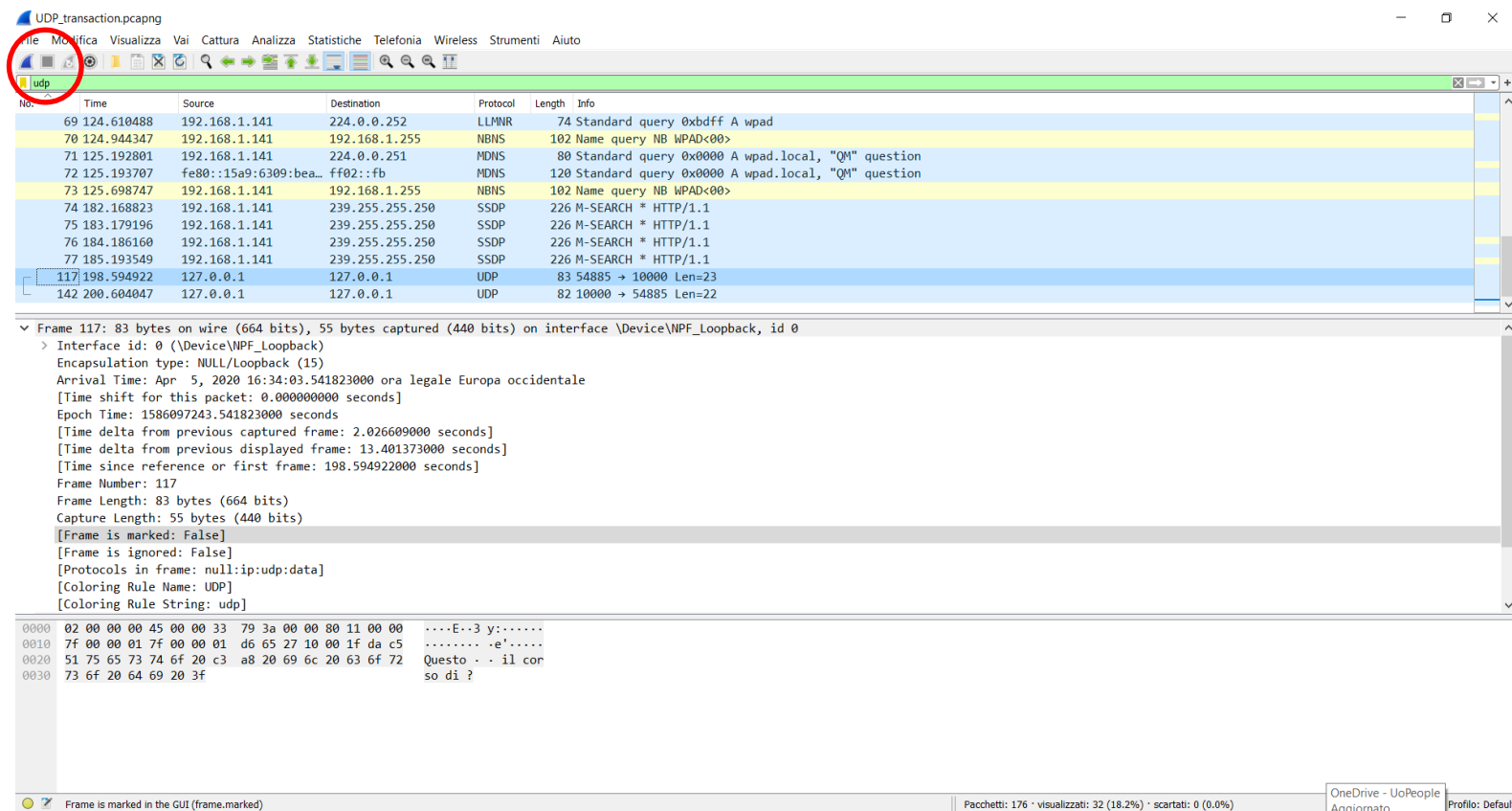
Wireshark è attivo e sta sniffando il traffico che proviene ed è destinato alla interfaccia di loopback.

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:



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.

The screenshot shows the Wireshark interface with the following details:

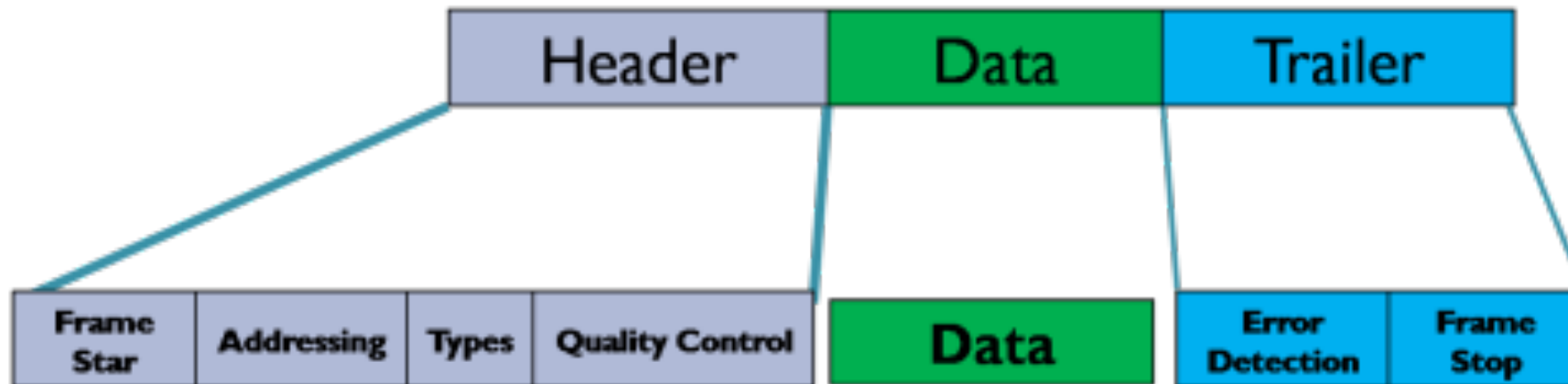
- Filter Bar:** The filter 'udp' is entered and highlighted with a red circle.
- Packet List:** A table of captured packets. Packet 117 is selected and highlighted with a red box.
- Packet Details:** The details pane for packet 117 shows the following information:
 - Frame 117: 83 bytes on wire (664 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]
- Packet Bytes:** The bottom pane shows the raw packet data in hexadecimal and ASCII.

Analizziamo la richiesta del client



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]

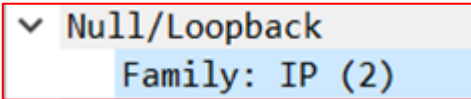
Epoch Time (noto anche come tempo UNIX) è il numero di secondi dal 1 ° gennaio 1970. Questo è ciò che è effettivamente memorizzato nel file .pcap o .pcapng. Gli altri formati di tempo in Wireshark sono conversioni del Epoch Time a scopo di visualizzazione.

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

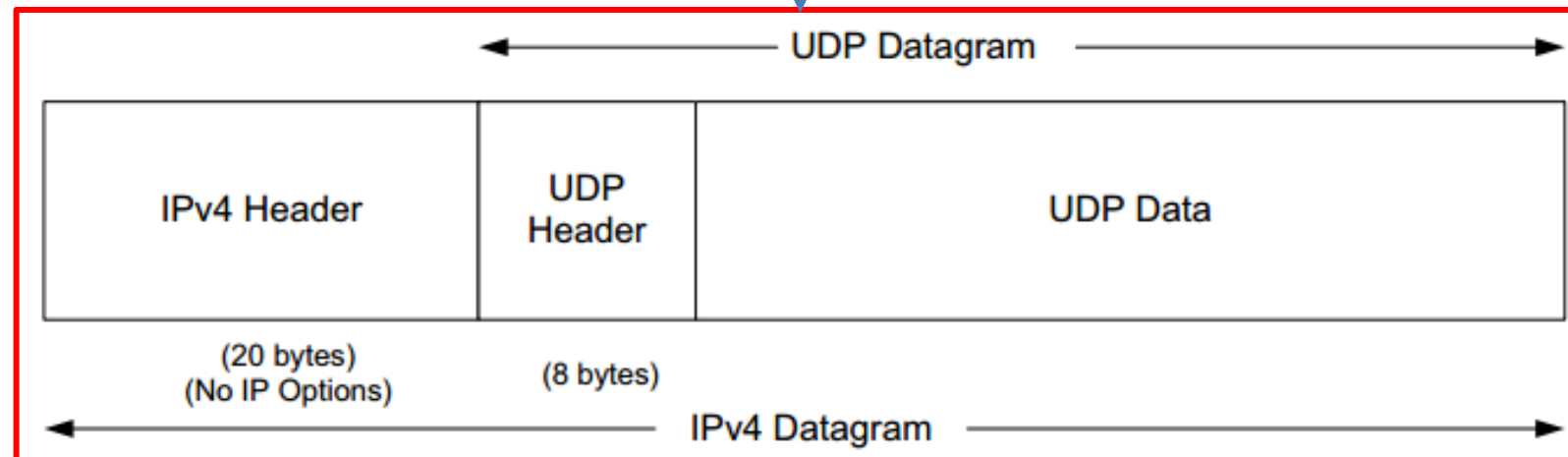
Length: 31

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 è 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 cor□so 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 0xbdf 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	108.504022	127.0.0.1	127.0.0.1	UDP	82	10000 → 54885 Len=22
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 00E--2 yS.....
0010 7f 00 00 01 7f 00 00 01 27 10 d6 65 00 1e da eef.....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

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							
v Null/Loopback							
Family: IP (2)							
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1							
v 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]							
v Data (22 bytes)							
Data: 50726f6772616d6d617a69666e652064692052657469							
[Length: 22]							
0000	02 00 00 00 45 00 00 32	79 53 00 00 80 11 00 00E..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				





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA