



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA  
CAMPUS DI CESENA

# **Programmazione di Reti**

## **Laboratorio #8**

**Andrea Piroddi**

Dipartimento di Informatica, Scienza e Ingegneria

# **Applicazione TCP per realizzazione di HTTP server Multithread**



# Applicazione socket TCP per realizzazione di HTTP server Multithread

```
import sys, signal
import http.server
import socketserver
```

Importiamo 4 moduli principali

- I due moduli `sys` e `signal`. Il [`sys`](#) per accettare informazioni inviate tramite la riga di comando e il modulo [`signal`](#) che consente di definire handlers personalizzati da eseguire quando si riceve un segnale. In particolare lo utilizzeremo per interrompere il processo `HttpServer` da tastiera.
- Importiamo il modulo [`http.server`](#). Una classe, `HTTPServer`, è una sottoclasse `socketserver.TCPServer`. Crea e ascolta sul socket HTTP, inviando le richieste a un handler.
- Importiamo infine il modulo [`socketserver`](#)

Creo la classe base **A()** con una proprietà.

```
1 class A:
2     nazione="Italia"
```

Poi creo la **classe derivata B(A)**.

La sottoclasse ha un'altra proprietà (regione).

```
1 class B(A):
2     regione="Lazio"
```

Tra parentesi indico la classe base (A) a cui appartiene la sottoclasse B. Questo vuol dire che la **classe B eredita automaticamente tutte le proprietà della classe base A**.



## Applicazione socket TCP per realizzazione di HTTP server Multithread

```
if sys.argv[1:]:  
    port = int(sys.argv[1])  
else:  
    port = 8080
```

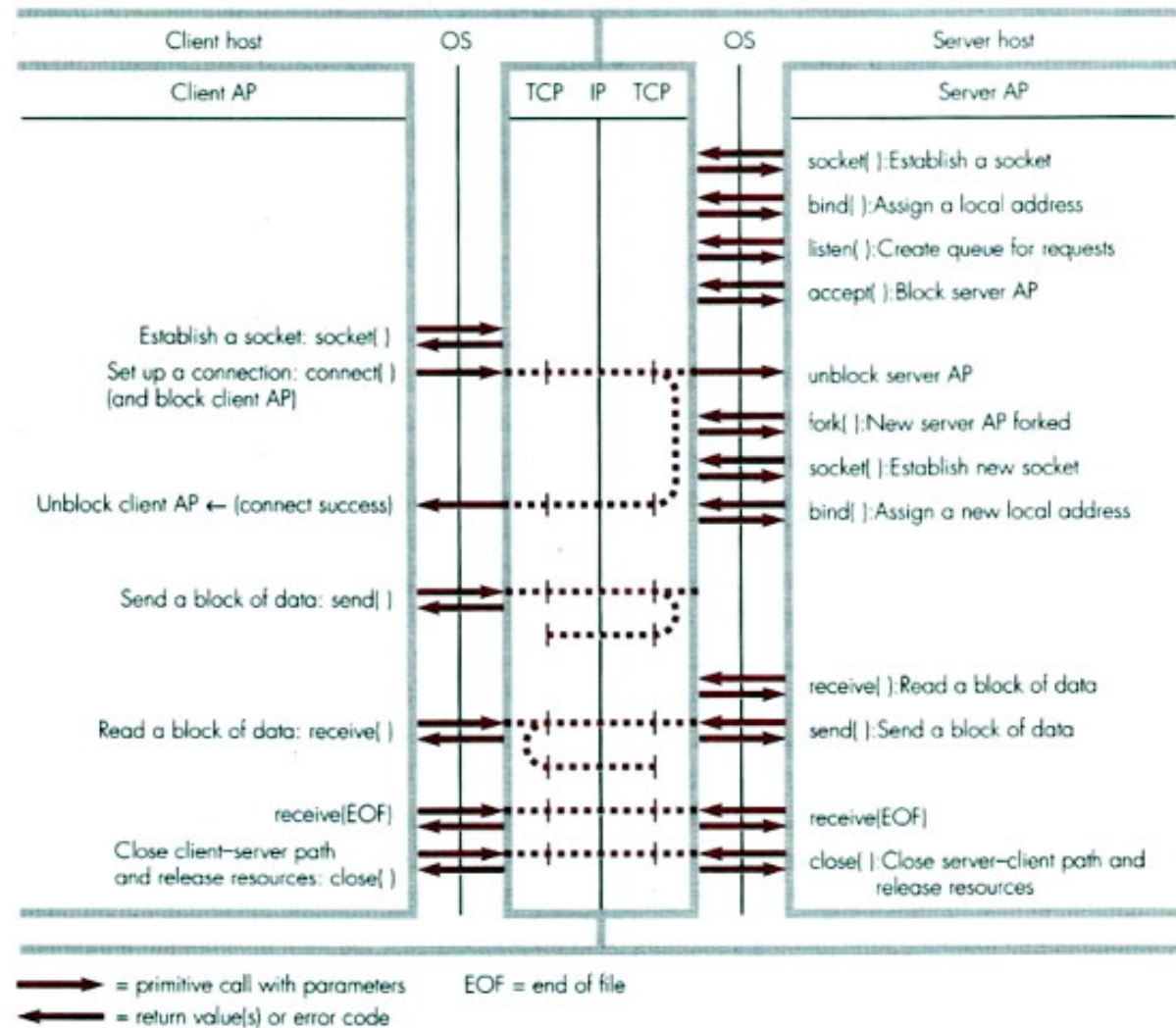
Facciamo in modo che il server http si metta in ascolto sulla porta che gli forniamo quando lo eseguiamo, in caso non venga specificata la porta in fase di esecuzione esso prenderà la 8080 come default



## Applicazione socket TCP per realizzazione di HTTP server Multithread

In una applicazione client-server i computer in comunicazione non hanno ruoli identici, ma sono definite, per così dire, due personalità, in cui uno (il client) svolge la funzione di richiedente, mentre l'altro (il server) risponde.

Nel caso in cui anche il computer server si trovi nella necessità di effettuare delle richieste, allora esisterà al suo interno un diverso processo, avente funzione di client.

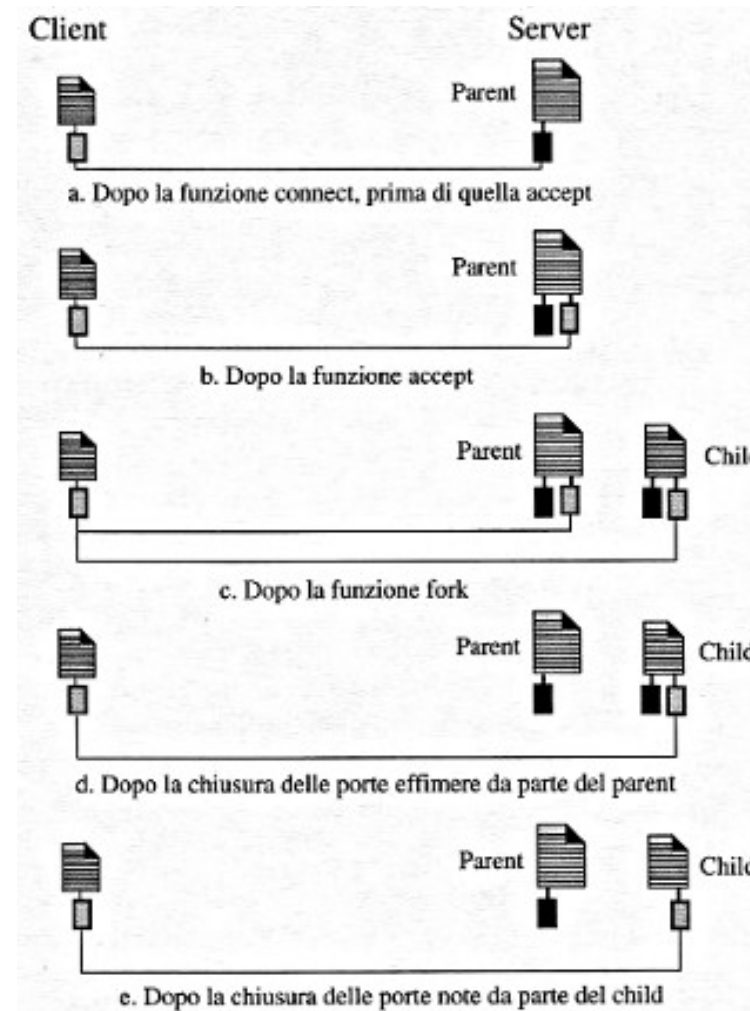


# Applicazione socket TCP per realizzazione di HTTP server Multithread

Osserviamo che quando la *accept()* del server ritorna, questo esegue una *fork()*.

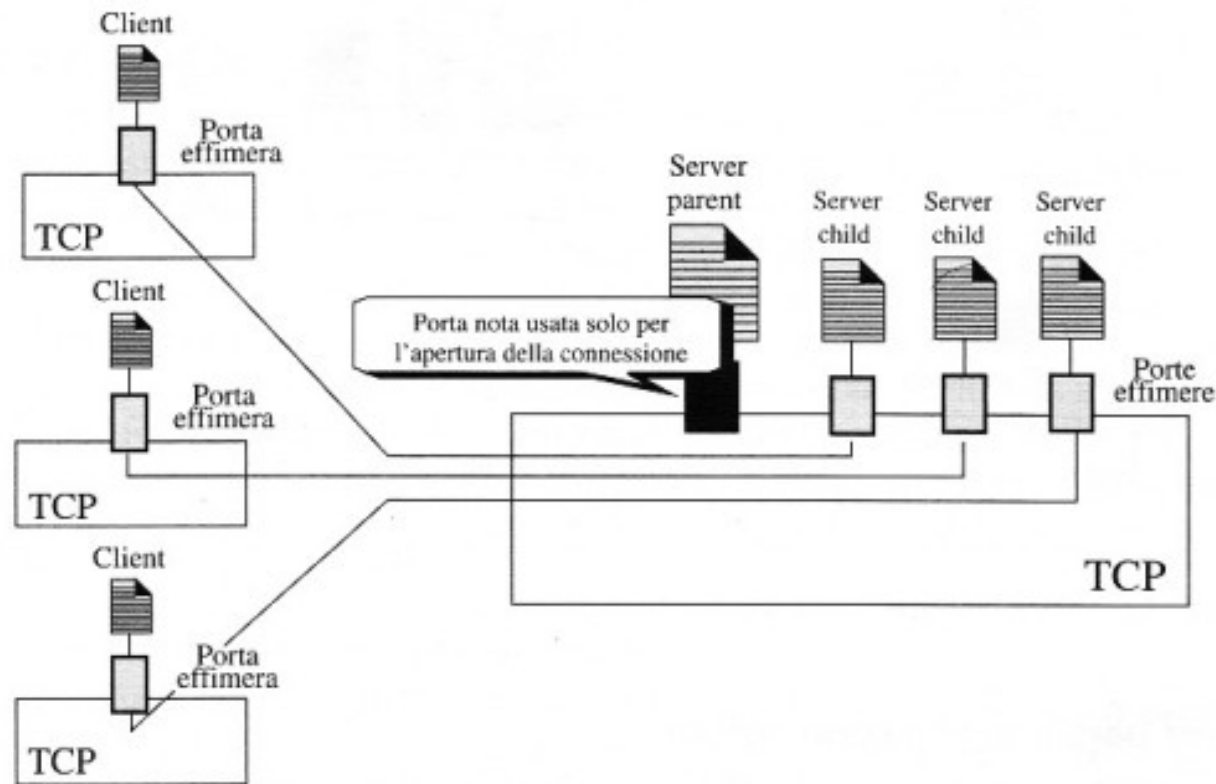
`pid_t fork();`

Ciò significa che il processo che sta eseguendo il codice del server viene **clonato**, ovvero vengono duplicate le sue aree di memoria programma e memoria dati, che vengono associate ad un nuovo PID (Process Identifier).

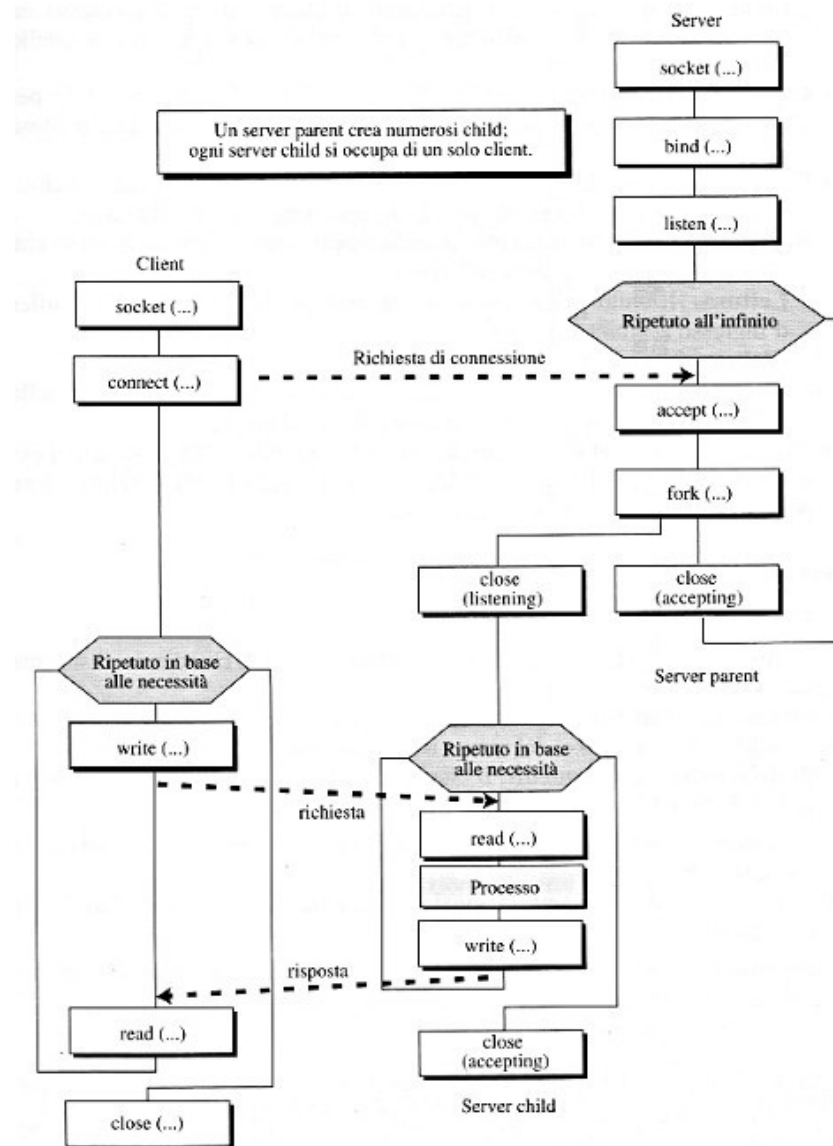


# Applicazione socket TCP per realizzazione di HTTP server Multithread

**TIPOLOGIA di SERVENTE Multithread: concorrenti, o paralleli.** Il server crea processi figli (o thread) incaricati di rispondere. I processi figli, una volta esaurito il loro compito, terminano. Si tratta del caso in cui l'interazione avviene utilizzando un trasporto con connessione ed affidabile, ovvero il TCP, e l'intervallo tra due richieste è generalmente inferiore a quello necessario a generare una risposta.



# Applicazione socket TCP per realizzazione di HTTP server Multithread





## Applicazione socket TCP per realizzazione di HTTP server Multithread

ForkingTCPServer non è però disponibile sui sistemi Windows come `os.fork()`.

E' necessario quindi usare la funzione alternativa ***ThreadingTCPServer*** per gestire più richieste.



```
server = socketserver.ThreadingTCPServer(('',port), http.server.SimpleHTTPRequestHandler )
```



### ***classe http.server.SimpleHTTPRequestHandler***

Questa classe fornisce i file dalla directory corrente e tutte quelle inferiori, mappando direttamente la struttura della directory alle richieste HTTP.

Gran parte del lavoro, come l'analisi della richiesta, viene svolto dalla classe base `BaseHTTPRequestHandler`. Questa classe implementa le funzioni `do_GET()` e `do_HEAD()`.



# Applicazione socket TCP per realizzazione di HTTP server Multithread

La classe SimpleHTTPRequestHandler definisce i seguenti metodi:

## **do\_HEAD ()**

Questo metodo serve il tipo di richiesta "HEAD": invia le intestazioni che invierebbe per la richiesta GET equivalente.

## **do\_GET ()**

La richiesta viene mappata su un file locale interpretando la richiesta come percorso relativo alla directory di lavoro corrente. Se la richiesta è invece mappata su una directory specifica, la directory viene controllata alla ricerca di un file denominato index.html o index.htm (in quest'ordine). Se trovato, i contenuti del file vengono restituiti; in caso contrario viene generato un elenco di directory chiamando il metodo list\_directory (). Questo metodo utilizza os.listdir () per scansionare la directory e restituisce una risposta di errore 404 se listdir () fallisce.



## Applicazione socket TCP per realizzazione di HTTP server Multithread

```
server.daemon_threads = True
```

Per gestire al meglio il comportamento della connessione **thread** ereditata da **ThreadingTCPServer**, è necessario dichiarare esplicitamente come si desidera che i thread si comportino in caso di arresto improvviso.

La classe **ThreadingTCPServer** definisce un attributo **daemon\_threads**, che indica se il server deve attendere la terminazione del thread.

Si dovrebbe impostare esplicitamente il flag se si desidera che i thread si comportino in modo autonomo; il valore predefinito è **False**, il che significa che Python non verrà chiuso fino alla chiusura di tutti i thread creati da **ThreadingTCPServer**.



## Applicazione socket TCP per realizzazione di HTTP server Multithread

```
server.allow_reuse_address = True
```

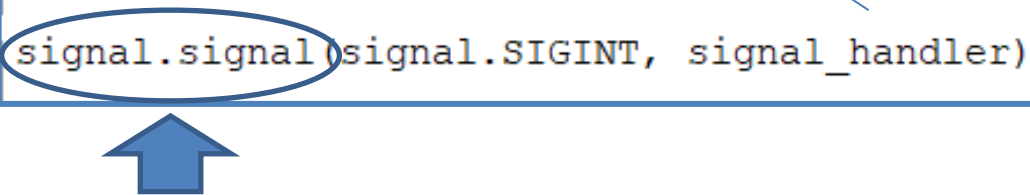
Una volta associato il server ad una porta e impostato ***server.allow\_reuse\_address = True*** stiamo indicando al Server di riutilizzare la stessa porta, senza dover attendere che il kernel rilasci la porta sottostante.

Per impostazione predefinita, infatti non è consentito associare un socket se esiste già un socket associato a quella porta, ma è possibile sovrascriverlo con ***allow\_reuse\_address***



## Applicazione socket TCP per realizzazione di HTTP server Multithread

```
def signal_handler(signal, frame):  
    print( 'Exiting http server (Ctrl+C pressed)')  
    try:  
        if( server ):  
            server.server_close()  
    finally:  
        sys.exit(0)  
  
signal.signal(signal.SIGINT, signal_handler)
```



La funzione *signal.signal ()* consente di definire handler personalizzati da eseguire quando si riceve un segnale per esempio da tastiera. Sono installati un numero limitato di handler predefiniti: *SIGPIPE, SIGINT,....*

*signal.SIGINT* interrompe l'esecuzione se da tastiera arriva la sequenza (CTRL + C) e fa chiamare l'*handler signal\_handler()*.



## Applicazione socket TCP per realizzazione di HTTP server Multithread

```
try:
    while True:

        server.serve_forever()
except KeyboardInterrupt:
    pass

server.server_close()
```

### *serve\_forever (poll\_interval = 0.5)*

Gestisce le richieste finché il programma non termina o finché non viene esplicitamente interrotto. Ogni poll\_interval secondi controlla se è stata richiesta l'interruzione tramite shutdown(), ed eventualmente ritorna. Questa funzione blocca il thread corrente indefinitamente.



## Applicazione socket TCP per realizzazione di HTTP server Multithread

Potete osservare che il server in questione è in grado anche di inviare immagini presenti nella pagina richiesta, il tutto grazie alla classe [\*http.server.SimpleHTTPRequestHandler\*](#).

Questa classe recupera i file dalla directory corrente e da quelle in essa contenute, mappando direttamente la struttura della directory alle richieste HTTP.

Uno degli attributi definiti a livello di classe di *SimpleHTTPRequestHandler* è il ***do\_GET()*** che a sua volta ha tra le caratteristiche quella di gestire il ***Content-Type*** e quindi identificare il tipo di file a cui il puntatore lo sta indirizzando e inviarlo sul socket con la codifica opportuna.



## Applicazione socket TCP per realizzazione di HTTP server Multithread

Questo server è in realtà un server leggero, adeguato per contenuti statici.  
Per contenuti Web semplici come HTML, JavaScript e fogli di stile CSS funziona bene.

Tuttavia, non gestisce molto bene la pubblicazione di file di grandi dimensioni come video e audio.

Nel caso in cui si desideri un server Web semplice che gestisca sia il contenuto statico semplice sia i file audio e video in streaming, è consigliabile utilizzare il modulo `http-server node.js`.

Questo server gestisce abbastanza bene la pubblicazione di file di grandi dimensioni (>300 MB), grazie al funzionamento asincrono di `node.js`.





# Client – Router - Server



# MAC Address

Gli indirizzi MAC sono indirizzi univoci a livello globale. Come questi siano assegnati è un argomento di discussione interessante, ma al momento è sufficiente sapere che gli indirizzi MAC sono fissi per un determinato nodo della rete, indipendentemente dalla rete a cui è connesso il nodo.

Ora immaginiamo un modello di rete in cui ci sono solo indirizzi MAC.

Supponiamo di avere un nodo (per esempio il nostro portatile) con indirizzo

MAC AA: AA: AA: AA: AA: AA

e che il vostro amico abbia un nodo con indirizzo

MAC BB: BB: BB: BB: BB: BB.

Immaginiamo di voler inviare alcuni dati al vostro amico.

## Example MAC Address

**3A-34-52-C4-69-B8**

Organizationally  
Unique Identifier

(OUI)

Network Interface  
Controller

(NIC)



# MAC Address

Ci sono milioni di nodi nel mondo, che comprendono piccole reti che si uniscono per formare Internet.

In quale di queste reti più piccole si trova l'indirizzo MAC del vostro amico?...

Emerge quindi, che un modello di rete basato su indirizzo MAC non è scalabile.

Questo perché gli indirizzi MAC sono legati indelebilmente (...o quasi) ai nodi e non abbiamo modo di determinare in quale rete può trovarsi il vostro MAC, ad esempio BB: BB: BB: BB: BB: BB: BB non ha modo di dirvi in quale parte del mondo si trova attualmente.

La consegna dei dati implica quindi che il router a cui il vostro nodo è collegato debba eseguire la scansione delle reti di tutto il mondo per determinare a quale rete è collegato questo indirizzo MAC.



# MAC Address e IP Address

Motivo per cui nasce la modalità di indirizzamento IP.

Mentre gli indirizzi MAC sono proprietà di nodi specifici, gli indirizzi IP sono proprietà delle reti (gli indirizzi IP assegnati ai nodi dipendono dalla rete a cui è collegato il nodo).

Ciò significa che, dato un indirizzo IP, ad esempio 9.100.100.8, esistono metodi per determinare a quale rete del mondo appartiene questo IP.

Ad esempio, il nodo del vostro amico potrebbe avere l'indirizzo IP 9.100.100.12 e questa potrebbe essere la rete di un Mobile Network Carrier con sede in Europa.



```
Administrator: C:\Windows\system32\cmd.exe

Wireless LAN adapter Wireless Network Connection:
Media State . . . . . : Media disconnected
Connection-specific DNS Suffix . :

Ethernet adapter Local Area Connection:
Connection-specific DNS Suffix . :
Link-local IPv6 Address . . . . . : fe80::259c:8937:2185:1cdh::1
IPv4 Address. . . . . : 192.168.0.100
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.0.1

Tunnel adapter isatap.{17D36277-52BD-43DB-92C7-389E1D9175E9}:
Media State . . . . . : Media disconnected
Connection-specific DNS Suffix . :

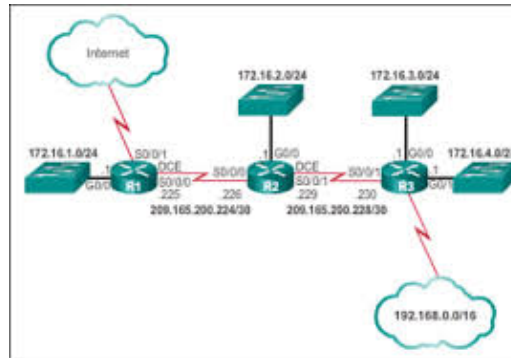
Tunnel adapter Icrdo Tunneling Pseudo-Interface:
Media State . . . . . : Media disconnected
Connection-specific DNS Suffix . :

C:\Users\DLINMCA>
```



## MAC Address e IP Address

Ciò semplifica la ricerca del nodo di destinazione rispetto alla ricerca nel modello di indirizzo MAC della rete discusso in precedenza. Il router ora indirizzerà il traffico verso la rete specificata in Europa e i router in Europa restringeranno ulteriormente la posizione geografica del nodo del vostro amico, fino a quando non raggiungeranno un elemento a cui il vostro amico è direttamente collegato.



Questo switch invierà quindi direttamente i dati al vostro amico, in base al suo indirizzo MAC. Cerchiamo quindi di capire nel dettaglio cosa succede.

Consideriamo lo switch e il router come un singolo dispositivo (utilizzato dalle reti moderne). Il router che avete a casa fa le funzioni sia di router che di switch.



## MAC Address e IP Address


Quanto segue si verifica quando inviate dati al vostro amico che potrebbe trovarsi in qualsiasi parte del mondo:

1. Il router X collegato direttamente al vostro nodo riceve i dati dal portatile.
2. Il router X individua nel pacchetto informativo l'indirizzo IP della destinazione.
3. Se il router X non è collegato direttamente alla destinazione, considera il percorso «**migliore**» verso la destinazione (secondo algoritmi di instradamento che vedrete nella teoria) e inoltra i dati a un altro router Y. Il router Y fa la stessa cosa e inoltra i dati a un altro router Z. Il processo continua fino a quando non raggiungiamo il router a cui è direttamente collegato il nodo di destinazione.
4. Si arriva quindi ad un punto in cui il router che attualmente ha i dati è direttamente collegato alla destinazione, vale a dire che il nodo di destinazione si trova all'interno della rete a cui appartiene il router corrente. Chiamiamo il router corrente come router A.



## MAC Address e IP Address

5. Questo router A ha una tabella, chiamata tabella **ARP** (**Address Resolution Protocol** ossia protocollo di risoluzione degli indirizzi), che mappa tutti gli indirizzi IP nella rete del router a tutti gli indirizzi MAC dei nodi collegati nella rete del router.



**R1's ARP Table**

<u>IP Address</u>		<u>MAC Address</u>
11.11.11.77	<--->	aaaa
22.22.22.2	<--->	bb22
22.22.22.88	<--->	bbbb

*PRAC NET .NET*

### Perché?

Ricordate che gli indirizzi IP sono gestiti dalla rete, quindi oggi il laptop del vostro amico potrebbe avere l'indirizzo IP 9.100.100.12 ma domani potrebbe averne uno differente, implicando che gli indirizzi IP non sono univoci. **Ciò che è unico sono gli indirizzi MAC.**

Pertanto, per determinare veramente lo stato di una rete, il router memorizza una mappatura che mappa tutti gli indirizzi IP che il router riconosce come sua rete, ai rispettivi indirizzi MAC dei nodi attualmente connessi al router.



# MAC Address e IP Address

Pertanto, il router A ha una tale mappatura  
 $9.100.100.12 \rightarrow \text{BB: BB: BB: BB: BB: BB: BB}$ .

Ogni volta che il router A riceve i dati destinati a  
9.100.100.12, sa che deve inviare i dati  
all'indirizzo MAC BB: BB: BB: BB: BB: BB.

In questo modo il router è in grado di  
consegnare al vostro amico il messaggio a lui  
destinato.

In breve, gli indirizzi IP vengono utilizzati per  
inviare dati tra reti diverse, mentre gli indirizzi  
MAC vengono utilizzati per inviare dati  
all'interno di una rete.

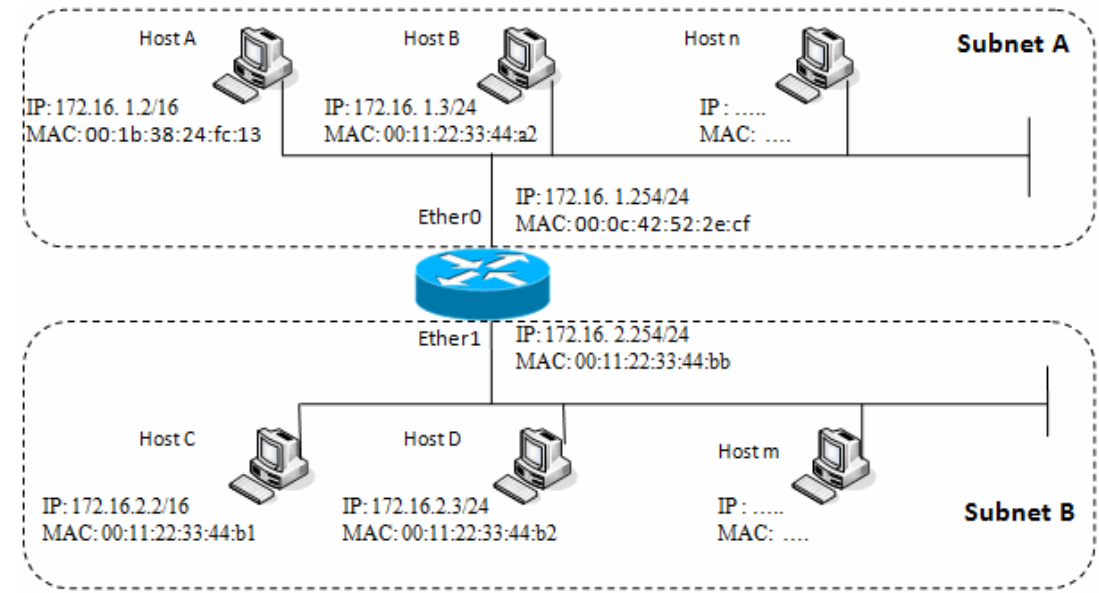


Figure 10.2. Network example for ARP proxy





# Esercizio

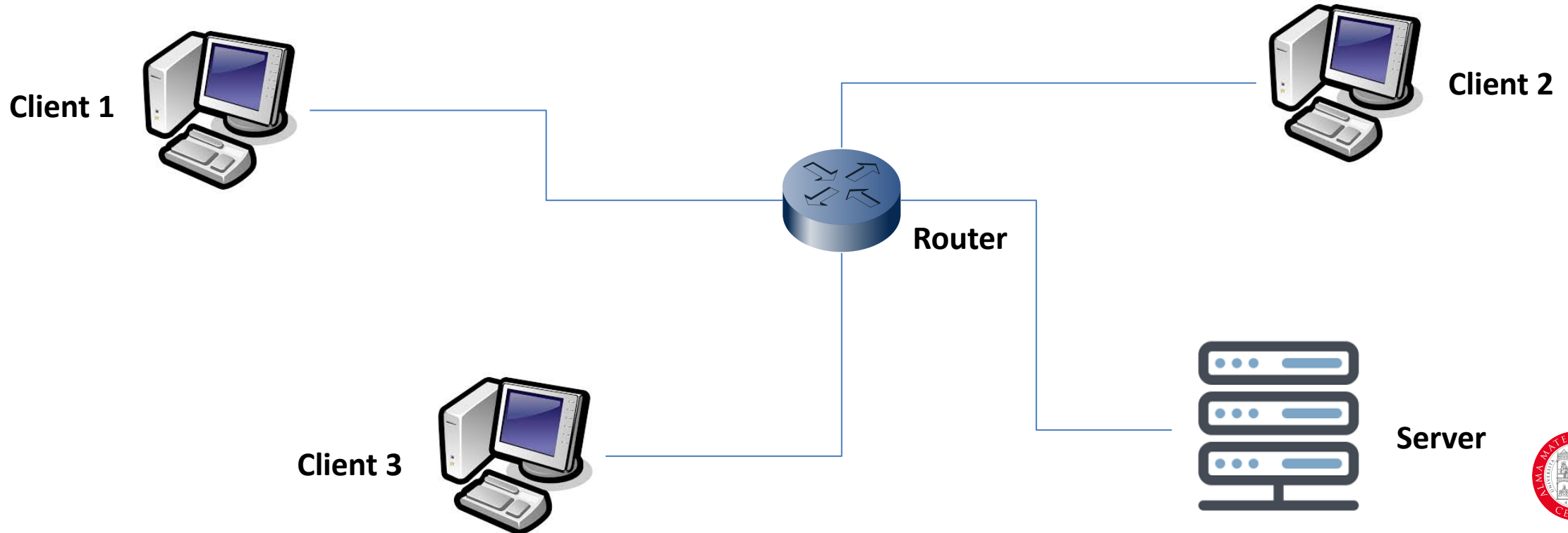
## Client – Router - Server



# Architettura

Abbiamo visto nelle passate lezioni come realizzare una semplice connessione TCP tra un server ed un client.

Ora vogliamo complicare un po' le cose cercando di capire come funziona una rete in cui i client e il server non siano direttamente connessi fra loro ma siano connessi ad un router, ossia qualcosa di simile alla rete in figura:



# Architettura

Il codice Python per client e server TCP li abbiamo già trattati nelle precedenti lezioni, anche se dovremo successivamente apportare qualche piccola aggiunta.

Quindi concentriamoci per il momento sul router.

Obiettivo di un router è quello di consentire a più client di connettersi ad esso e indirizzare un pacchetto dal server a uno degli altri client connessi.

Prima di proseguire, soffermiamoci sugli **HEADERS**.

I router decidono dove instradare il traffico in base alle intestazioni (headers), ovvero informazioni aggiuntive presenti nel messaggio originale utilizzate dai router per recapitare correttamente i pacchetti a destinazione.

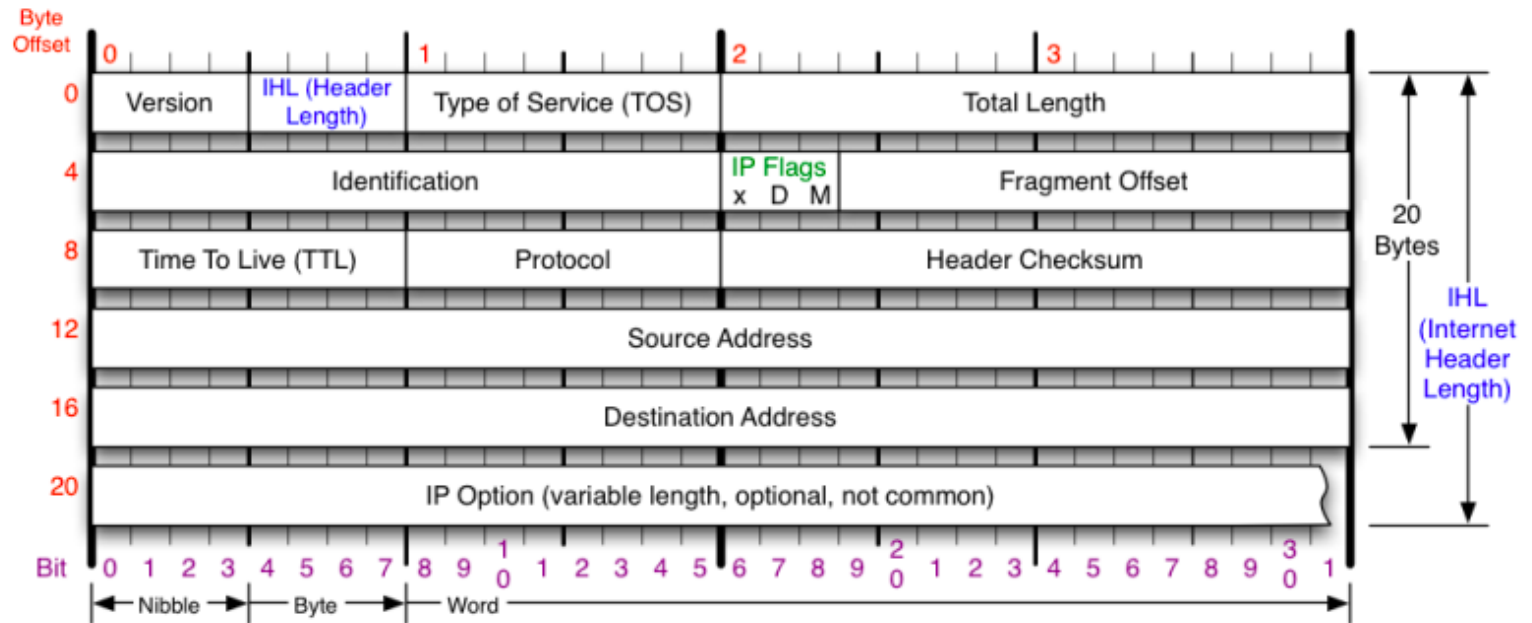


# Architettura

Gli Headers contengono molte informazioni, come ad esempio la **versione** che indica se si tratta del protocollo IPv4 o IPv6, **fragment flags** e **offset** che indicano se questo pacchetto ricevuto è effettivamente parte di un

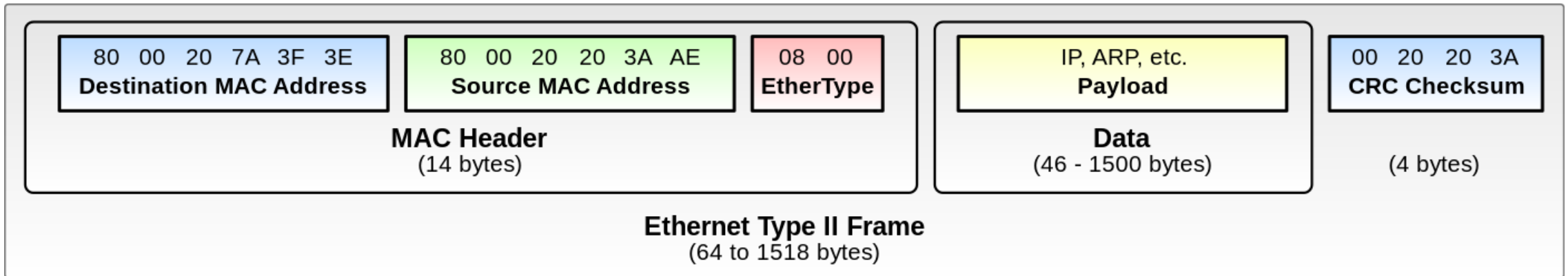
messaggio più grande e quindi la destinazione deve prevedere di ricostruire il messaggio originale da una serie di pacchetti, **header checksum** che indica alcuni codici di correzione degli errori.

Per la nostra applicazione, considereremo un'intestazione IP molto semplice, contenente **solo l'indirizzo di origine e l'indirizzo di destinazione**.



# Architettura

Allo stesso modo, c'è un altro Header, chiamato **Header Ethernet** (che incapsula informazioni relative agli indirizzi MAC).



Quindi nella nostra applicazione, costruiremo un pacchetto come segue:

*Packet* = Source MAC – Destination MAC – Source IP –  
Destination IP – Message

In Python: `packet = ethernet_header + IP_header + message`



# CLIENT



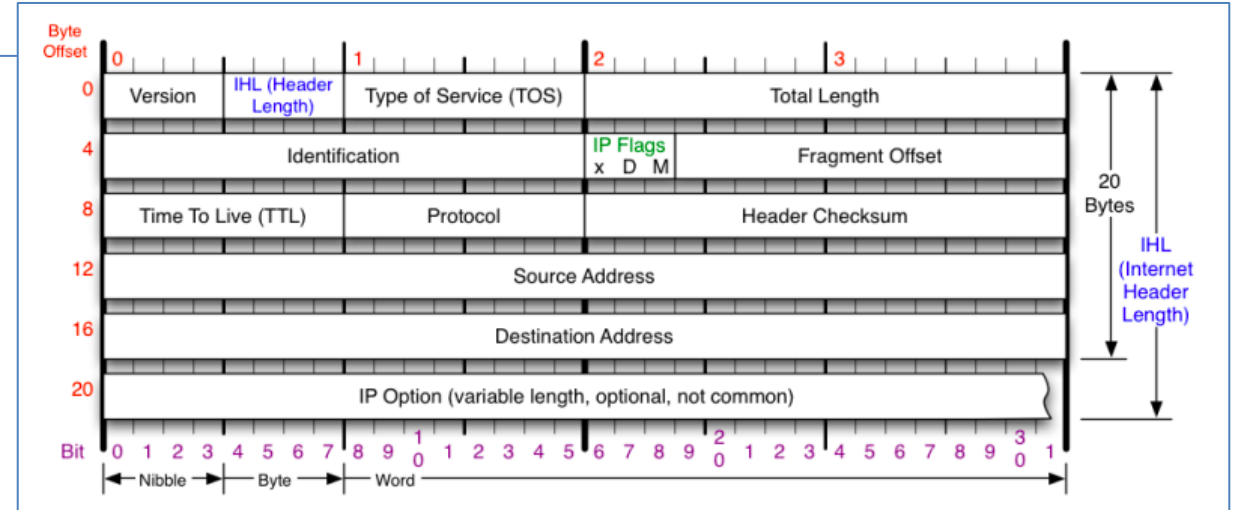
# Architettura - Client

```
'''
|-----CLIENT 1-----
Esercitazione 5 - Programmazione di Reti - Università di Bologna
G. Pau - A. Piroddi
'''
```

```
'''
import socket
import time
client1_ip = "92.10.10.15"
client1_mac = "32:04:0A:EF:19:CF"
router = ("localhost", 8200)
client1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
time.sleep(1)
client1.connect(router)
while True:
    received_message = client1.recv(1024)
    received_message = received_message.decode("utf-8")
    source_mac = received_message[0:17]
    destination_mac = received_message[17:34]
    source_ip = received_message[34:45]
    destination_ip = received_message[45:56]
    message = received_message[56:]
    print("\nPacket integrity:\ndestination MAC address matches client 1 MAC address: {mac}".format(mac=(client1_mac == destination_mac)))
    print("\ndestination IP address matches client 1 IP address: {mac}".format(mac=(client1_ip == destination_ip)))
    print("\nThe packed received:\n Source MAC address: {source_mac}, Destination MAC address: {destination_mac}".format(source_mac=source_mac, destination_mac=destination_mac))

    print("\nSource IP address: {source_ip}, Destination IP address: {destination_ip}".format(source_ip=source_ip, destination_ip=destination_ip))

    print("\nMessage: " + message)
```



Per motivi legati alla simulazione, forniamo indirizzi MAC e indirizzi IP **arbitrari** ai diversi client e al server, poiché stiamo eseguendo l'intera rete su un singolo nodo.

La cosa più semplice è definire prima i client: essi ricevono alcuni pacchetti dal router (più precisamente dal **socket** che emula il router), sezionano quel pacchetto in base alla struttura dei pacchetti discussa prima e visualizzano i contenuti a schermo.



## Architettura - Client

Per definire questo client, abbiamo impostato ***client1\_ip*** e ***client1\_mac***

```
client1_ip = "92.10.10.15"
client1_mac = "32:04:0A:EF:19:CF"
```

$2^8 \cdot 2^8 \cdot 2^8 \cdot 2^8 \rightarrow 8 + 8 + 8 + 8 = 32bit$

Abbiamo definito una specifica del **socket** del **router** (che sarà in seguito dettagliata nel codice del router)

```
router = ("localhost", 8200)
```

e abbiamo collegato questo client al router.

```
client1.connect(router)
```

Quando questo client riceve il messaggio, lo decodifica e seziona il pacchetto ricevuto.

```
received_message = client1.recv(1024)
received_message = received_message.decode("utf-8")
source_mac = received_message[0:17]
destination_mac = received_message[17:34]
source_ip = received_message[34:45]
destination_ip = received_message[45:56]
message = received_message[56:]
```

Prende i caratteri con  
indice da 0 a 16





## Architettura - Client

Nella nostra semplice applicazione relativa alla funzionalità di un router, abbiamo usato stringhe per gli indirizzi IP e MAC.

Generalmente, vedreste rispettivamente 32 e 48 bit,

$$2^8 \cdot 2^8 \cdot 2^8 \cdot 2^8 \Rightarrow 8bit + 8bit + 8bit + 8bit = 32bit$$

```
client1_ip = "92.10.10.15"
```

```
client1_mac = "32:04:0A:EF:19:CF"
```

$$8bit + 8bit + 8bit + 8bit + 8bit + 8bit = 48bit$$

ma dato che qui abbiamo appena incluso i numeri manualmente nelle stringhe, gli indirizzi IP e MAC sono indicizzati in base al carattere (ovvero la loro dimensione è uguale al numero di caratteri che hanno nella rappresentazione della stringa).

Quindi ci sono 17 caratteri in un indirizzo MAC (compresi i due punti) e 11 in un indirizzo IP (compresi i due punti).

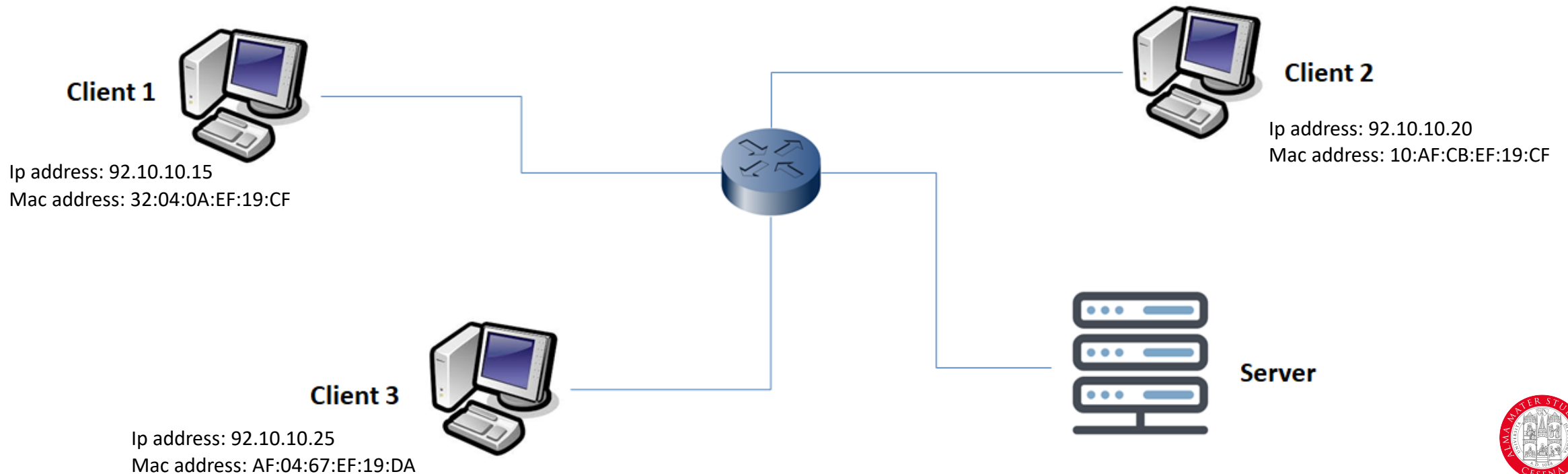


## Architettura - Client

Creiamo gli altri due client con lo stesso principio, assegnando loro indirizzamenti differenti:

```
client2_ip = "92.10.10.20"  
client2_mac = "10:AF:CB:EF:19:CF"
```

```
client3_ip = "92.10.10.25"  
client3_mac = "AF:04:67:EF:19:DA"
```



# SERVER

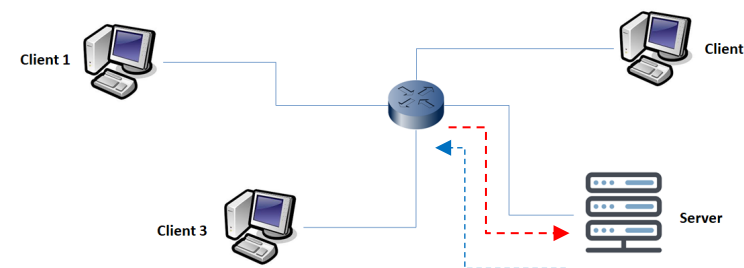


# Architettura - Server

Ora che abbiamo definito i client, passiamo a definire il server.

Il server funziona nel seguente modo (basato sui concetti di socket discussi nelle precedenti lezioni):

1. Il server riceve una richiesta di connessione dal router (dal socket 8100)
2. Il server crea un nuovo pacchetto
3. Il server invia un pacchetto al router



Vediamolo passo per passo.

Innanzitutto, consentiamo al server di ricevere una richiesta di connessione dal router.



# Architettura - Server

```
'''
-----SERVER-----
Esercitazione 5 - Programmazione di Reti - Università di Bologna
G. Pau - A. Piroddi

'''
import socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("localhost", 8000))
server.listen(2)
server_ip = "92.10.10.10"
server_mac = "00:00:0A:BB:28:FC"
router_mac = "05:10:0A:CB:24:EF"
while True:
    routerConnection, address = server.accept()
    if(routerConnection != None):
        print(routerConnection)
        break
while True:
    ethernet_header = ""
    IP_header = ""

    message = input("\nEnter the text message to send: ")
    destination_ip = input("Enter the IP of the clients to send the message to:\n1. 92.10.10.15\n2. 92.10.10.20\n3. 92.10.10.25\n")
    if(destination_ip == "92.10.10.15" or destination_ip == "92.10.10.20" or destination_ip == "92.10.10.25"):
        source_ip = server_ip
        IP_header = IP_header + source_ip + destination_ip

        source_mac = server_mac
        destination_mac = router_mac
        ethernet_header = ethernet_header + source_mac + destination_mac

        packet = ethernet_header + IP_header + message

        routerConnection.send(bytes(packet, "utf-8"))
    else:
        print("Wrong client IP inputted")
```

Associamo un nuovo **socket server** da **localhost** sulla **porta 8000** (ricordate che abbiamo stabilito che il router deve essere su 8200, come codificato nei nostri client). Forniamo al server un indirizzo IP arbitrario e un indirizzo MAC.

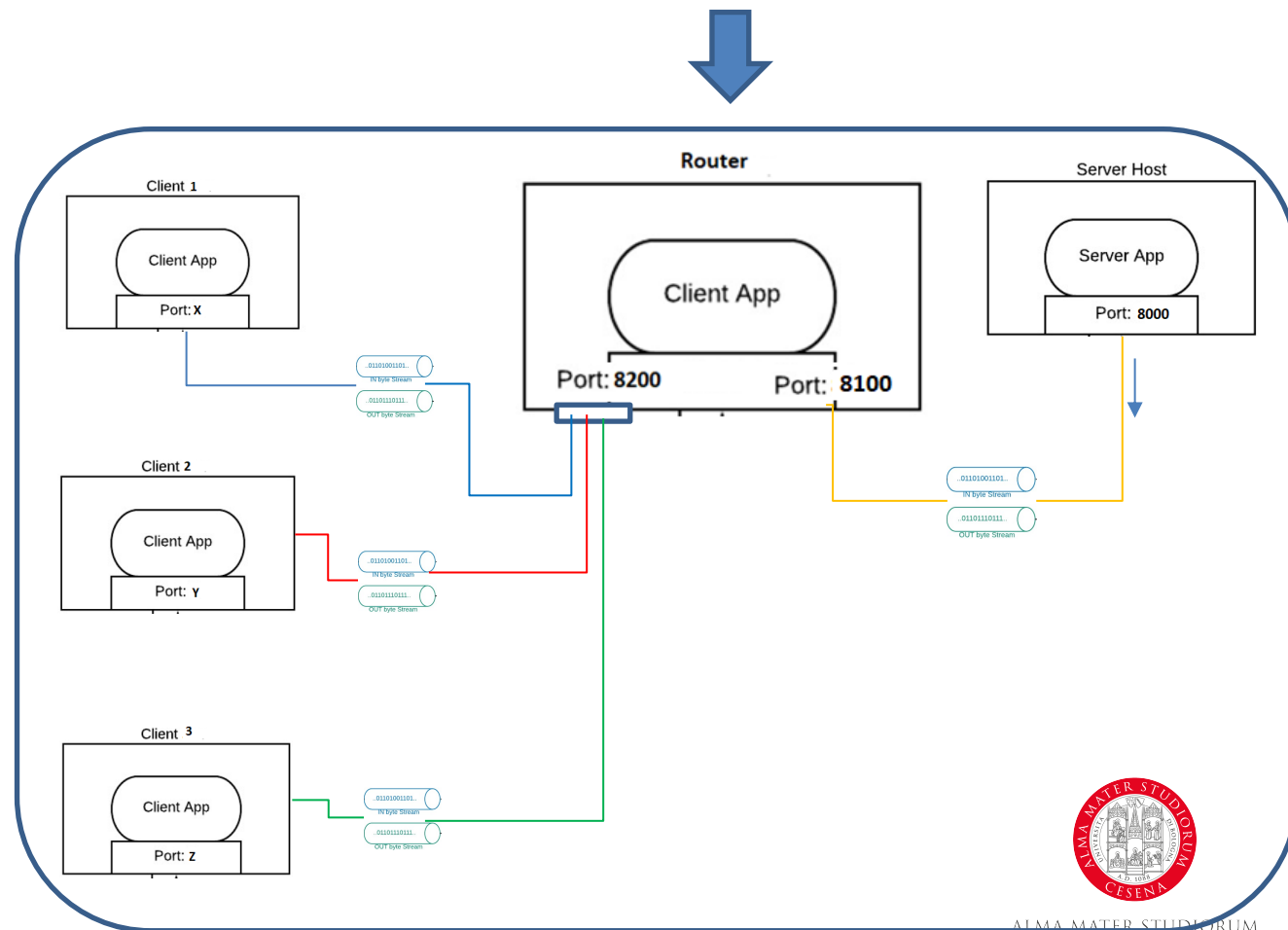
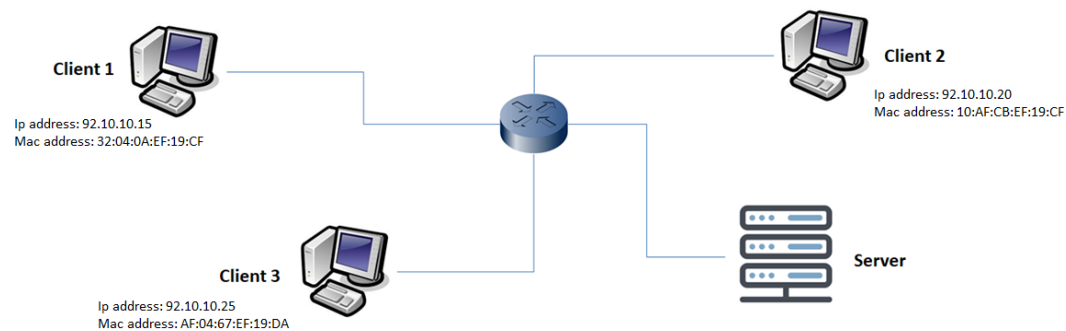


# Architettura - Server

Il server conosce anche l'indirizzo MAC del router.

## Perché?

Perché questo server è direttamente collegato al router (o meglio questo server appartiene alla rete aperta dal router sulla porta 8200; ricordate ancora che stiamo facendo tutto su un singolo nodo). Quindi questo server conosce l'indirizzo MAC di destinazione, che è l'indirizzo MAC del router.



# Architettura - Server

```
while True:
    routerConnection, address = server.accept()
    if(routerConnection != None):
        print(routerConnection)
        break
```

Il ciclo *while* serve per accettare una connessione dal router e di conseguenza uscire dal *while* quando è stata stabilita una connessione (la connessione rimane aperta).

Ora **creiamo un nuovo pacchetto da inviare al server**. Ricordiamo dalle slide precedenti che sulla struttura del pacchetto dobbiamo fare qualcosa del tipo:

*Packet* = Source MAC – Destination MAC – Source IP –  
Destination IP – Message

Abbiamo il MAC sorgente e l'IP sorgente (quello del server) e il MAC di destinazione (quello del router). **L'IP di destinazione, tuttavia, è del nodo a cui si sta inviando il pacchetto. Nel nostro caso, sarà uno dei tre client che abbiamo creato.**



# Architettura - Server

Qui, per semplicità, chiediamo all'utente di inserire un messaggio e l'IP di destinazione del client a cui inviare quel messaggio.

Notate come viene generato il pacchetto.

In termini generali, questo è il processo di generazione dei pacchetti, solo a un livello più complesso in cui dovrete creare i pacchetti come stabilito dalle intestazioni dei protocolli in uso.

```
while True:
    ethernet_header = ""
    IP_header = ""

    message = input("\nEnter the text message to send: ")
    destination_ip = input("Enter the IP of the clients to send the message to:\n1. 92.10.10.15\n2. 92.10.10.20\n3. 92.10.10.25\n")
    if(destination_ip == "92.10.10.15" or destination_ip == "92.10.10.20" or destination_ip == "92.10.10.25"):
        source_ip = server_ip
        IP_header = IP_header + source_ip + destination_ip

        source_mac = server_mac
        destination_mac = router_mac
        ethernet_header = ethernet_header + source_mac + destination_mac

        packet = ethernet_header + IP_header + message

        routerConnection.send(bytes(packet, "utf-8"))
    else:
        print("Wrong client IP inputted")
```

Qui, ***routerConnection*** è l'oggetto socket (8200) corrispondente alla connessione del router definita nel passaggio della progettazione del server.





# ROUTER



# Architettura - Router

Ora dobbiamo pensare a come deve essere fatto il router. Server → Router → Client 1/2/3.

Il router funziona così:

1. **attende che i clienti siano attivi (online).** Stabiliamo la regola che il router non accetti i pacchetti dal server a meno che i client non siano tutti online.
2. **Stabilisce una connessione al server** e riceve un pacchetto dallo stesso.
3. **Rimuove l'intestazione Ethernet** dal pacchetto ricevuto e ne crea uno nuovo.

Perché?

**Perché l'intestazione ethernet originale** aveva il **MAC di origine = MAC del server** e **MAC di destinazione = MAC del router**; ora le cose sono cambiate. **MAC di origine = MAC del router** e **MAC di destinazione = MAC del client**. Ora abbiamo tutto per creare la nuova intestazione Ethernet. **Ricordiamo**, tuttavia, che il pacchetto ricevuto **ha l'IP di destinazione del client** (non l'indirizzo MAC di destinazione) e quindi **dobbiamo mappare questo IP con l'indirizzo MAC**. Ecco a cosa serve la **tabella ARP** (che abbiamo visto prima) **che risiede già nel router**.

4. **Gira il pacchetto al client «interessato».**



# Architettura - Router

Per prima cosa definiamo i parametri del router e stabiliamo che i client siano online per procedere con eventuali attività di routing.

**Osservate che abbiamo due socket, una sulla porta 8100 e l'altra sulla porta 8200** all'interno di una singola applicazione. Questo è del tutto normale; infatti

**Server(8000) → (8100) router(8200) → client 1/2/3.**

Quindi una connessione è per ricevere un pacchetto dal server e l'altra connessione è per inviare un altro pacchetto ad un client. Come è possibile notare nei client, abbiamo la porta 8200.

```
'''
-----ROUTER-----
Esercitazione 5 - Programmazione di Reti - Università di Bologna
G. Pau - A. Piroddi
'''
import socket
import time

router = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
router.bind(("localhost", 8100))

router_send = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
router_send.bind(("localhost", 8200))

router_mac = "05:10:0A:CB:24:EF"

server = ("localhost", 8000)

client1_ip = "92.10.10.15"
client1_mac = "32:04:0A:EF:19:CF"
client2_ip = "92.10.10.20"
client2_mac = "10:AF:CB:EF:19:CF"
client3_ip = "92.10.10.25"
client3_mac = "AF:04:67:EF:19:DA"
router_send.listen(4)
client1 = None
client2 = None
client3 = None
while (client1 == None or client2 == None or client3 == None):
    client, address = router_send.accept()

    if(client1 == None):
        client1 = client
        print("Client 1 is online")

    elif(client2 == None):
        client2 = client
        print("Client 2 is online")

    else:
        client3 = client
        print("Client 3 is online")
```



## Architettura - Router

Abbiamo anche *router\_send.listen()* per le probabili connessioni dai client. Abbiamo anche un ciclo *while* che viene eseguito fino a quando tutti i client non sono connessi alla porta 8200. Gli oggetti socket vengono archiviati per ulteriore lavoro: invio di dati ricevuti dal server.

È ora il momento di definire la tabella ARP:

```
arp_table_socket = {client1_ip : client1, client2_ip : client2, client3_ip : client3}
arp_table_mac = {client1_ip : client1_mac, client2_ip : client2_mac, client3_ip : client3_mac}
```

Due dizionari:

1. Mappiamo l'IP di destinazione all'oggetto socket (del client) per inviare i dati e
2. Mappiamo l'IP di destinazione al MAC di destinazione per le informazioni relative all'Header Ethernet.



# Architettura - Router

Eseguiamo il secondo passaggio, stabiliamo una connessione al server e riceviamo un pacchetto da esso e in base alla definizione del pacchetto utilizzata sezioniamo il pacchetto

```
router.connect(server)
while True:
    received_message = router.recv(1024)
    received_message = received_message.decode("utf-8")

    source_mac = received_message[0:17]
    destination_mac = received_message[17:34]
    source_ip = received_message[34:45]
    destination_ip = received_message[45:56]
    message = received_message[56:]

    print("The packed received:\n Source MAC address: {source_mac}, Destination MAC address: {destination_mac}".format(source_mac=source_mac, destination_mac=destination_mac))
    print("\nSource IP address: {source_ip}, Destination IP address: {destination_ip}".format(source_ip=source_ip, destination_ip=destination_ip))
    print("\nMessage: " + message)
```



## Architettura - Router

**Il passaggio 3 è la funzionalità principale del router.** Consiste nel vedere se il nodo di destinazione si trova all'interno della rete direttamente connessa a questo router e quindi agire di conseguenza. Se il nodo non si trova nella rete, inviare il pacchetto a un altro router. Se il nodo si trova nella rete, rimuovere l'intestazione ethernet e crearne uno nuovo.

```
ethernet_header = router_mac + arp_table_mac[destination_ip]
IP_header = source_ip + destination_ip
packet = ethernet_header + IP_header + message

destination_socket = arp_table_socket[destination_ip]

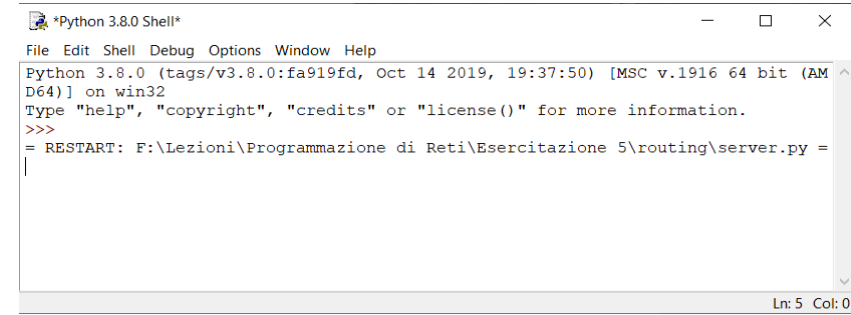
destination_socket.send(bytes(packet, "utf-8"))
time.sleep(2)
```



# TEST

Poiché il router richiede la connessione al server e i client richiedono la connessione al router, è necessario seguire il seguente ordine di richiamo degli script Python:

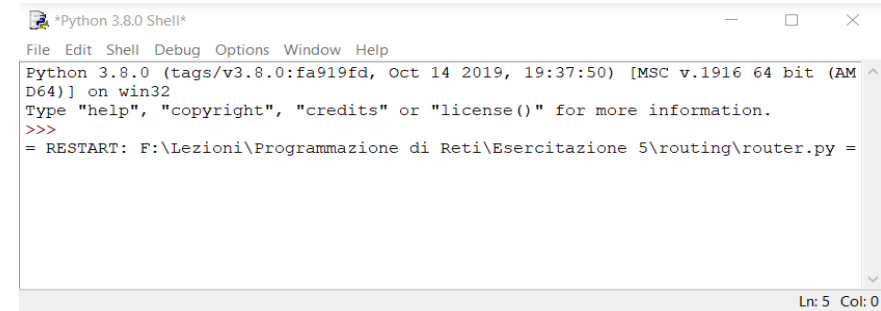
1. Avviate lo script del server



```
*Python 3.8.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:\Lezioni\Programmazione di Reti\Esercitazione 5\routing\server.py =
|
```

*Non accade nulla! Il server è ora nello stato in attesa che il router si connetta ad esso.*

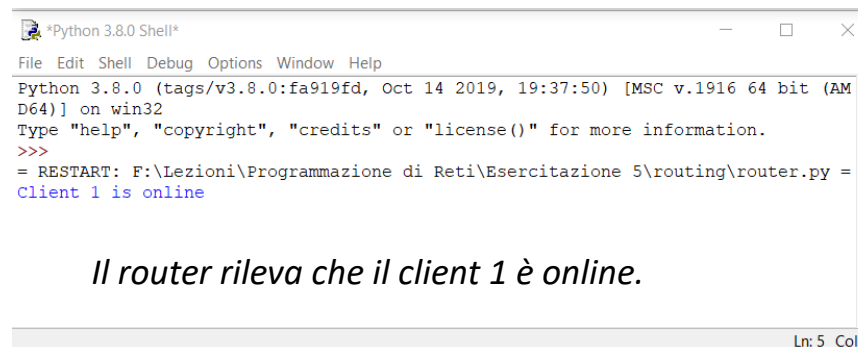
2. Avviate lo script del router



```
*Python 3.8.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:\Lezioni\Programmazione di Reti\Esercitazione 5\routing\rout...py =
```

*Non accade nulla! Il router è in attesa che i client si connettano ad esso.*

3. Avvia lo script dei client



```
*Python 3.8.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:\Lezioni\Programmazione di Reti\Esercitazione 5\routing\rout...py =
Client 1 is online
```

*Il router rileva che il client 1 è online.*



# TEST

Non appena tutti i client sono online, il router invia una richiesta di connessione al server e il server (già in attesa dello stesso) lo accetta (l'oggetto socket stampato a video è il socket router collegato). Il server richiede all'utente l'invio di alcuni messaggi sulla rete.

```
*Python 3.8.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:\Lezioni\Programmazione di Reti\Esercitazione 5\routing\router.py =
Client 1 is online
Client 2 is online
Client 3 is online
Ln: 5 Col: 0
```

*Quando anche i client 2 e client 3 sono online, il router li rileva*

```
*Python 3.8.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:\Lezioni\Programmazione di Reti\Esercitazione 5\routing\server.py =
<socket.socket fd=812, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 8000), raddr=('127.0.0.1', 8100)>
Enter the text message to send:
Ln: 5 Col: 0
```





# Test

Qui è il client 1 che riceve il messaggio a lui destinato

```
*Python 3.8.0 Shell*
File Edit Shell Debug Options Window Help
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:\Lezioni\Programmazione di Reti\Esercitazione 5\routing\client1.py

Packet integrity:
destination MAC address matches client 1 MAC address: True

destination IP address matches client 1 IP address: True

The packed received:
Source MAC address: 05:10:0A:CB:24:EF, Destination MAC address: 32:04:0A:EF:19:CF

Source IP address: 92.10.10.10, Destination IP address: 92.10.10.15

Message: ciao client 1
```

```
*Python 3.8.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:\Lezioni\Programmazione di Reti\Esercitazione 5\routing\client2.py
```

```
*Python 3.8.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: F:\Lezioni\Programmazione di Reti\Esercitazione 5\routing\client3.py
```

Il client 2 e 3 ovviamente non ricevono nulla

```
*Python 3.8.0 Shell*
File Edit Shell Debug Options Window Help
= RESTART: F:\Lezioni\Programmazione di Reti\Esercitazione 5\routing\server.py =
<socket.socket fd=812, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM,
proto=0, laddr=('127.0.0.1', 8000), raddr=('127.0.0.1', 8100)>

Enter the text message to send: ciao client 1
Enter the IP of the clients to send the message to:
1. 92.10.10.15
2. 92.10.10.20
3. 92.10.10.25
92.10.10.15

Enter the text message to send:

Ln: 14 Col: 32

*Python 3.8.0 Shell*
File Edit Shell Debug Options Window Help
= RESTART: F:\Lezioni\Programmazione di Reti\Esercitazione 5\routing\routerr.py =
Client 1 is online
Client 2 is online
Client 3 is online
The packed received:
Source MAC address: 00:00:0A:BB:28:FC, Destination MAC address: 05:10:0A:CB:24:EF

Source IP address: 92.10.10.10, Destination IP address: 92.10.10.15

Message: ciao client 1

Ln: 8 Col: 0
```

Inviare qualche messaggio attraverso il server.  
Ora vedete come le altre applicazioni ricevano informazioni aggiornate.  
Notate il router riceve entrambi i messaggi

