POLITECNICO
MILANO 1863

Fondamenti di Internet e Reti

Antonio Capone, Achille Pattavina,
Francesco Musumeci, Matteo Cesana

POLITECNICO
MILANO 1863

Programmazione Socket

Antonio Capone, Achille Pattavina,
Francesco Musumeci, Matteo Cesana

Attività di laboratorio: Versioni software

Gli esempi mostrati a lezione usano:

- Python versione 3
- Pycharm IDE education edition
    - include Python 3

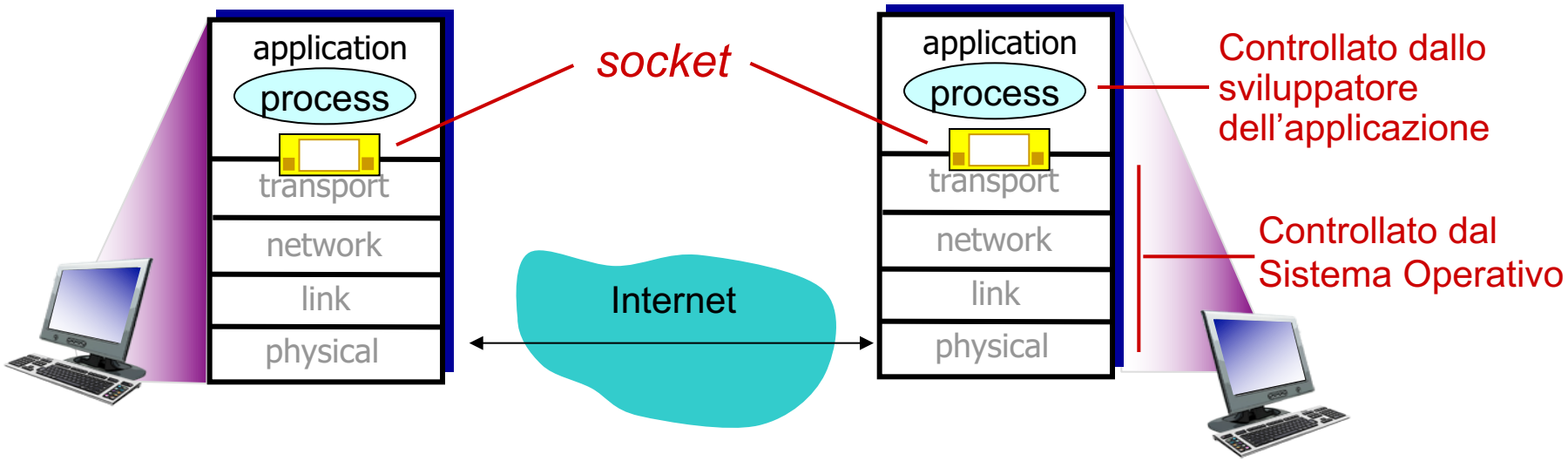
Tutti gli esempi sono testati nella macchina virtuale

- usiamo i socket per far comunicare processi in esecuzione sulla stessa macchina
- gli esempi funzionano altrettanto bene se i processi sono in esecuzione in due macchine distinte

Programmazione Socket

Obiettivo: imparare a sviluppare applicazioni client/server che comunicano utilizzando i sockets

Socket: porta tra il processo applicativo e il protocollo di trasporto end-to-end



Programmazione Socket

API = Application Programming Interface

Socket API

Introdotto in BSD4.1 UNIX, 1981

Creati, utilizzati e rilasciati esplicitamente dalle applicazioni

Paradigma client/server

Socket API offre due tipi di servizio di trasporto:

- UDP
- TCP

BSD = Berkeley Software Distribution

socket

È un'interfaccia (porta) *creata dall'applicazione e controllata dal SO* attraverso la quale un processo applicativo può *inviare e ricevere* messaggi a/da un altro processo applicativo

Programmazione Socket - Basi

- Il server deve essere in esecuzione prima che il client possa inviare dati ad esso (deamon)
- Il server deve avere un socket (porta) attraverso il quale riceve ed invia segmenti
- Allo stesso modo anche il client necessità di un socket
- Il client deve conoscere l'indirizzo IP del server e il numero di porta del processo server

Programmazione Socket *con UDP*

UDP: non c'è "connessione" tra client and server

Non c'è handshaking

Il mittente inserisce esplicitamente indirizzo IP e porta destinazione ad ogni segmento

Il SO inserisce l'indirizzo IP e la porta del socket origine ad ogni segmento

Il server può ricavare indirizzo IP e porta del mittente dai segmenti ricevuti

Punto di vista dell'applicazione

UDP fornisce trasporto non affidabile di gruppi di bytes all'interno di datagrammi scambiati tra client e server

Nota: il termine corretto per "pacchetto UDP" sarebbe "datagramma", ma in questa lezione useremo invece il termine "segmento UDP".

Esempio

Client:

- L'utente inserisce una riga di testo
- L'applicazione client invia la riga al server

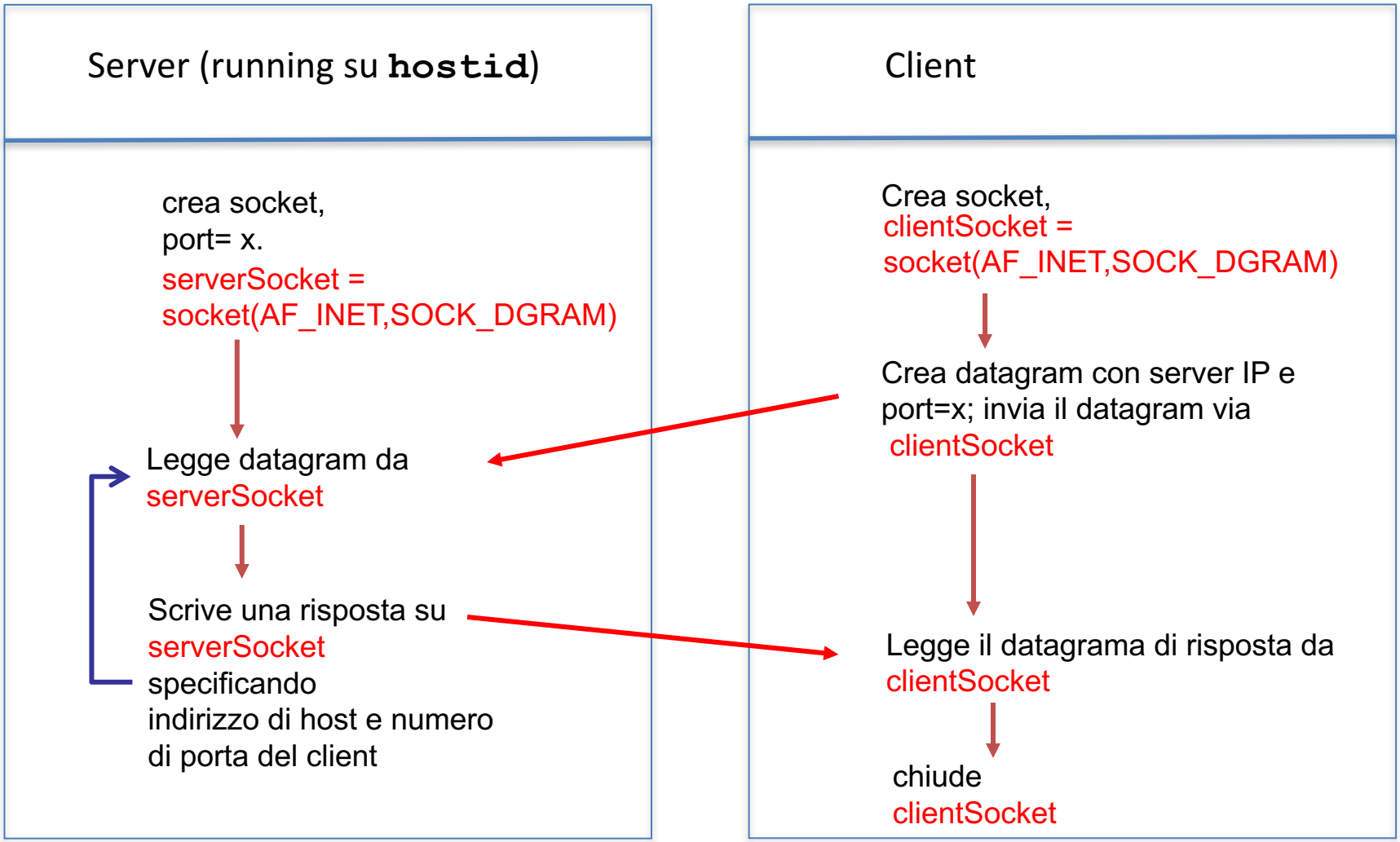
Server:

- Il server riceve la riga di testo
- Rende maiuscole tutte le lettere
- Invia la riga modificata al client

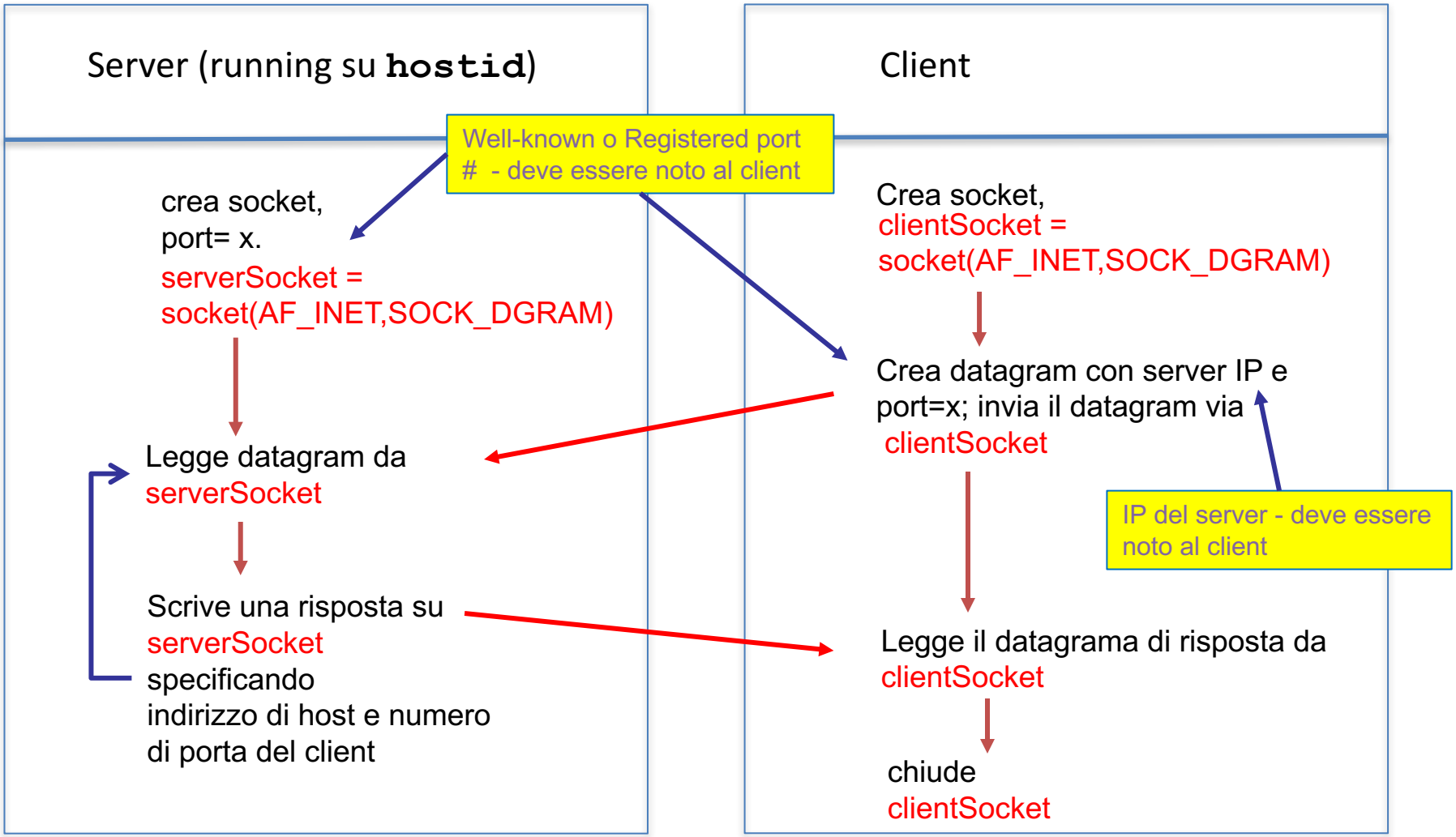
Client:

- Riceve la riga di testo
- La visualizza

Interazione tra socket Client/server: UDP



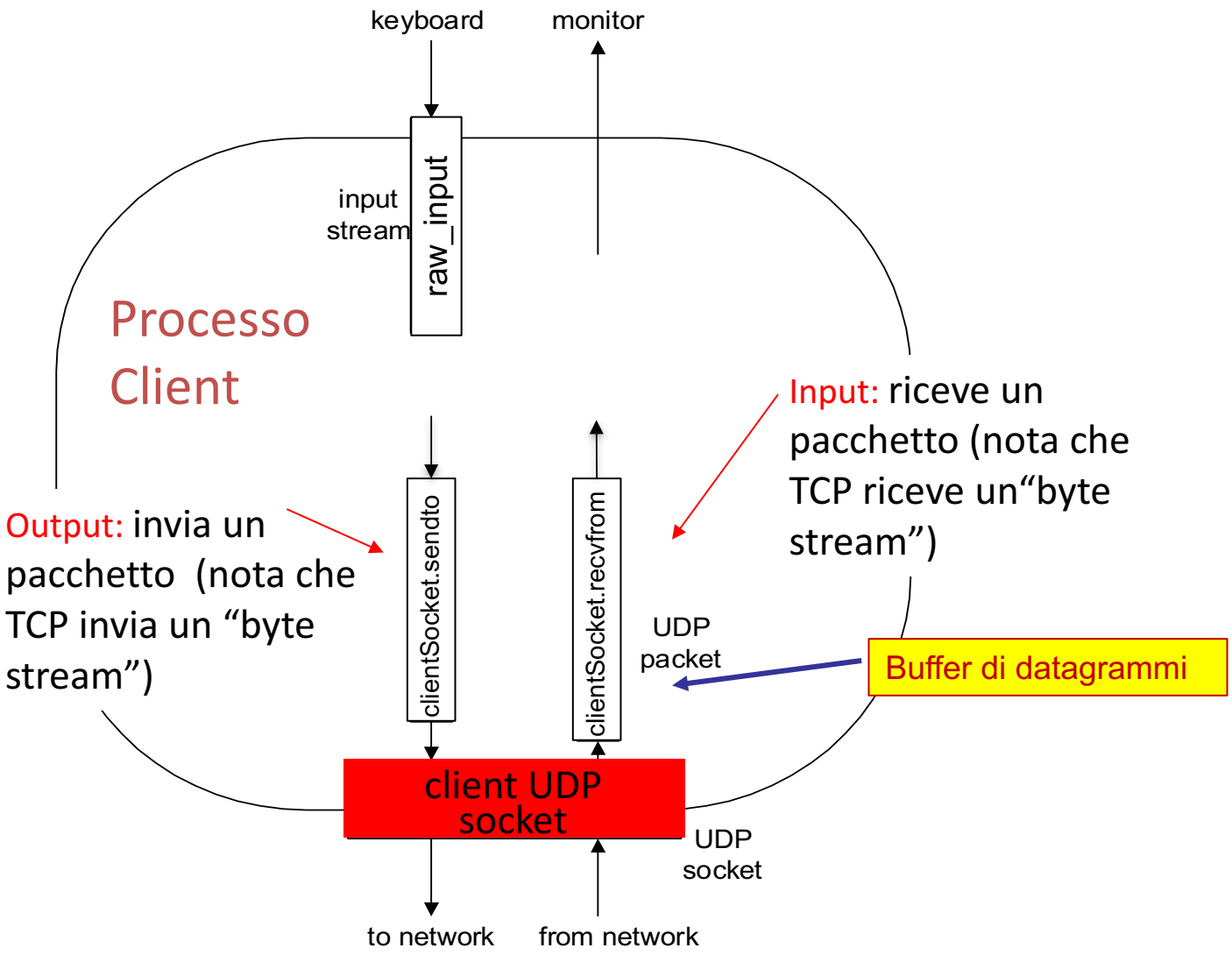
Interazione tra socket Client/server: UDP



serverSocket = socket(AF_INET,SOCK_DGRAM)

Nome oggetto (metodo) Nome classe

Esempio: Python client (UDP)



Applicazione esempio: *UDP client*

Include la libreria `Socket` di Python

```
from socket import *
serverName = 'localhost'
serverPort = 12000
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
message = input(Inserisci lettere:')
clientSocket.sendto(message.encode('utf-8'),(serverName, serverPort))
modifiedMessage, serverAddress =
                      clientSocket.recvfrom(2048)
modifiedMessage = modifiedMessage.decode('utf-8')
Print(modifiedMessage)
clientSocket.close()
```

Nome simbolico del server

12000: # porta processo server

Crea un socket UDP per il server

SOCK_DGRAM → UDP

Legge l'input da tastiera

Aggiunge il nome del server e la porta al messaggio; lo invia nel socket

Il socket si pone in ascolto (l'esecuzione procede quando arriva un segmento UDP)

Legge i caratteri della risposta del server dal socket e li memorizza in una stringa

Dimensione del buffer

Mostra la stringa ricevuta e chiude il socket

La chiamata al DNS per traduzione serverName (hostname) → IP server è fatta dal sistema operativo

Applicazione esempio: *UDP server*

Crea un socket UDP

Assegna il socket all porta locale 12000

loop infinito

Legge dal socket UDP message, memorizzando l'indirizzo del client (IP e porta)

Risponde al client con la stringa tutta in maiuscolo

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
Print("Il server è pronto a ricevere")
while 1:
    message, clientAddress =
        serverSocket.recvfrom(2048)
    message = message.decode('utf-8')
    modifiedMessage = message.upper()
    serverSocket.sendto
    (modifiedMessage.enc
```

12000: registered port #

Il socket si pone in ascolto (l'esecuzione procede quando arriva un segmento UDP)

Dimensione del buffer

I dati per costruire la risposta sono esplicitamente ricavati dal pacchetto UDP ricevuto

Elaborazione strettamente seriale dei pacchetti

UDP: osservazioni e domande

- Client e server utilizzando entrambi DatagramSocket
- IP destinazione e porta sono esplicitamente inseriti nel segmento
- Il client non può inviare un segmento al server senza conoscere l'indirizzo IP e la porta del server
- Il server può essere utilizzato da più di un client, ma tutti invieranno i pacchetti tramite lo stesso socket
  - La separazione dei dati da/a diversi client è un compito che spetta interamente al server

Eseguire il server

FIR2019 - [~/PycharmProjects/FIR2019] - .../Lab2_udp-server.py - PyCharm E

File Edit View Navigate Code Help

Project

FIR2019 ~/PycharmProjects/FIR2019
- lab.py
- Lab1_HTTP_Scripting.py
- Lab2-3_tcp-auto-client.py
- Lab2-3_tcp-auto-server.py
- Lab2-3_tcp-client.py
- Lab2-3_tcp-client-persistent.py
- Lab2-3_tcp-server.py
- Lab2-3_tcp-server-persistent.py
- Lab2-3_tcp-server-thread.py
- Lab2_udp-client.py
- Lab2_udp-client-error-management.p
- Lab2_udp-server.py

External Libraries

lab.py × | Lab2_udp-client.py × | Lab2_udp-server.py ×

```
from socket import *
serverPort = 12000

serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))

print("Il server è pronto a ricevere")

while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    [redacted]s)
    message = message.decode('utf-8')
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage.encode('utf-8'), clientAddress)
```

Paste	Ctrl+V
Paste from History...	Ctrl+Maiusc+V
Column Selection Mode	Alt+Maiusc+Ins
Refactor	▸
Folding	▸
Go To	▸
Run 'Lab2_udp-server'	Ctrl+Maiusc+F10
Step Through 'Lab2udp-server'	
Select 'Lab2_udp-server'	
Local History	▸
Create Gist...	

In pycharm il server si può lanciare dal menu contestuale (tasto destro sul nome dello script) oppure con la sequenza di tasti Ctrl+Maiusc+F10

Messaggi del server

FIR2019 - [~/PycharmProjects/FIR2019] - .../Lab2_udp-server.py - PyCharm Edu 3.5

File Edit View Navigate Code Help

Project

FIR2019 ~/PycharmProjects/FIR2019
- lab.py
- Lab1_HTTP_Scripting.py
- Lab2-3_tcp-auto-client.py
- Lab2-3_tcp-auto-server.py
- Lab2-3_tcp-client.py
- Lab2-3_tcp-client-persistent.py
- Lab2-3_tcp-server.py
- Lab2-3_tcp-server-persistent.py
- Lab2-3_tcp-server-thread.py
- Lab2_udp-client.py
- Lab2_udp-client-error-management.p
- Lab2_udp-server.py

External Libraries

lab.py × | Lab2_udp-client.py × | Lab2_udp-server.py ×

```python
from socket import *
serverPort = 12000

serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))

print("Il server è pronto a ricevere")

while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    print('[redacted]s)
    message = message.decode('utf-8')
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage.encode('utf-8'), clientAddress)
```

Run: Lab2_udp-server | Lab2_udp-client

```
Il server è pronto a ricevere
```

Verifica nell'elenco dei processi: netstat

```
user@user-VirtualBox: ~
File  Modifica  Schede  Aiuto
user@user-VirtualBox:~$ netstat -aut
Connessioni Internet attive (server e stabiliti)
Proto CodaRic CodaInv Indirizzo locale        Indirizzo remoto       Stato
tcp        0      0 localhost:63342         *:*                    LISTEN
tcp        0      0 *:pop3                  *:*                    LISTEN
tcp        0      0 user-VirtualBox:domain  *:*                    LISTEN
tcp        0      0 localhost:ipp           *:*                    LISTEN
tcp        0      0 *:smtp                  *:*                    LISTEN
tcp        0      0 localhost:6942          *:*                    LISTEN
tcp       86      0 10.0.2.15:42476         mail.stepik.org:https  CLOSE_WAIT
tcp6       0      0 [::]:http               [::]:*                 LISTEN
tcp6       0      0 [::]:ftp                [::]:*                 LISTEN
tcp6       0      0 ip6-localhost:ipp       [::]:*                 LISTEN
tcp6       0      0 [::]:smtp               [::]:*                 LISTEN
udp        0      0 *:12000                 *:*
udp        0      0 *:27367                 *:*
udp        0      0 user-VirtualBox:domain  *:*
udp        0      0 *:bootpc                *:*
udp        0      0 10.0.2.15:ntp           *:*
udp        0      0 localhost:ntp           *:*
udp        0      0 *:ntp                   *:*
udp6       0      0 [::]:44966              [::]:*
udp6       0      0 fe80::a00:27ff:feae:ntp [::]:*
udp6       0      0 ip6-localhost:ntp       [::]:*
udp6       0      0 [::]:ntp                [::]:*
user@user-VirtualBox:~$
```

Comando:
`netstat -aut`

Il server è in esecuzione

Eseguire il client

File Edit View Navigate Code Help

Project

FIR2019 ~/PycharmProjects/FIR2019
- lab.py
- Lab1_HTTP_Scripting.py
- Lab2-3_tcp-auto-client.py
- Lab2-3_tcp-auto-server.py
- Lab2-3_tcp-client.py
- Lab2-3_tcp-client-persistent.py
- Lab2-3_tcp-server.py
- Lab2-3_tcp-server-persistent.py
- Lab2-3_tcp-server-thread.py
- Lab2_udp-client.py
- Lab2_udp-client-error-management.p
- Lab2_udp-server.py

External Libraries

lab.py × | Lab2_udp-client.py × | Lab2_udp-server.py ×

```python
from socket import *

serverName = 'localhost'
serverPort = 12000

clientSocket = socket(AF_INET, SOCK_DGRAM)

message = input('Inserisci lettere:')
clientSocket.sendto(message.encode('utf-8'), (serverName, serverPort))

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
modifiedMessage = modifiedMessage.decode('utf-8')
print(modifiedMessage)

clientSocket.close()
```

- Paste Ctrl+V
- Paste from History... Ctrl+Maiusc+V
- Column Selection Mode Alt+Maiusc+Ins
- Refactor
- Folding
- Go To
- Run 'Lab2_udp-client' Ctrl+Maiusc+F10
- Step Through 'Lab2udp-client'
- Save 'Lab2_udp-client'
- Local History
- Create Gist...

Analogamente al server anche il client si può lanciare dal menu contestuale (tasto destro sul nome dello script).

Inserimento input

Project

FIR2019 ~/PycharmProjects/FIR2019

- lab.py
- Lab1_HTTP_Scripting.py
- Lab2-3_tcp-auto-client.py
- Lab2-3_tcp-auto-server.py
- Lab2-3_tcp-client.py
- Lab2-3_tcp-client-persistent.py
- Lab2-3_tcp-server.py
- Lab2-3_tcp-server-persistent.py
- Lab2-3_tcp-server-thread.py
- Lab2_udp-client.py
- Lab2_udp-client-error-management.p
- Lab2_udp-server.py

External Libraries

lab.py × | Lab2_udp-client.py × | Lab2_udp-server.py ×

```
from socket import *

serverName = 'localhost'
serverPort = 12000

clientSocket = socket(AF_INET, SOCK_DGRAM)

message = input('Inserisci lettere:')
clientSocket.sendto(message.encode('utf-8'), (serverName, serverPort))

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
modifiedMessage = modifiedMessage.decode('utf-8')
print(modifiedMessage)

clientSocket.close()
```

Run: Lab2_udp-server | Lab2_udp-client

```
Inserisci lettere:
```

Inserire la frase da convertire

Ricezione e visualizzazione output

File Edit View Navigate Code Help

Project

FIR2019 ~/PycharmProjects/FIR2019

- lab.py
- Lab1_HTTP_Scripting.py
- Lab2-3_tcp-auto-client.py
- Lab2-3_tcp-auto-server.py
- Lab2-3_tcp-client.py
- Lab2-3_tcp-client-persistent.py
- Lab2-3_tcp-server.py
- Lab2-3_tcp-server-persistent.py
- Lab2-3_tcp-server-thread.py
- Lab2_udp-client.py
- Lab2_udp-client-error-management.p
- Lab2_udp-server.py

External Libraries

lab.py × | Lab2_udp-client.py × | Lab2_udp-server.py ×

```python
from socket import *

serverName = 'localhost'
serverPort = 12000

clientSocket = socket(AF_INET, SOCK_DGRAM)

message = input('Inserisci lettere:')
clientSocket.sendto(message.encode('utf-8'), (serverName, serverPort))

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
modifiedMessage = modifiedMessage.decode('utf-8')
print(modifiedMessage)

clientSocket.close()
```

Output

Run: Lab2_udp-server | Lab2_udp-client

```
Inserisci lettere: ciao
 CIAO

Process finished with exit code 0
```

Stop del Server

FIR2019 - [~/PycharmProjects/FIR2019] - .../Lab2_udp-server.py - PyCharm Edu 3.5

File Edit View Navigate Code Help

Project

FIR2019 ~/PycharmProjects/FIR2019
- lab.py
- Lab1_HTTP_Scripting.py
- Lab2-3_tcp-auto-client.py
- Lab2-3_tcp-auto-server.py
- Lab2-3_tcp-client.py
- Lab2-3_tcp-client-persistent.py
- Lab2-3_tcp-server.py
- Lab2-3_tcp-server-persistent.py
- Lab2-3_tcp-server-thread.py
- Lab2_udp-client.py
- Lab2_udp-client-error-management.p
- Lab2_udp-server.py

External Libraries

lab.py × | Lab2_udp-client.py × | Lab2_udp-server.py ×

```
from socket import *
serverPort = 12000

serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))

print("Il server è pronto a ricevere")

while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    pr[redacted])
    message = message.decode('utf-8')
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage.encode('utf-8'), clientAddress)
```

Run: Lab2_udp-server | Lab2_udp-client

```
Il server è pronto a ricevere
```

Per fermare il server premere il pulsante rosso

Passi successivi

Il server è ancora in esecuzione. Il client può essere eseguito altre volte e il sistema operativo assegna una porta diversa ogni volta.

Esercizio: modificare il server in modo che scriva l’indirizzo IP e numero di porta del client.

Esercizio

FIR2019 - [~/PycharmProjects/FIR2019] - .../Lab2_udp-server.py - PyCharm Edu 3.5

File Edit View Navigate Code Help

Project

FIR2019 ~/PycharmProjects/FIR2019
- lab.py
- Lab1_HTTP_Scripting.py
- Lab2-3_tcp-auto-client.py
- Lab2-3_tcp-auto-server.py
- Lab2-3_tcp-client.py
- Lab2-3_tcp-client-persistent.py
- Lab2-3_tcp-server.py
- Lab2-3_tcp-server-persistent.py
- Lab2-3_tcp-server-thread.py
- Lab2_udp-client.py
- Lab2_udp-client-error-management.p
- Lab2_udp-server.py

External Libraries

lab.py × | Lab2_udp-client.py × | Lab2_udp-server.py ×

```python
from socket import *
serverPort = 12000

serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))

print("Il server è pronto a ricevere")

while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    print("Datagramma da: ", clientAddress)
    message = message.decode('utf-8')
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage.encode('utf-8'), clientAddress)
```

Run: Lab2_udp-server | Lab2_udp-client

```
Il server è pronto a ricevere
Datagramma da:  ('127.0.0.1', 38154)
```

porte UDP sorgente del client (per ogni riga un'esecuzione del client)

indirizzo IP del client
127.0.0.1 = loopback

Gestione degli errori

Vediamo cosa accade se inviamo dati (e aspettiamo dati) da un server inesistente.

Nel client modifichiamo la porta destinazione in 12001 (nessun processo è in ascolto su tale porta)

Di norma il server risponde con un messaggio di errore, che deve essere gestito dal client. Tuttavia il messaggio di errore potrebbe perdersi oppure arrivare quando il client è bloccato sulla `socket.recvfrom`. In tal caso il messaggio viene ignorato.

Due possibili esiti:

1) il messaggio di errore dal server diventa un'eccezione Python

2) il client aspetta all'infinito un messaggio dal server

Soluzione:

- impostare un timeout alle operazioni sui socket
- catturare le eccezioni Python
    - scadenza timeout
    - messaggi di errore

Impostare un timeout

Si usa il comando:

```
clientSocket.settimeout(<timeout>)
```

Può essere inserito ovunque prima della chiamata a `recvfrom`.

Il timeout è un valore con virgola espresso in secondi

Catturare le eccezioni

Gli errori di socket (compresi i timeout) lanciano eccezioni python

Per catturare l'eccezione racchiudere il codice che può lanciare l'eccezione in un blocco try … except

```
try:
   modifiedMessage, serverAddress =
   clientSocket.recvfrom(2048)
   print(modifiedMessage.decode('utf-8))
except:
    print("Timeout scaduti: Server non
raggiungibile"
finally:
   clientSocket.close()
```

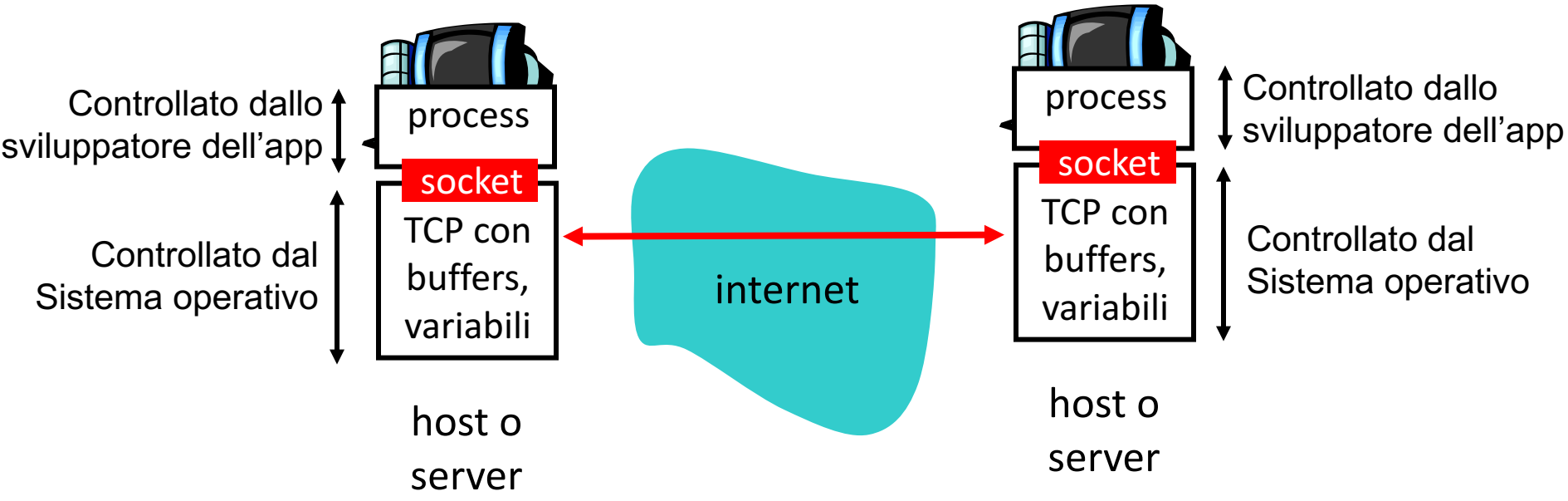
Esercizio 1

a) Scrivere un nuovo client UDP che non dia errore nel caso di server inesistente.

b) Scrivere un client UDP errato che termina prima di ricevere la risposta dal server. Cosa accade al server? Che modifiche sono necessarie?

Programmazione socket con TCP

Servizio TCP: trasferimento affidabile di **bytes** da un processo all'altro



Programmazione socket con TCP

Welcome socket: assomiglia un po' a socket UDP; può ricevere da tutti

Il client deve contattare il server

Il processo server deve essere eseguito per primo e deve aver creato

un socket (porta) per accogliere le richieste del client

Il client contatta il server:

Punto di vista dell'applicazione

TCP fornisce trasferimento affidabile e ordinato di bytes ("pipe") tra client eserver

Creando un socket TCP

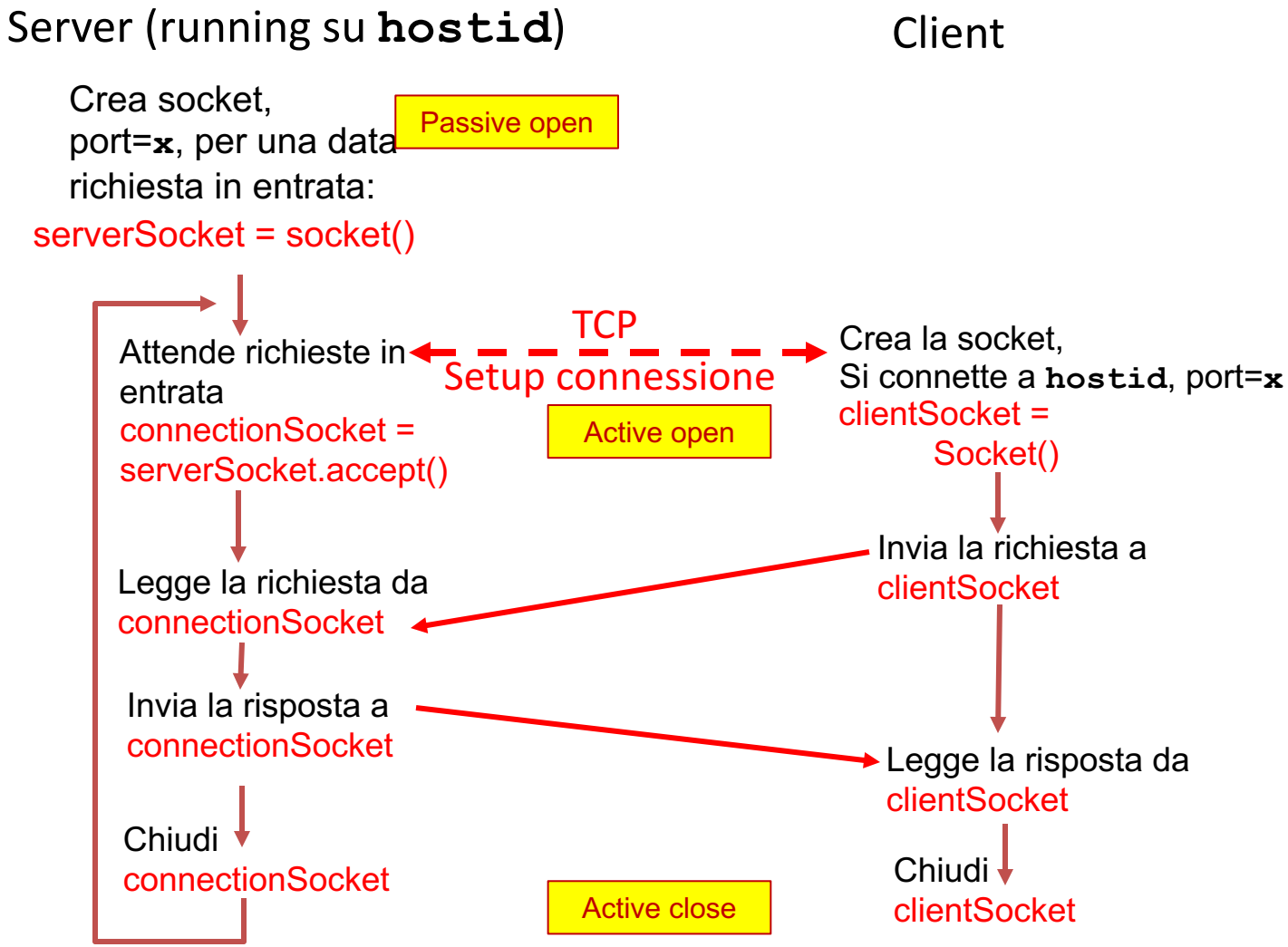
specificando indirizzo IP e numero di porta del processo server

Quando il client crea il socket: il client TCP instaura una connessione col server TCP

Quando contattato dal client, il server TCP crea un nuovo socket per la comunicazione tra processo server e client

- Permettendo al server di comunicare con diversi client
- Usando i numeri di porta sorgente per distinguere i client

Interazione tra socket Client/server: TCP

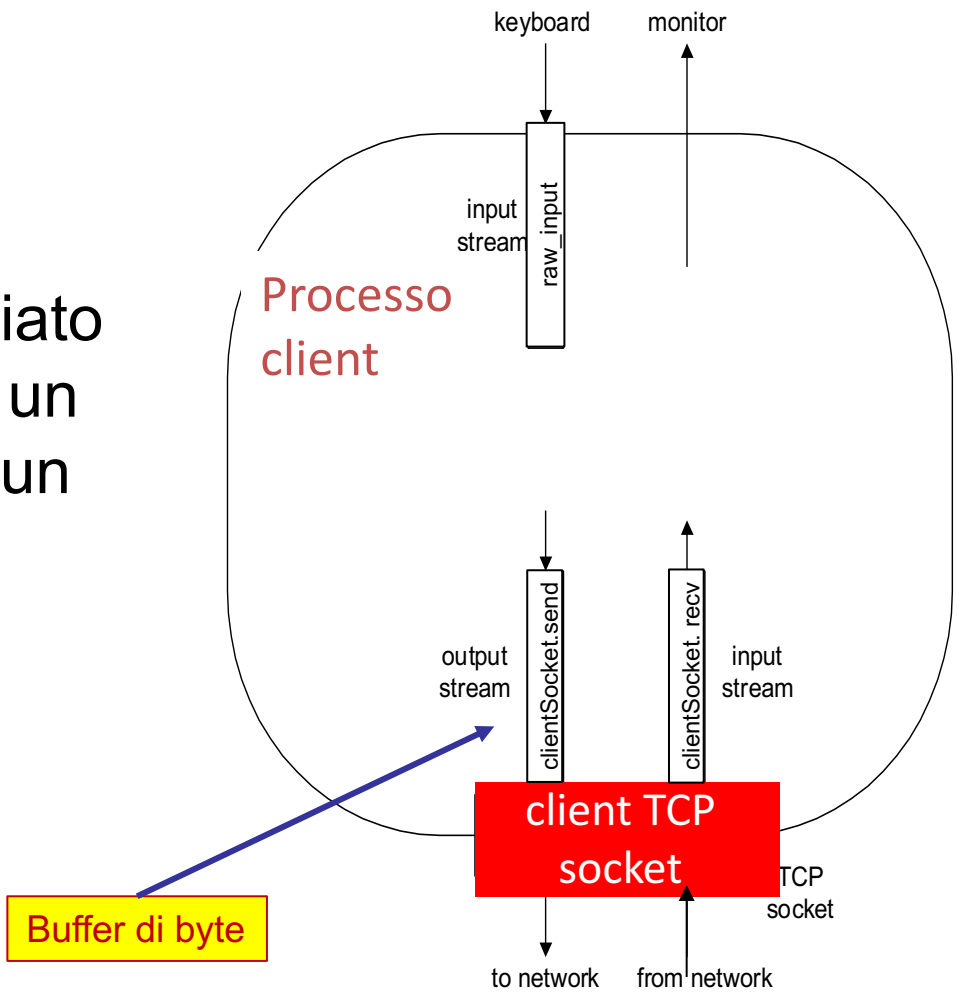


Stream jargon

Uno stream (flusso) è una sequenza di caratteri che scorrono in/fuori da un processo.

Un input stream viene associato ad una sorgente di input per un processo (es: una tastiera o un socket)

Un output stream viene associato ad una sorgente d'uscita (es: un monitor o un socket)



Programmazione socket con TCP

Esempio applicazione client-server:

1) Il client legge una riga dallo standard input (**`inFromUser`** stream) e la invia al server attraverso un socket (**`outToServer`** stream)

2) Il server legge una riga dal socket

3) Il server converte la riga in maiuscola e le invia al client

4) Il client legge la riga dal socket (**`inFromServer`** stream) e la mostra all'utente

Applicazione d'esempio: client TCP

Python TCP Client

```
from socket import *
serverName = [servername]
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = input('Inserisci lettere:')
clientSocket.send(sentence.encode('utf-8'))
modifiedSentence = clientSocket.recv(1024)

Print('Dal Server:', modifiedSentence.decode('utf-8'))
clientSocket.close()
```

Nome simbolico del server

Server port #

SOCK_STREAM → TCP

Crea un socket TCP verso il server remoto, porta 12000

Non è necessario appendere indirizzo e porta del server

Dimensione del buffer

La chiamata al DNS per traduzione hostname → IP server è fatta dal sistema operativo

Applicazione d'esempio: server TCP

Python TCP Server

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
Print('Il server è pronto a ricevere')
while 1:
    connectionSocket, clientAddress = serverSocket.accept()
    print('Connesso con: ', clientAddress)
    sentence = connectionSocket.recv(1024)
    sentence = sentence.decode('utf-8')
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode('utf-8'))
    connectionSocket.close()
```

Crea welcoming socket TCP

Il server si mette in ascolto per connessioni TCP in ingresso

Socket di ascolto (passive open)

loop infinito

All'arrivo di una active open (accept) la server socket crea una connection socket dedicata a quel client

Il server resta in attesa di richieste di connessione sull' accept(); a connessione instaurata viene ritornato un nuovo socket

Legge bytes dal socket (non legge anche l'indirizzo, come in UDP)

Chiude la connessione col client corrente (ma *non* il welcoming socket!)

Esercizio 1

Eseguire il server TCP e il client TCP

Connessioni persistenti

Una coppia client e server TCP possono usare una socket aperta più volte.

TCP non fornisce un meccanismo di delimitazione dei messaggi applicativi. Quindi un protocollo applicativo deve determinare:

- quando iniziano/finiscono i messaggi
- quando terminare la connessione

Nel nostro esempio ogni messaggio è terminato da un carattere «a capo» e la connessione va terminata quando il messaggio è un singolo punto «.»

Client persistente

```python
from socket import *

serverName = 'localhost'
serverPort = 12000

clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))

while True:
    sentence = input('Inserisci lettere ( . per fermare):')
    clientSocket.send(sentence.encode('utf-8'))
    if sentence == '.':
        break
    modifiedSentence = clientSocket.recv(1024)
    print('Dal Server:', modifiedSentence.decode('utf-8'))

clientSocket.close()
```

Server persistente

```python
from socket import *


serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)

serverSocket.bind(('', serverPort))
serverSocket.listen(1)

while True:
    print('Il server è pronto a ricevere')
    connectionSocket,  clientAddress = serverSocket.accept()
    print("Connesso con: ", clientAddress)
    while True:
        sentence = connectionSocket.recv(1024)
        if sentence.decode('utf-8') == '.':
            break
        capitalizedSentence = (sentence.decode('utf-8')).upper()
        connectionSocket.send(capitalizedSentence.encode('utf-8'))
    connectionSocket.close()
```

Esercizio 2

a) Eseguire il client persistente e inviare messaggi multipli
b) Senza chiudere il primo client, aprirne un secondo e poi un terzo. Cosa accade?
c) Inviare un messaggio con il secondo client. Cosa accade?
d) Terminare il primo client. Cosa accade ora al secondo client?

Coda delle connessioni incomplete

Le richieste di apertura di connessione si accodano in una coda di connessioni incomplete, che prende il nome di **backlog**. Il comando accept preleva la prima connessione incompleta e crea una socket.

Con il comando:

```
serverSocket.listen(1)
```

abbiamo istanziato un **backlog** di lunghezza 1+1 (il numero massimo di connessioni inomplete è pari ad 1 più il valore specificato nell'argomento del metodo listen).

Quando il server sta servendo il primo client, la **seconda** e la **terza** richiesta rimangono in **coda**, mentre la **quarta** viene **rifiutata**.

Multithreaded servers

Per gestire client multipli il server deve attivare più worker:

- un worker per gestire le richieste di connessione entranti (listening socket)
- un worker per gestire ciascuna connessione (active socket)

Due paradigmi:

- ogni worker è un thread
    - facile e veloce
    - alcune limitazioni (numero max di thread paralleli, tutti i thread hanno gli stessi permessi etc…)
- ogni worker è un processo
    - aggiunge complessità e latenza

Esiste un terzo paradigma: un solo worker che serve a turno un messaggio da ogni socket (molto efficiente, ma più complicato da gestire).

Multithreaded servers in python

```python
from socket import *
from threading import Thread


def handler(connectionSocket):
    while True:
        sentence = connectionSocket.recv(1024)
        if sentence.decode('utf-8') == '.':
            break
        capitalizedSentence = (sentence.decode('utf-8')).upper()
        connectionSocket.send(capitalizedSentence.encode('utf-8'))
    connectionSocket.close()


serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
serverSocket.bind(('', serverPort))
serverSocket.listen(1)

while True:
    print('Il server è pronto a ricevere')
    newSocket,  addr = serverSocket.accept()
    thread = Thread(target=handler, args=(newSocket,))
    thread.start()
```

Il codice per gestire ogni connessione attiva va in una funzione

Bisogna consentire al socket locale di usare più volte la stessa porta

Ogni connessione viene servita in un nuovo thread

Esercizio 3

Scrivere un client che invia 100 caratteri, usando una chiamata send alla volta.

Scrivere un server che scriva il numero di caratteri ricevuti ad ogni recv.

a) Cosa accade?

b) Inserire una breve pausa dopo ogni send. Cosa accade?