Programmazione su Rete in Python

Advanced Computer Programming

Prof. Luigi De Simone

Summary

- **Argomenti**
  - Cenni su Internet
  - Concetto di Socket
  - Socket TCP ed UDP
  - Server Multi-thread

- **Riferimenti**
  - https://docs.python.org/3/library/socket.html#socket-objects

Internet

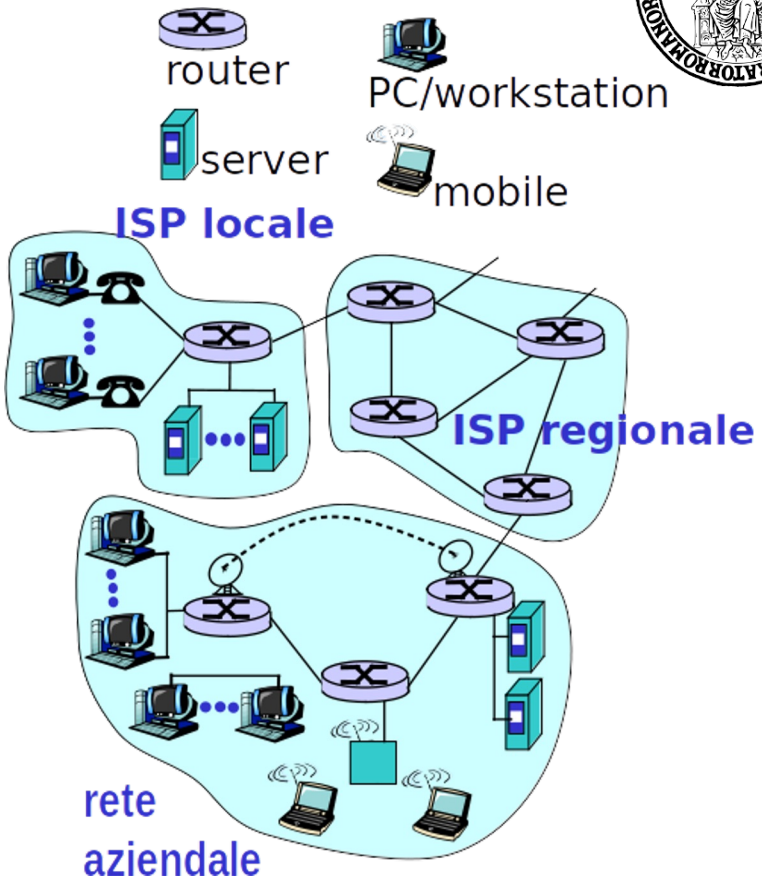
- Rete di estensione mondiale basata sul protocollo TCP/IP
  - Progenitrice: **ARPANET**, 1969 finanziata dall'ARPA (Advanced Research Project Agency), agenzia del Dipartimento della Difesa USA
  - In origine creata per esigenze militari e per connettere università e centri di ricerca

- Oggi costituisce uno strumento per:
  - scambiare e cercare informazioni
  - sviluppare sistemi distribuiti
  - connettere centinaia di milioni di dispositivi

Internet

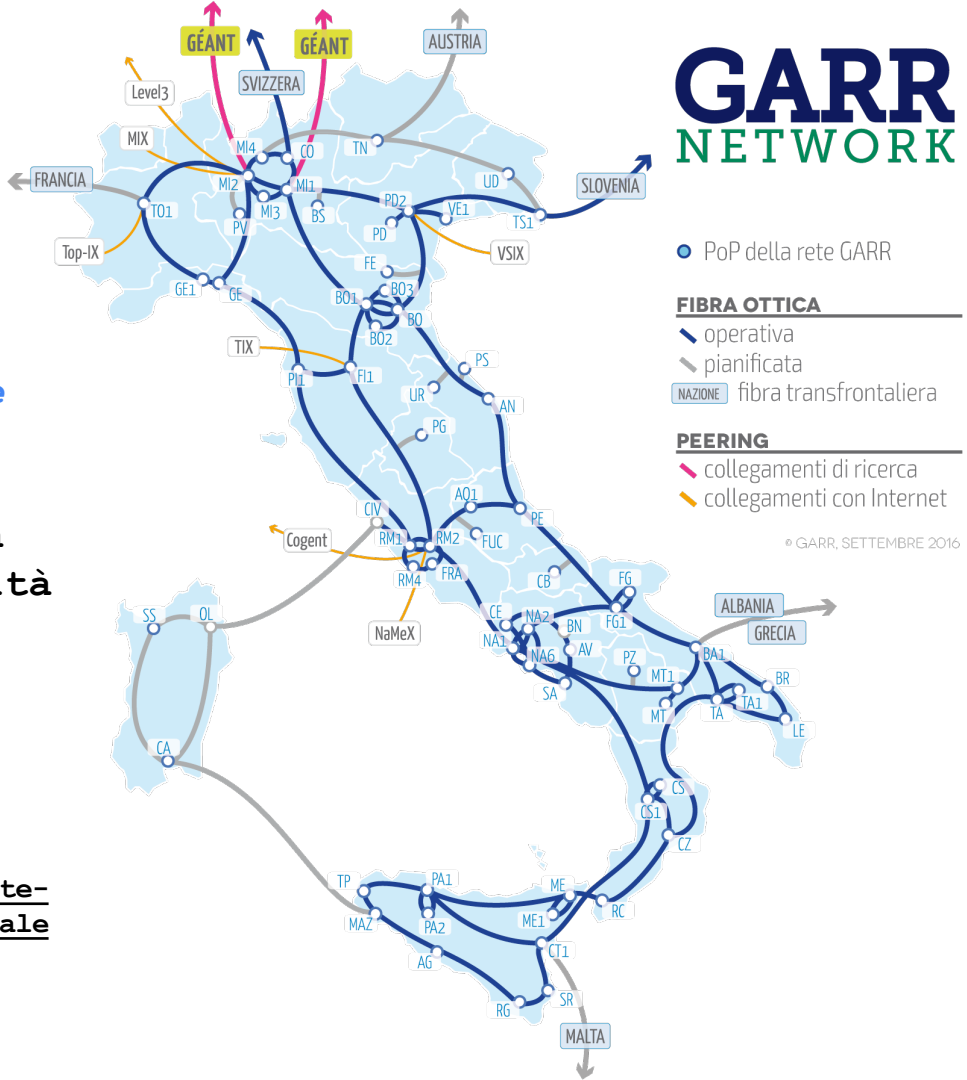
- Internet è una rete di reti

- Componenti principali:
  - Dispositivi connessi detti **nodi** o **host**:
    - PC, server, smartphone, tablet, …
  - Link di comunicazione:
    - cavi, fibra ottica, radio, satellitari,…
  - **Router**:
    - dispositivi che instradano i messaggi attraverso la rete



Una rete di reti

GARR (*Gruppo per l'Armonizzazione delle Reti della Ricerca*): rete in fibra ottica dedicata alla comunità italiana dell'università e della ricerca



http://www.garr.it/it/infrastrutture/rete-nazionale/infrastruttura-di-rete-nazionale

Gli indirizzi IP

- Ad ogni nodo è assegnato un indirizzo Internet (**indirizzo IP**)
  - **IPv4:** formato da 32 bit
    - circa 4,3 miliardi di indirizzi diversi
  - **IPv6:** indirizzi a 128 bit
    - 655571 miliardi di miliardi

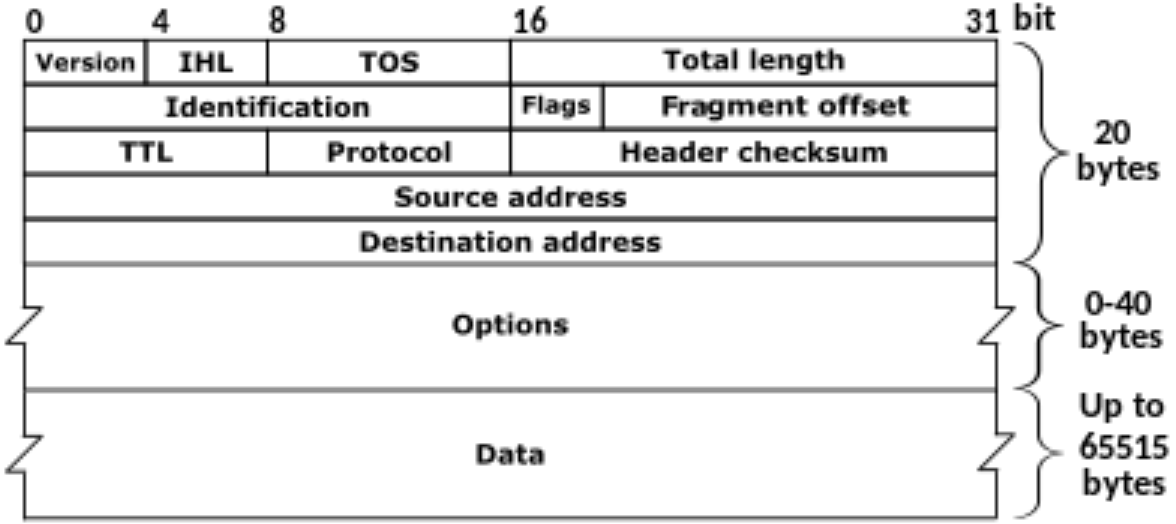
- Un indirizzo IP è suddiviso in:
  - **IPv4**: 4 campi da 8 bit, separati da un punto
    - 192.168.0.4
  - **IPv6**: 8 campi da 16 bit, separati da ":"
    - 2001:0db8:85a3:0000:1319:8a2e:0370:7344

- Grazie agli indirizzi IP è possibile identificare un computer sulla rete ed inviargli dati
  - Tipicamente ci si riferisce agli host remoti con nomi simbolici, es. www.unina.it
  - La traduzione nome/indirizzo è realizzata dal DNS (Domain Name Service)

IP e instradamento (routing)

- I dati end-to-end sono trasmessi in pacchetti (**IP datagram**)
  - Dimensione massima **65.535** byte

- **Servizio best-effort**: i pacchetti potrebbero giungere a destinazione *corrotti*, *duplicati*, *non-in-ordine* o anche essere **persi**

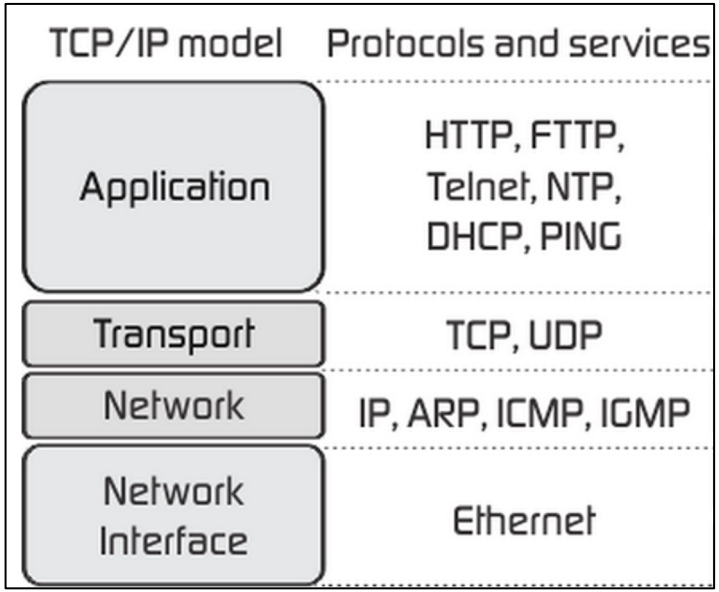


TCP e UDP

TCP (Transmission Control Protocol) implementa un trasporto affidabile da sorgente a destinazione su IP:

- Fornisce un flusso di byte, full-duplex
- **Orientato alla connessione**:
  - Instaurazione, utilizzo, e chiusura di una **connessione**
- Preserva l'ordine, eliminando duplicati
- Implementa la ri-trasmissione dei pacchetti

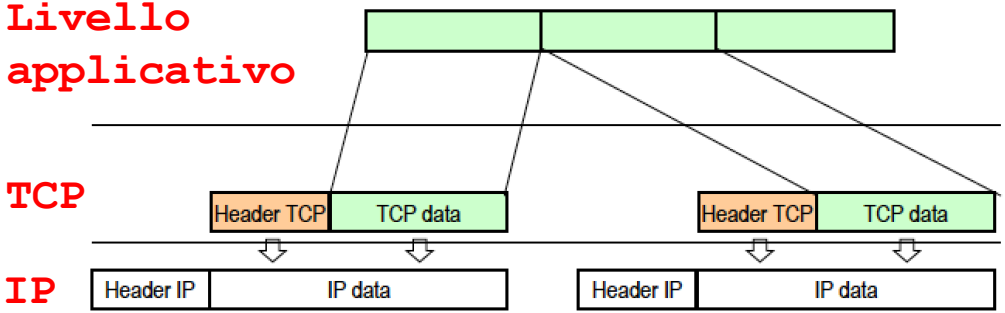
NOTA: il livello IP è Best-effort!

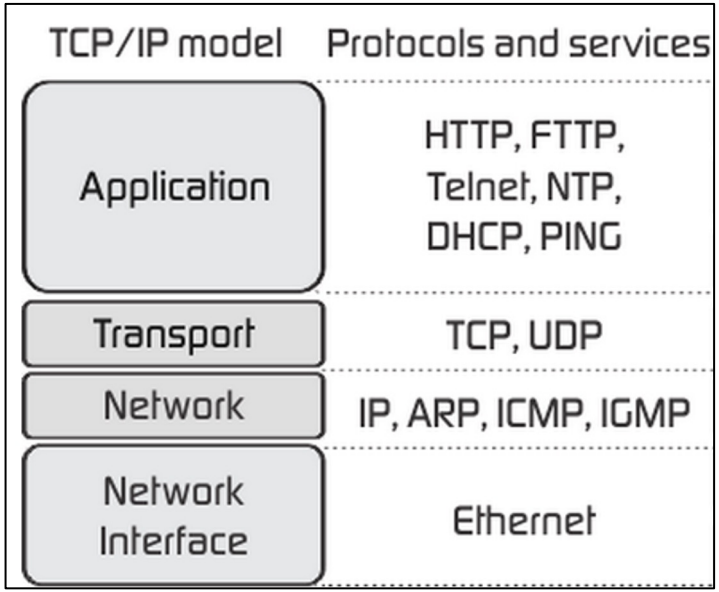


TCP e UDP

TCP (Transmission Control Protocol) implementa un trasporto affidabile da sorgente a destinazione su IP:

- Fornisce un flusso di byte, full-duplex
- **Orientato alla connessione**:
  - Instaurazione, utilizzo, e chiusura di una **connessione**
- Preserva l'ordine, eliminando duplicati
- Implementa la ri-trasmissione dei pacchetti

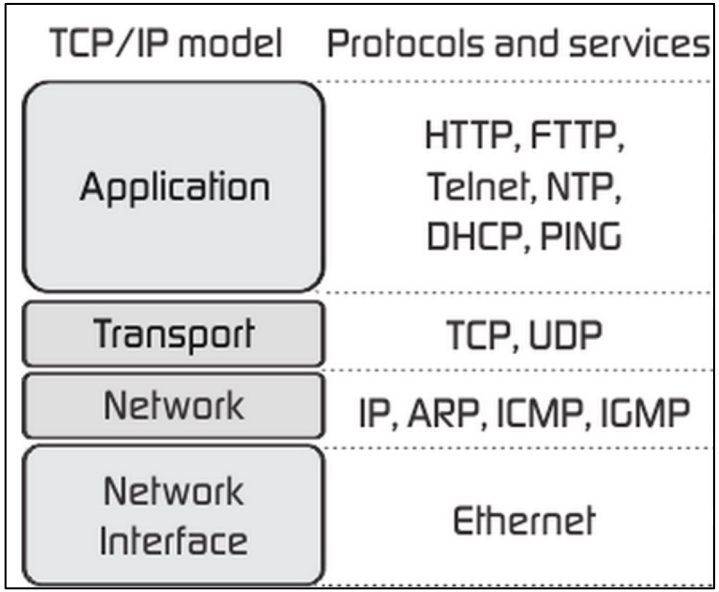




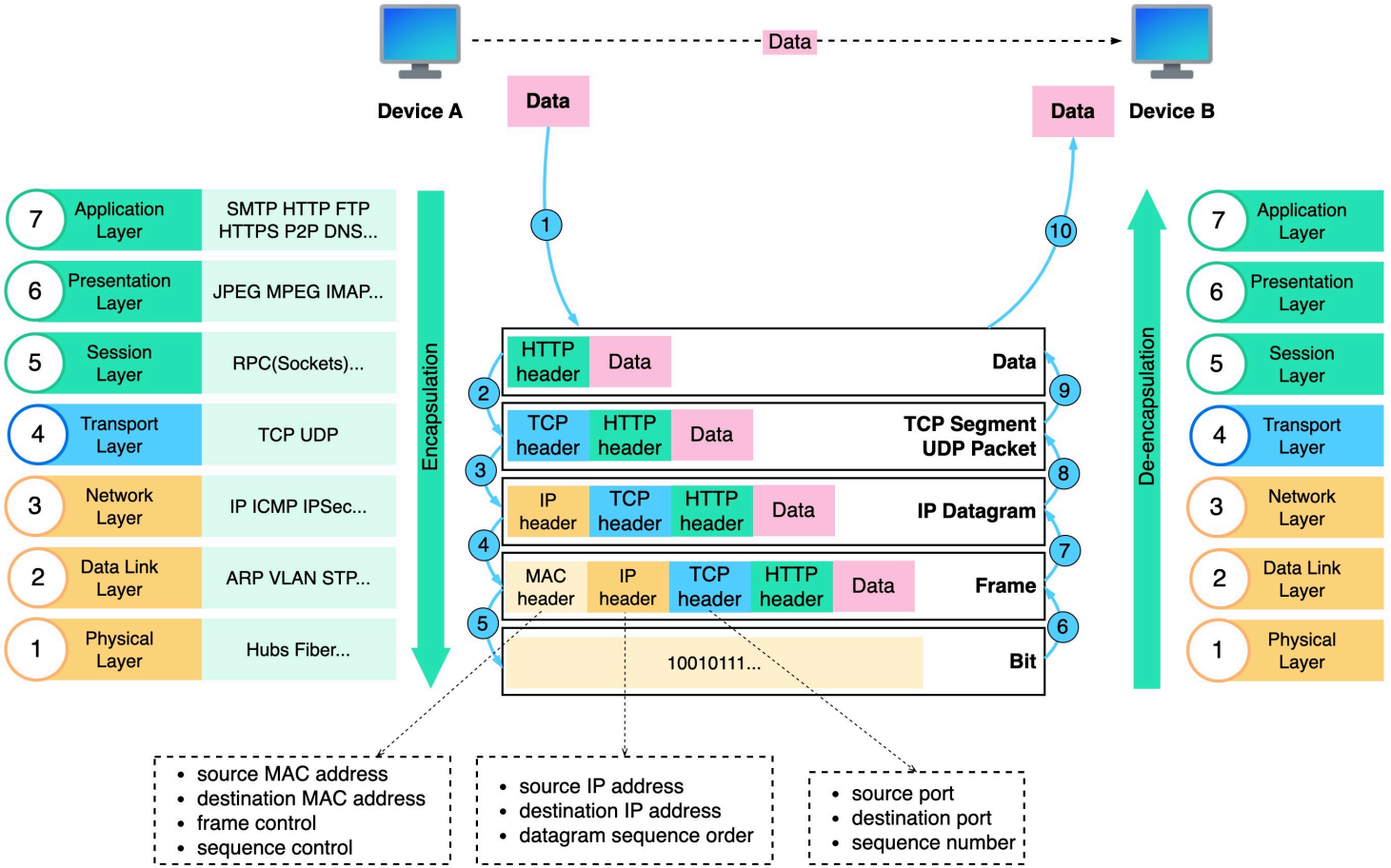
TCP e UDP

UDP (User Datagram Protocol):

- Protocollo di trasporto **non affidabile**
- Molto simile al protocollo IP, ma offre la capacità di **distinguere tra porte differenti** nello stesso host
- Basato sul concetto di **datagramma**



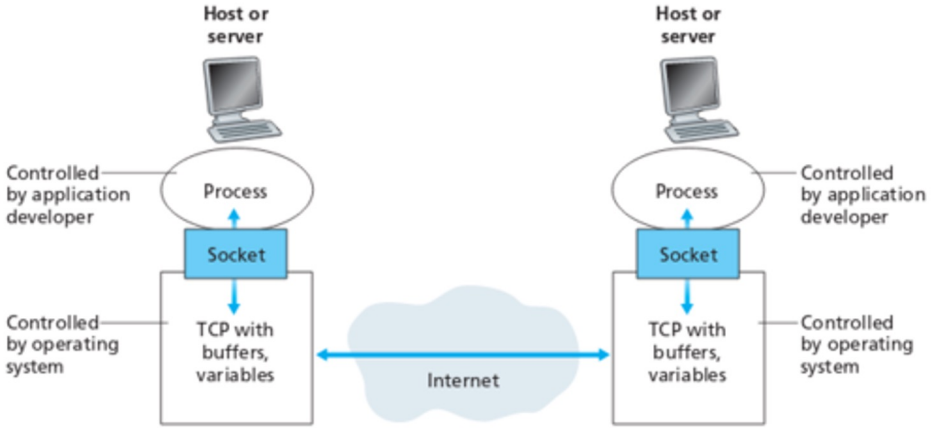
Dati attraverso lo stack ISO/OSI



Socket

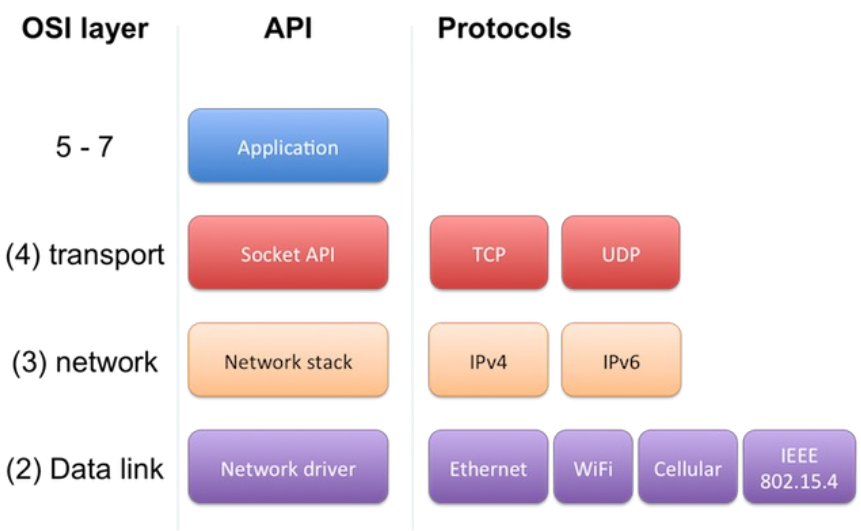
- Meccanismo di comunicazione tra processi
  - eventualmente distribuiti su macchine differenti
  - **basato sulla rete**

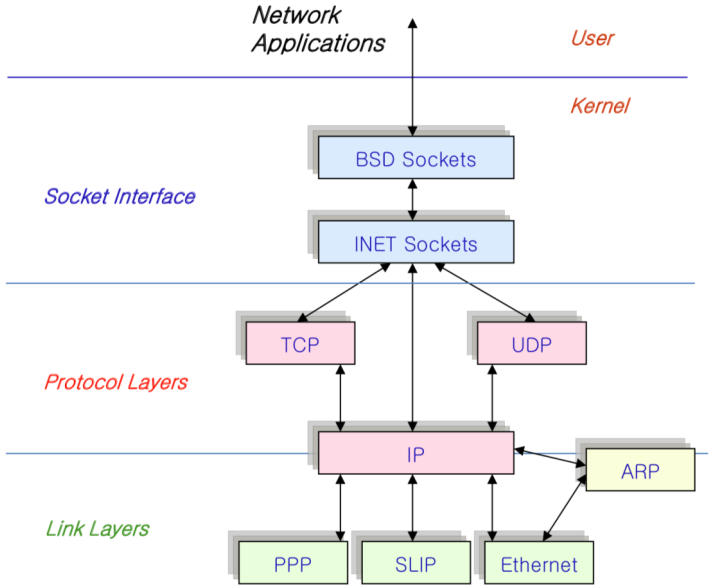
- Una socket fornisce ai processi un meccanismo per l'accesso alla rete



Socket

- È nata con il sistema operativo **Unix 4.2 BSD**, rilasciato nel **1983**
- Una socket è caratterizzata da **Indirizzo IP** e **Numero di porto** del nodo
- Possono essere di tipo **TCP** o **UDP**





Esempi di porte note

- 20/tcp **FTP** - Il file transfer protocol data
- 21/tcp **FTP** - Il file transfer protocol – control
- 22/tcp **SSH** - Secure login
- 23/tcp **Telnet** insecure text communications
- 25/tcp **SMTP** - Simple Mail Transfer Protocol (E-mail)
- 53/tcp **DNS** - Domain Name Server
- 53/udp **DNS** - Domain Name Server
- 69/udp **TFTP** Trivial File Transfer Protocol
- 80/tcp **HTTP** HyperText Transfer Protocol (WWW)

Socket in Python

- Per creare una socket si utilizza la funzione **socket.socket()** inclusa nel modulo socket

```
import socket
s = socket.socket(socket_family, socket_type)
```

  - **socket_family** è la tipologia di indirizzo con cui si intende lavorare:
    - `socket_family= socket.AF_INET` **#IPv4**
  - **socket_type** è la tipologia di socket che vogliamo creare:
    - `socket_type=socket.SOCK_STREAM` **#TCP**
    - `socket_type=socket.SOCK_DGRAM` **#UDP**

Principali funzioni socket

- **`socket():`** crea una nuova socket
- **`bind(address):`** binding della socket con *address*
  - `address` è una tupla `(IP, porto)`. Nota che se `porto` è nullo il SO fornisce il primo porto libero
- **`listen(backlog):`** si mette in ascolto di connessioni verso la socket (usata per TCP)
  - *backlog* specifica il numero massimo di connessioni ammesse in coda
- **`accept():`** accetta una connessione
  - segue *bind()* e *listen()*
  - restituisce la coppia *(conn, add)*
    - *conn* è una nuova socket utilizzabile per inviare/ricevere dati
    - *add* è l'indirizzo associato alla socket dall'altro capo della connessione
- **`connect(address):`** apre una connessione verso l'indirizzo `address`
  - `address` è una tupla `(IP, porto)`
- **`close():`** chiude la socket

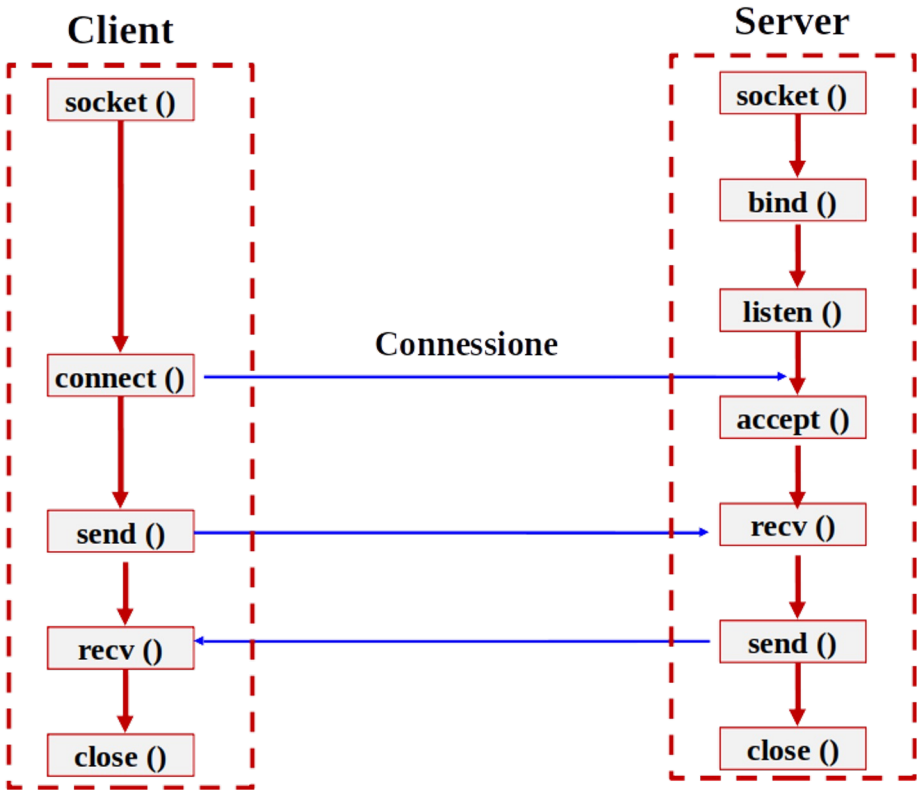
Scambio dati su socket

- **TCP**
  - **`send (string)`:** invia dati tramite una socket
    - la socket deve essere connessa ad una socket remota
  - **`recv (bufsize)`:** riceve i dati dalla socket.
    - il valore di ritorno è una stringa
    - *bufsize* è la massima quantità di dati che può essere ricevuta

Implementazione client-server TCP

Il processo **Client**, istanzia un Socket, e richiede una connessione

- L'indirizzo e la porta per la connessione è indicata nella *connect()*



Il processo **Server**, si pone in attesa di richieste di connessioni

- Le connessioni sono attese sulla porta definita nel metodo *bind()*
- *accept()* blocca il Server fino alla ricezione di una connessione

Esempio Server TCP

```
import socket

IP = 'localhost'
PORT = 0 # get first port available
BUFFER_SIZE = 1024

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((IP, PORT))
s.listen(1)

cur_port = s.getsockname()[1] # get used port

print("server on: ", IP, "port: ", cur_port)

conn, addr = s.accept()
print  ('Connection address: {}' .format(addr))
toClient= "The world never says hello back!\n"

data = conn.recv(BUFFER_SIZE)
print ("received data: " + data.decode("utf-8"))

conn.send(toClient.encode("utf-8"))

conn.close()
s.close()
```

Esempio Client TCP

```
import socket, sys

def client(PORT):

    IP = 'localhost'
    BUFFER_SIZE = 1024
    MESSAGE = "Hello, World!\n"

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((IP, PORT))
    s.send(MESSAGE.encode("utf-8"))

    data = s.recv(BUFFER_SIZE)
    print ("received data: " + data.decode("utf-8"))

    s.close()

if __name__ == "__main__":
    try:
        PORT = sys.argv[1]
    except IndexError:
        print("Please, specify PORT arg...")

    assert PORT != "", 'specify port'
    client(int(PORT))
```

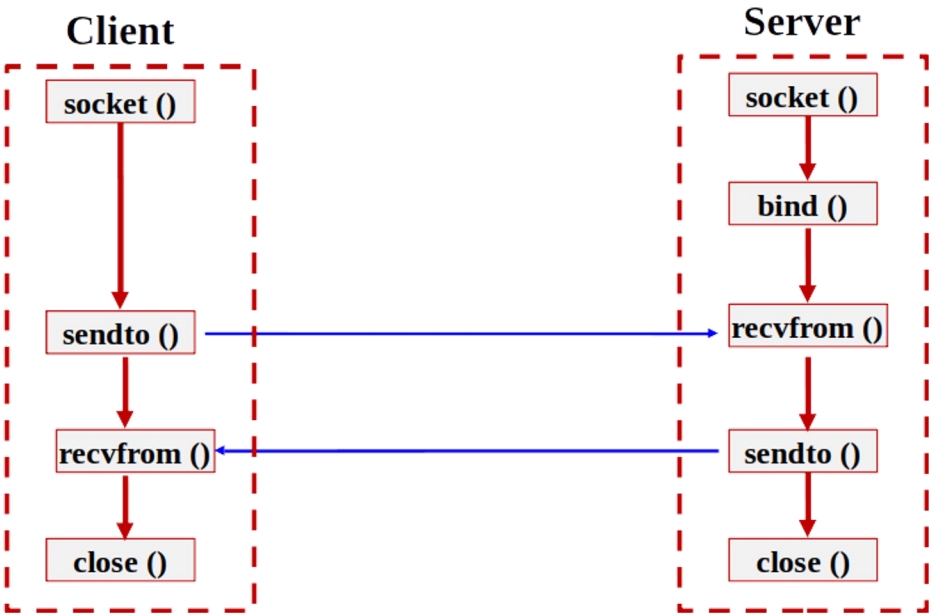
Scambio dati su socket

- **UDP**
  - **sendto (string, address):** invia dati alla socket
    - la socket non deve essere connessa ad una socket remota, dato che la destinazione è specificata da address
  - **recvfrom (bufsize):** riceve i dati dalla socket
    - il valore di ritorno è una coppia (string, address)
      - *string* è una stringa contenente i dati ricevuti
      - *address* è l'indirizzo della socket che invia i dati
    - *bufsize* è la massima quantità di dati che può essere ricevuta

Implementazione client-server UDP

Il processo **Client**, istanzia un Socket, ed invia un pacchetto

- L'indirizzo e la porta del destinatario sono specificati nella *sendto()*



UDP è *connectionless* quindi non è necessario lato **Server** mettersi in attesa di connessione

- Le informazioni del mittente sono ritornate dalla *recvfrom()*

Esempio Server UDP

```
import socket

msgServer = "Ciao client!"
serverAddressPort = ("localhost".encode("utf-8"), 7000)
bufferSize = 1024

s = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
s.bind(serverAddressPort)

msgClient, addr = s.recvfrom(bufferSize)
print("[Server]: Messaggio ricevuto: " + msgClient.decode("utf-8"))
print("[Server]: Indirizzo client: {}" .format(addr))

print ("[Server]: Invio dati al client")
s.sendto(msgServer.encode("utf-8"), addr)

s.close()
```

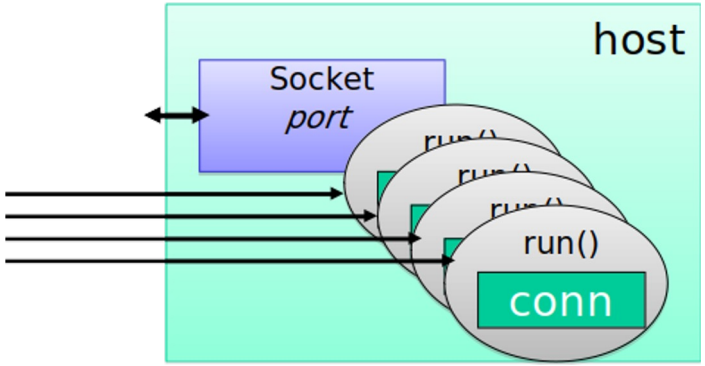
Esempio Client UDP

```
import socket, sys

def client(port):

    msgClient = "Ciao server!"
    serverAddressPort = ("localhost".encode("utf-8"), port)
    bufferSize = 1024

    s = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

    print ("[Client]: Invio dati al server. msg: ", msgClient)
    s.sendto(msgClient.encode("utf-8"), serverAddressPort)

    msgServer, addr = s.recvfrom(bufferSize)
    print("[Client]: Risposta server: " + msgServer.decode("utf-8"))

    s.close()

if __name__ == "__main__":
    try:
        port = sys.argv[1]
    except IndexError:
        print("Please, specify PORT arg...")

    assert port != "", 'specify port'
    client(int(port))
```

Server Multi-thread/Multi-process

Migliorano l'efficienza di un server (TCP/UDP) creando **più thread/process** per gestire le richieste:

- i Thread/Process devono prevedere un **costruttore** che accetti come parametro/i
  - **TCP:** una **connessione** (restituita dal metodo **`accept`**),
    - tutte le connessioni sono gestite da un thread differente
  - **UDP:** l'**indirizzo/porta** del mittente ed il **messaggio ricevuto** (restituito dal metodo **`recvfrom`**)
    - ogni richiesta è gestita da un thread differente
- nel metodo **run()** dei Thread/Process deve esserci tutto il codice per gestire una richiesta



Esempio Server Multi-thread: Server TCP

```
import socket
import threading

… # Thread function

if __name__ == '__main__':

    host = ""
    port = 12345

    # create and bind a TCP socket
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((host, port))
    print("Socket binded to port", port)

    s.listen(5)
    print("Socket is listening")

    while True:

        # establish a connection with client
        c, addr = s.accept()
        print('Connected to :', addr[0], ':', addr[1])

        # start a new thread
        t = threading.Thread(target=thd_fun, args=(c,))
        t.start()

    s.close()
```

```
…

# Thread function
def thd_fun(c):


    # data received from client
    data = c.recv(1024)

    # reverse the given string from client
    data = data[::-1]

    # send back reversed string to client
    c.send(data)

    # connection closed
    c.close()

…
```

Esempio Server Multi-process: Server TCP

```
import socket
import multiprocess as mp

… # Process function

if __name__ == '__main__':

    host = ""
    port = 12345

    # create and bind a TCP socket
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((host, port))
    print("Socket binded to port", port)

    s.listen(5)
    print("Socket is listening")

    while True:

        # establish a connection with client
        c, addr = s.accept()
        print('Connected to :', addr[0], ':', addr[1])

        # start a new process
        p = mp.Process (target=proc_fun, args=(c,))
        p.start()

    s.close()
```

```
…

# Process function
def proc_fun(c):


    # data received from client
    data = c.recv(1024)

    # reverse the given string from client
    data = data[::-1]

    # send back reversed string to client
    c.send(data)

    # connection closed
    c.close()

…
```

Applicazione client-server con socket

In un'applicazione con comunicazione basata su socket, client e server prevedono meccanismi di:

- connessione;
- *esternalizzazione* dei dati;
- invio-ricezione delle richieste;
- ricostruzione dei valori ricevuti

Potenziali rischi:

– l'implementazione dei meccanismi di comunicazione "distrae" dalla realizzazione delle funzionalità effettive dell'applicazione;

– sovrapposizione della logica i) applicativa con quella di ii) comunicazione client-server.

Applicazione client-server con socket

Il programmatore lato client dovrebbe concentrarsi sulla logica applicativa (invocando i servizi del server specificati da un'interfaccia), e separare la logica applicativa dai meccanismi di comunicazione/interazione con il server.

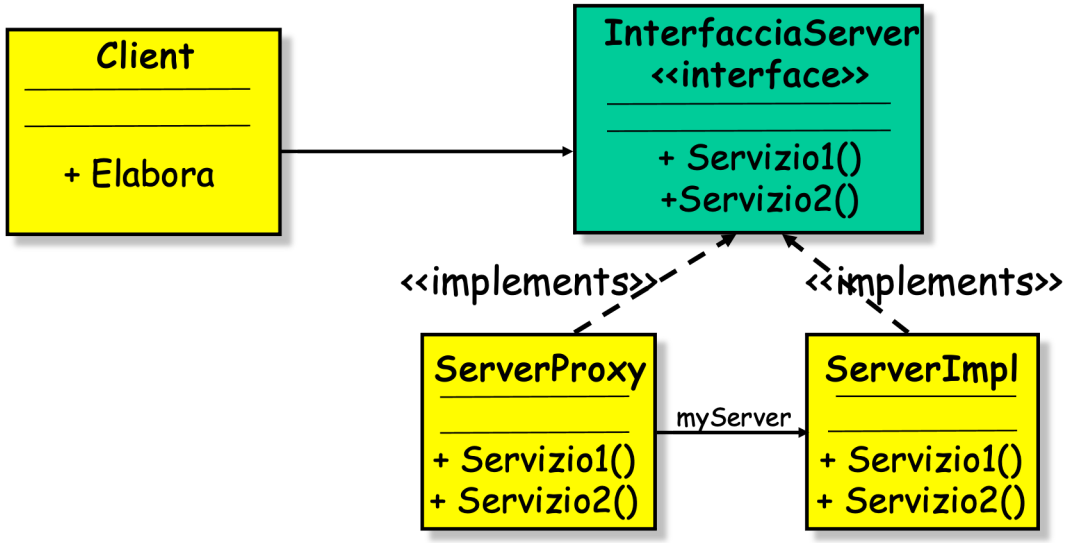
Analogamente, il programmatore lato server dovrebbe concentrarsi sulla codifica dei servizi offerti, separandola dai meccanismi di comunicazione/interazione con il client.

Il pattern Proxy

Il pattern introduce nel modello di interazione client-server un Proxy (detto anche "Surrogato" o "Ambassador"):

- il Proxy si presenta come una implementazione del servizio remoto, locale al client;
- i meccanismi di "*basso livello*" necessari per l'instaurazione della comunicazione, e per l'invio-ricezione dei parametri delle invocazioni tramite TCP o UDP, sono implementati ed incapsulati all'interno del Proxy.

Proxy-Skeleton: il proxy



Utilizzo del pattern Proxy

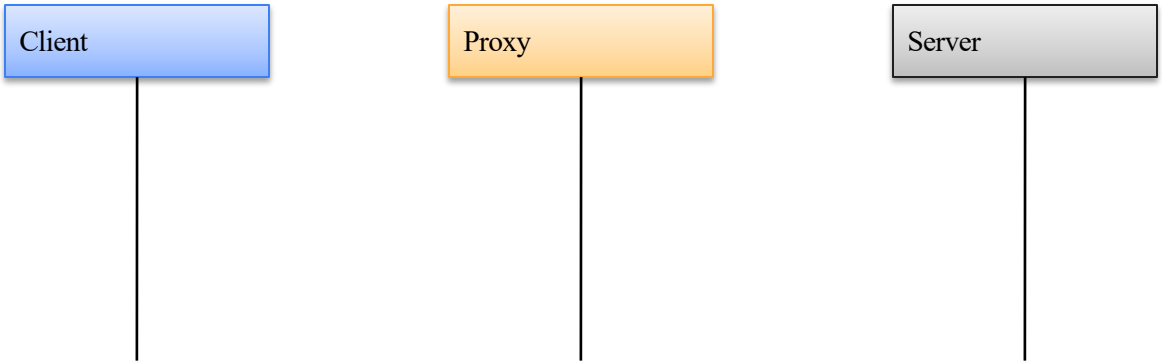
- **Il servizio fornito è specificato da InterfacciaServer.**

- **Duplice implementazione, ServerImpl e ServerProxy:**
  - ServerImpl implementa effettivamente i servizi dichiarati nell'interfaccia;
  - ServerProxy si fa carico dell'inoltro delle richieste (effettuate dal client) verso il lato server.

- **L'associazione direzionale tra ServerProxy e ServerImpl (indicata nella figura precedente) sintetizza tutti i meccanismi necessari al proxy per riferirsi al server.**

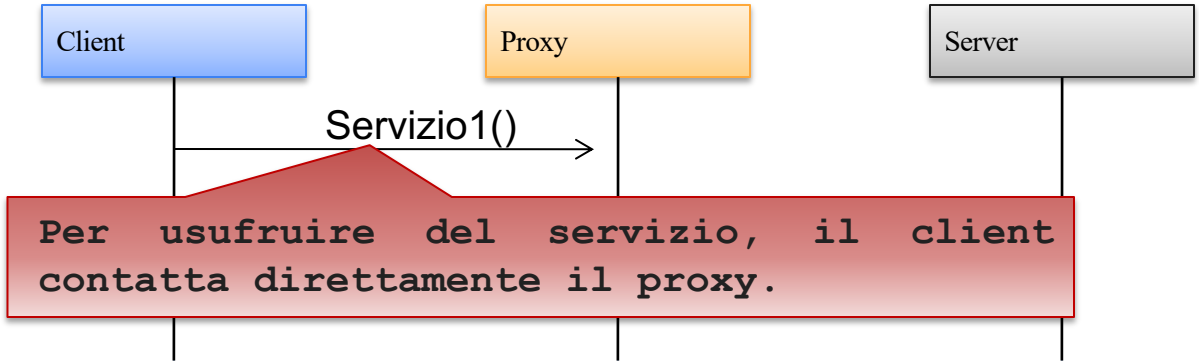
Utilizzo del pattern Proxy

- Il client possiede un riferimento ad un oggetto di tipo "InterfacciaServer” (ossia ServerProxy);
- Il client usa il proxy in sostituzione del servizio reale; il proxy si fa carico dell’interazione con il lato server.



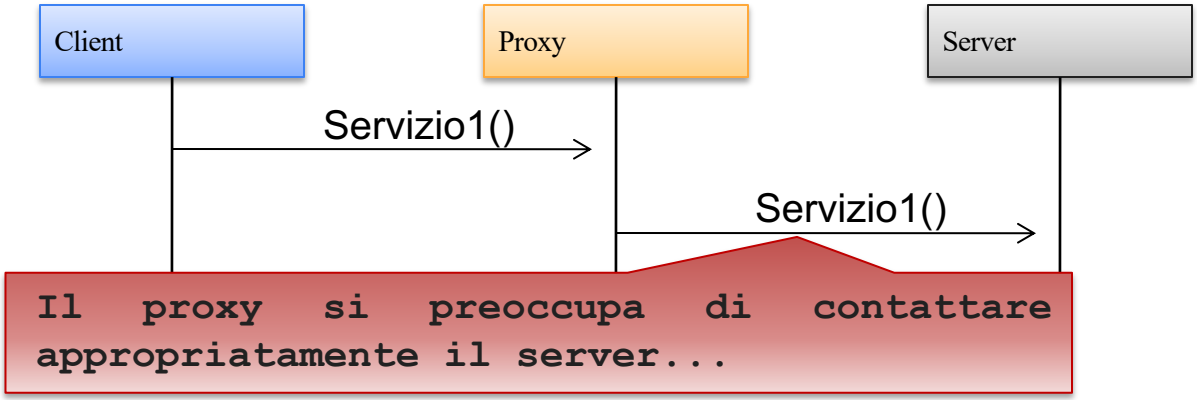
Utilizzo del pattern Proxy

- Il client possiede un riferimento ad un oggetto di tipo "InterfacciaServer” (ossia ServerProxy);
- Il client usa il proxy in sostituzione del servizio reale; il proxy si fa carico dell’interazione con il lato server.



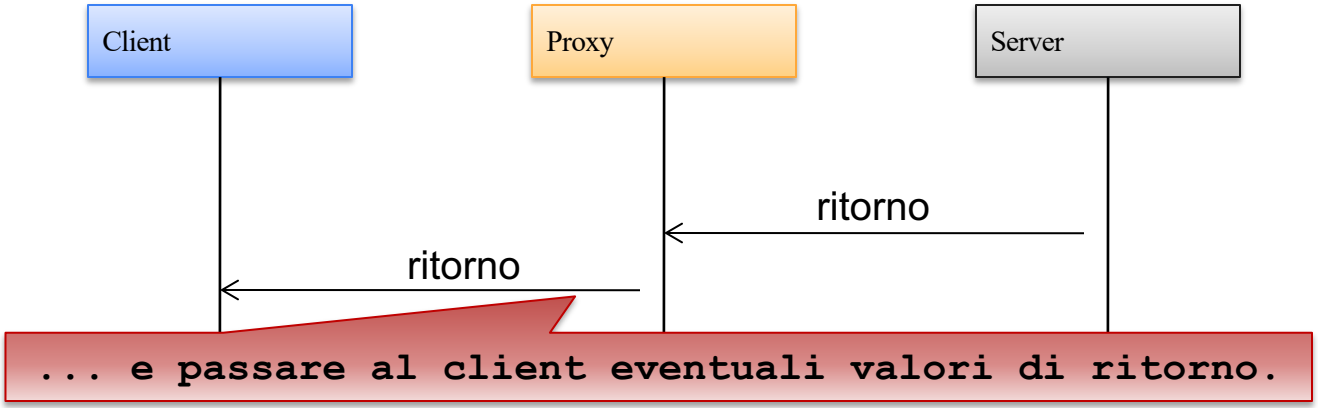
Utilizzo del pattern Proxy

- Il client possiede un riferimento ad un oggetto di tipo "InterfacciaServer” (ossia ServerProxy);
- Il client usa il proxy in sostituzione del servizio reale; il proxy si fa carico dell’interazione con il lato server.



Utilizzo del pattern Proxy

- Il client possiede un riferimento ad un oggetto di tipo "InterfacciaServer” (ossia ServerProxy);
- Il client usa il proxy in sostituzione del servizio reale; il proxy si fa carico dell’interazione con il lato server.



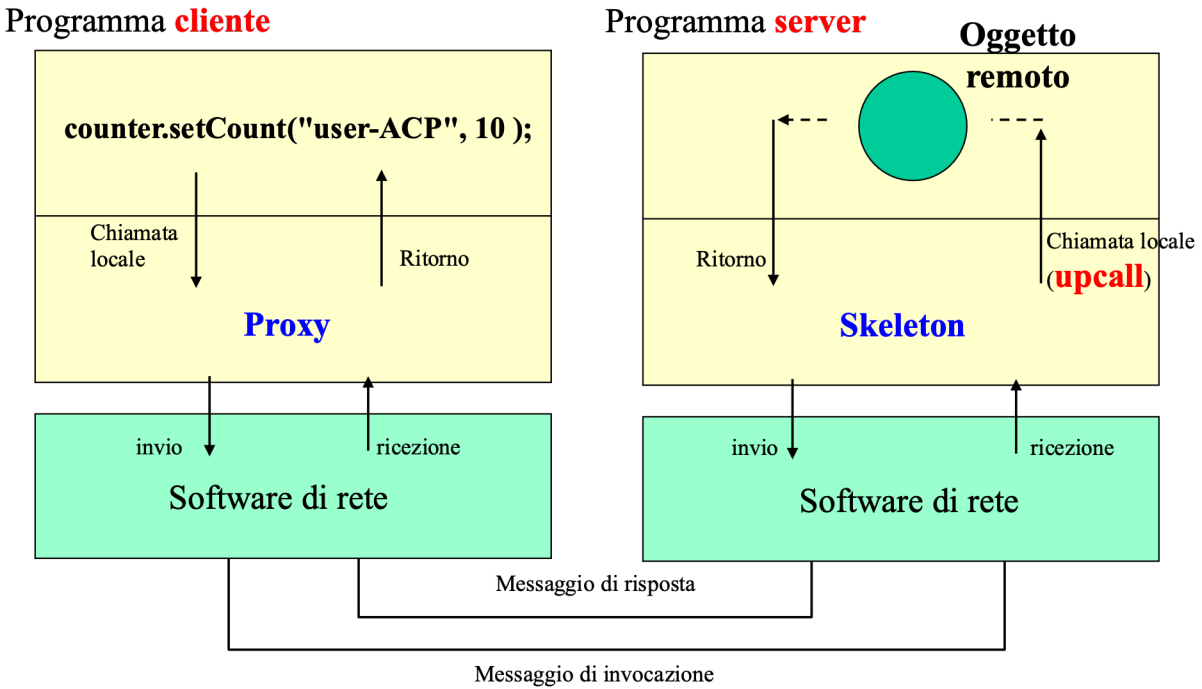
Il ruolo dello Skeleton (lato server)

Con l'aggiunta del Proxy il client è “sollevato” dalle problematiche di comunicazione.
In maniera analoga un proxy lato server, detto Skeleton, si fa carico della comunicazione con il Proxy lato client.

Lo skeleton:

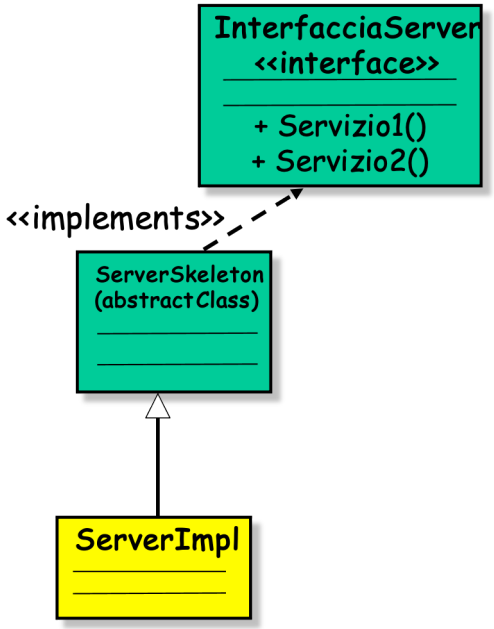
1. riceve le richieste di servizio
2. struttura l'informazione fornita in ingresso tramite socket TCP o UDP
3. effettua l'up-call al servizio reale
4. riceve da questi eventuali risultati
5. invia una risposta al proxy lato client.

Proxy-Skeleton: scherma concettuale



Proxy-Skeleton: lo skeleton

Per ereditarietà

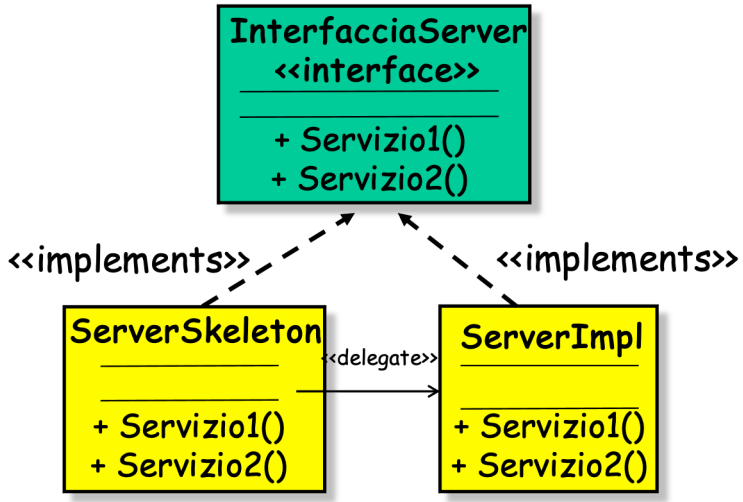


Lo skeleton può essere implementato per ereditarietà: la classe astratta Skeleton implementa solo gli opportuni schemi di comunicazione, ma lascia senza implementazione i metodi dell'interfaccia.

ServerImpl è una sottoclasse dello skeleton e fornisce implementazione ai metodi astratti.

Proxy-Skeleton: lo skeleton

Per delega



Lo skeleton può anche essere implementato per delega: la classe Skeleton presenta al suo interno un riferimento a InterfacciaServer (delegate), e i metodi sono così realizzati:

```
Servizio1() {
     delegate.Servizio1();

}
```

Proxy-skeleton in Python con interfacce formali


```
#interface.py
from abc import ABC, abstractmethod

class Subject(ABC):

        @abstractmethod
        def request(self, data):
                pass
```

Proxy in Python con interfacce formali

```
#client.py

import socket, sys
from interface import Subject

class Proxy(Subject):

    def __init__(self, port):
        self.port = port

    def request(self, message):
        IP = 'localhost'
        BUFFER_SIZE = 1024
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((IP, self.port))
        s.send(message.encode("utf-8"))
        data = s.recv(BUFFER_SIZE)
        print ("received data: " + data.decode("utf-8"))
        s.close()

...
```

```
...

if __name__ == "__main__":

    try:
        PORT = sys.argv[1]
        MESSAGE = sys.argv[2]
    except IndexError:
        print("Specify args!")

    print("Client: Generating request)
    proxy = Proxy(int(PORT))
    proxy.request(MESSAGE)
```

Skeleton (ereditarietà) in Python con interfacce formali

```
#server.py – skelethon per ereditarietà
import socket, sys, threading
from abc import ABC, abstractmethod
from interface import Subject
class Skeleton(Subject, ABC):

    def __init__(self, port):
        self.port = port

    @abstractmethod
    def request(self, data):
        pass

    def run_skeleton(self):
        host = 'localhost'
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.bind((host, self.port))
        self.port = s.getsockname()[1] # get used port
        s.listen(5)
        print("Socket is listening")
        while True:
            c, addr = s.accept()
            print('Connected to :', addr[0], ':', addr[1])
            t = threading.Thread(target=thd_fun, args=(c, self))
            t.start()
        s.close()
...
```

```
...

#thread function
def thd_fun(c, self):

    # data received from client
    data = c.recv(1024)

    # upcall
    result = self.request(data)

    # send reversed string to client
    c.send(result)

    # connection closed
    c.close()

...
```

Skeleton (ereditarietà) in Python con interfacce formali

```
...

class RealSubject(Skeleton):

    def request(self, data):

        # reverse the given string from client
        data = data[::-1]

        return data


if __name__ == "__main__":

    try:
        PORT = sys.argv[1]
    except IndexError:
        print("Please, specify PORT arg")

    print("Server running")

    realSubject = RealSubject(int(PORT))
    realSubject.run_skeleton()

...
```