Il modello Client/Server con Python

Layering

- ISO/OSI propone un'architettura di soluzione per arrivare a descrivere una comunicazione ICT complessa
  - L'architettura si basa sul principio dell'astrazione che richiede di nascondere dettagli e mostrare solo le entità utili significative per l'utente finale (cliente)
- Avendo un problema complesso, si introducono una serie di astrazioni, i livelli, che consentono di risolvere il problema separando gli ambiti in modo ordinato e ben identificato
- Si definiscono una serie di livelli per decomporre il problema e affrontare le complessità separatamente
- Il "divide et impera" è il principio ingegneristico 0 per la gestione di complessità ed eterogeneità ed è di fondamentale importanza per le reti di calcolatori

Vantaggi dell'approccio layered

- L'approccio layered permette a progettisti e sviluppatori che realizzano soluzioni di livello X di non curarsi dei protocolli di livello sottostante
  - È sufficiente rispettare le interfacce fornite dai protocolli di livello sottostante
- A ogni livello possiamo ragionare indipendentemente
  - È possibile impilare i protocolli come vogliamo, purché soddisfino i vincoli di interfacciamento
  - Questo consente di **sostituire, sviluppare, e aggiornare i singoli protocolli singolarmente senza modificare il resto dello stack**

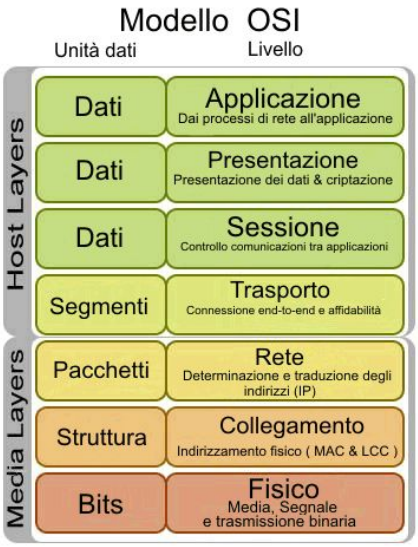
Il Modello ISO/OSI

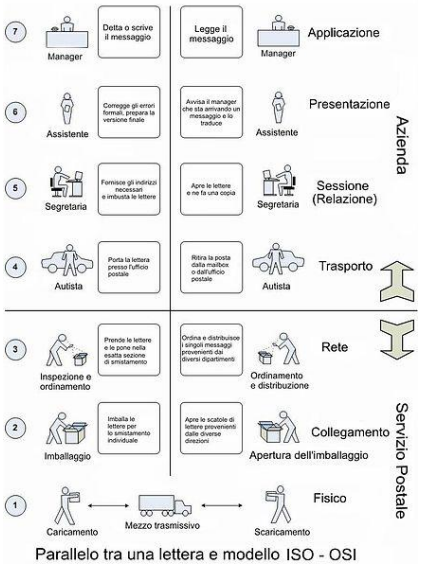
- Modello a 7 layer basato su IBM SNA
- Partendo dal livello applicazione (top-down), ogni livello ha l'obiettivo di comunicare con il pari, e lo realizza tramite un protocollo, ossia un insieme di passi, che realizza usando il servizio sottostante
- Ogni livello fornisce un servizio specificato e disponibile al livello superiore
- Ogni sistema a livelli si basa sul **principio della separazione dei compiti (delega) e della trasparenza della realizzazione (astrazione)**

Il Modello ISO/OSI

- In genere, per un'azione di invio informazioni, possiamo avere:
  - **Mittente**: entità che ha la responsabilità di iniziare la comunicazione
  - **Ricevente**: entità che accetta la comunicazione e poi la sostiene
  - **Intermediari**: eventuali nodi intermedi (access point, bridge, switch, router) che devono partecipare alla comunicazione - e fornire risorse per sostenerla
- Il mittente manda dei dati a un ricevente che può anche rispondere all'invio con un'azione applicativa conseguente
- Ogni azione comporta una comunicazione che passa attraverso i livelli da applicativo a fisico, del mittente e ricevente e almeno fino al livello di rete per gli intermediari

Il Modello ISO/OSI





Courtesy: Wikipedia

Socket Programming in Python

- Come C e Java, anche Python ci permette di realizzare applicazioni client/server utilizzando il modello delle BSD Socket
- Sono disponibili sia le socket STREAM (TCP) che le socket DATAGRAM (UDP) con un'interfaccia molto simile a quella del C, aggiungendo però la versatilità del linguaggio Python (API)
- Possiamo realizzare un server concorrente in Python utilizzando il modello modello multi-threading / multi-processing o modelli con pool di thread / processi.

Socket Programming in Python

Le funzionalità fondamentale per l'utilizzo delle socket sono disponibili nel modulo ***socket***.

Con il solo modulo socket è possibile realizzare client e server utilizzando il protocollo TCP o UDP.

Inoltre, python ci mette a disposizione anche un modulo *socketserver* che semplifica l'implementazione della parte server, utilizzando anche modelli *fork per request* (quello visto in C) e *thread per request*.

La documentazione per il modulo *socketserver* è accessibile al seguente link.

Modello Client/Server TCP

PROGETTO DEL **SERVER (1/2)**

- Creare una **serversocket** specificando come parametro la classe di indirizzi IP e il tipo di protocollo (TCP)
    - `ss = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
- Impostare l'opzione si SO_REUSEADDR per evitare problemi di binding a restart multipli
    - `ss.setsockopt(socket .SOL_SOCKET, socket.SO_REUSEADDR, 1)`
- Chiamare **bind((indirizzo, porta))** per legare la server socket a una tupla (indirizzo, porta)
    - `ss.bind((indirizzo, porta))`
- Per recuperare l'indirizzo è possibile utilizzare **socket.gethostname();** L'indirizzo **"0.0.0.0"** lega la socket a tutte le interfacce disponibili;

Modello Client/Server TCP

Vediamo ora come implementare un'applicazione client/server utilizzando le socket in Python.

Anche se è possibile realizzare il supporto a IPv4 e IPv6 ci concentremo sul solo IPv4. Vi è la possibilità di definire server che funzionano con entrambe le classi di indirizzi IP.

I nomi dei metodi del modulo socket sono molto simili a quelli di C. Python infatti utilizza le system call UNIX delle BSD socket.

Modello Client/Server TCP

PROGETTO DEL **SERVER (2/2)**

- Chiamare il metodo **listen(maxConn)** a cui va specificato un intero rappresentante la coda delle richieste di connessione `ss.listen(maxConn)`
- Implementare un *ciclo infinito* in cui andiamo a gestire le richieste di connessione: chiamata ad ***accept()*** e creazione di un eventuale Thread per la gestione della connessione:
  - `(client, address) = ss.accept()`
- `client` è l'oggetto socket con cui gestire la connessione con il client appena connesso

Esempio: Echo Application

- Vediamo ora come implementare un semplice Echo Server TCP sequenziale
- Il server Echo si deve mettere in ascolto su una porta specificata in input dall'utente
- Al ricevimento della richiesta di connessione, il Server deve stampare a video la stringa ricevuta e infine restiturla al client che la stamperà a sua volta a video
- Una volta restituta la stringa al client, il server deve chiudere la connessione

Modello Client/Server TCP

PROGETTO DEL **CLIENT**

- Creare una **socket** client specificando come parametro la classe di indirizzi IP e il tipo di protocollo (TCP)
  - `s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
- Chiamare **connect((indirizzo, porta))** per effettuare la connessione a determinato host in ascolto sulla porta specificata
  - `s.connect((indirizzo, porta))`
- Utilizzare la socket chiamando i metodi di **send** e **recv** per l'invio e la ricezione di **bytes**
  - `s.send(buffer)`
  - `data = s.recv(buffSize)`

Esempio: Server 1/2

```
import socket

porta = input("Inserire porta server (> 1024):\n")

porta = int(porta)

ss = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# faccio il bind della socket su hostname e porta

ss.bind((socket.gethostname(), porta))

# specifico la dimensione della coda

ss.listen(5)
```

Esempio: Server 2/2

```
while True:
    (client, address) = ss.accept()
    print("Client connected from: {}".format(address))
    echo_txt = client.recv(1024)
    print("C: {}".format(str(echo_txt)))
    client.send(echo_txt)
    client.close())
```


Esempio: Client 1/2

```
import socket
# lettura di parametri con input
indirizzo = input("Inserire l'indirizzo IP del
server:\n")
porta = input("Inserire la porta del server:\n")
porta = int(porta)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((indirizzo, porta))
```

Esempio: Client 2/2

```
msg = input("Inserire un messaggio di echo:\n")

s.send(msg.encode())

# wait for reply

reply = s.recv(1024)

print("S: {}".format(reply.decode()))

s.close()
```


Altri metodi utili

- **socket.sendall(bytes, [flags,])**

  invia dati sulla socket; sendall continua a inviare tutto il contenuto di *bytes* fino a che non è stato inviato tutto il buffer o in caso di errori (Gestione eccezioni). Ritorna None in caso di successo.

- **socket.sendfile(file, offset=0, count=None)**

  invia un *file* sulla socket fino alla lettura di EOF utilizzando la syscall ad alte prestazioni **os.sendfile()**; file è un oggetto di tipo file aperto in modalità binaria; *count* può essere utilizzato per specificare il numero di byte da inviare;

Esercizio

Sviluppare un'applicazione client/server di **Remote Copy Procedure** che consente a un client di copiare un file su un server di backup. Il Server deve mettersi in ascolto sulla porta 32400.

Il client deve chiedere in input all'utente il nome del file da copiare sul server, per poi trasmettere il nome al server. Al ricevimento del nome del file, il Server deve inviare un ack al client, il quale una volta ricevuto procede con l'invio del file.

Modello Client/Server UDP

PROGETTO DEL **SERVER**

- Creare una **udp socket** specificando come parametro la classe di indirizzi IP e il protocollo UDP
- Chiamare **bind((indirizzo, porta))** per legare la socket server a una coppia indirizzo, porta
- Implementare un *ciclo infinito* in cui andiamo a leggere datagram dalla socket utilizzando il metodo **recvfrom()**
- **recvfrom()** ci restituisce il payload del datagram ricevuto unitamente alle informazioni del client (source IP, PORT)

Modello Client/Server UDP

PROGETTO DEL **CLIENT**

- Creare una udp **socket** client specificando come parametro la classe di indirizzi IP e il tipo di protocollo (UDP)
  - `us = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`
- A differenza di TCP non dobbiamo stabilire una connessione con il server
- Inviare un datagram sulla socket utilizzando il metodo **sendto**(msg, address)
- msg è un oggetto di tipo byte che vogliamo inviare sulla socket, mentre address è la coppia IP:PORT, ad esempio, (‘127.0.0.1’, 2300)

Esempio: Echo Application UDP version

- Come cambia il modello UDP? Quali sono le differenze fondamentali?
- Proviamo a implementare lo stesso esempio ma utilizzando il protocollo UDP e non TCP.
- Non verrà stabilita una “connessione” tra client e server perchè UDP è connectionless.
- Vediamo come cambia l’implementazione UDP…

Esempio: Server 1/2

```
import socket

porta = input("Inserire porta UDP (> 1024):\n")

porta = int(porta)

udp_ss = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)


# faccio il bind della socket su hostname e porta

udp_ss.bind((socket.gethostname(), porta))
```


Esempio: Server 2/2

```
# listen for UDP datagrams

while True:

    (msg, address) = udp_ss.recvfrom(1024)

    print("Received UDP Datagram from {}".format(address))

    print("C: {}".format(msg.decode()))

    udp_ss.sendto(msg, address) # devo specificare anche
l'indirizzo, UDP è connectionless
```

Esempio: Client 1/2

```
import socket

indirizzo = input("IP server:\n")

porta = input("Porta del server:\n")

porta = int(porta)

us = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

msg = input("Inserire un messaggio di echo:\n")
```


Esempio: Client 2/2

```
#send a message

us.sendto(msg.encode(),(indirizzo, porta))

# wait for reply

(reply, _) = us.recvfrom(1024)

print("S: {}".format(reply.decode()))

us.close()
```

Esercizio: Date Server

Sviluppare un'applicazione client/server (utilizzando le socket UDP) per ottenere da un Server in ascolto su una determinata PORTA la data e il tempo corrente. A tal fine, il Server deve rispondere alle richieste dei client che richiedono il tempo configurato sul sistema.

Per richiedere il servizio, il client invia un datagram contenente la stringa "date" al Server, che una volta ricevuto il messaggio esegue il comando **date**, ne legge l'output e lo invia al client che ha fatto richiesta.

Dual-stack IPv6

Con poche semplici istruzioni è possibile creare un server dual-stack, con supporto sia a IPv4 che IPv6.

```
import socket

addr = ("", 45000)

if socket.has_dualstack_ipv6():
    s =
socket.create_server(addr, family=socket.AF_INET6, dualstack_ipv6=True)
else:
    s = socket.create_server(addr)

s.listen(1)
```

Notare la differenza di sintassi con il codice precedente…

Multi-threading in Python

Python supporta il modello multi-threading, tuttavia come altri linguaggi interpretati (Ruby), la versione CPython soffre del problema del **Global Interpreter Lock (GIL)**. Il GIL può essere visto come un mutex che implementa un accesso in mutua esclusione all'interprete Python → *assenza di un effettivo parallelismo, anche in caso di applicazione multithreaded.*

Tuttavia il GIL di Python è fondamentale per garantire la safety di un'applicazione quando per esempio si va a modificare la reference a un oggetto.

Il problema della concorrenza verrà affrontato in corsi più avanzati.

Una soluzione parziale al problema del GIL è l'utilizzo del modello **multi-processing.**

Multi-threading in Python

Con multi-processing è possibile creare più sub-processi, ognuno con un proprio spazio di memoria, annullando così il problema del GIL.

```
from multiprocessing import Pool
import time
COUNT = 50000000
def countdown(n):
    while n>0:
        n -= 1

if __name__ == '__main__':
    pool = Pool(processes=4)
    r1 = pool.apply_async(countdown, [COUNT//4])
    r2 = pool.apply_async(countdown, [COUNT//4])
    r3 = pool.apply_async(countdown, [COUNT//4])
    r4 = pool.apply_async(countdown, [COUNT//4])
    pool.close()
    pool.join()
```

Server multi-processing in Python

Possiamo utilizzare il metodo visto nell'esempio per implementare un server multi-processo.

Il codice della gestione della richiesta andrà implementato **in una funzione di handling**.

Al ricevimento di una richiesta di connessione andiamo a creare un nuovo processo (*metodo **start** su un'istanza di **Process***) passando la socket client creata alla funzione di handling.

Vediamo il nostro EchoServer implementato con il modello multi-processing…

Sistemi Publish and Subscribe

RabbitMQ