Python Standard Library

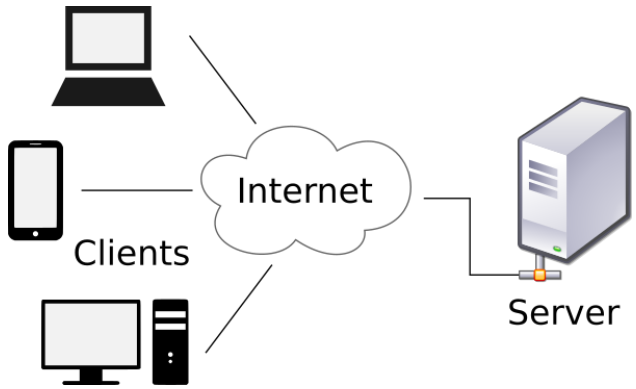
Socket e thread

Cos'è un socket

- Un *socket*, nel dominio delle reti, è un oggetto che rappresenta un *endpoint* di comunicazione.
- Esso espone un'interfaccia di programmazione a livello applicativo che permette di sfruttare i meccanismi di comunicazione sottostanti.
- Comunemente, i *socket* sfruttano i *layer* TCP o UDP per inviare e ricevere dati.

Sistema Server/Client

- L'idea alla base del sistema server/client è che una macchina (il server) mette a disposizione un servizio per altre macchine sulla rete (i client) che ne usufruiscono.
- La comunicazione è iniziata dal client, il quale invia la richiesta alla macchina server, che la processa e successivamente risponde al client.

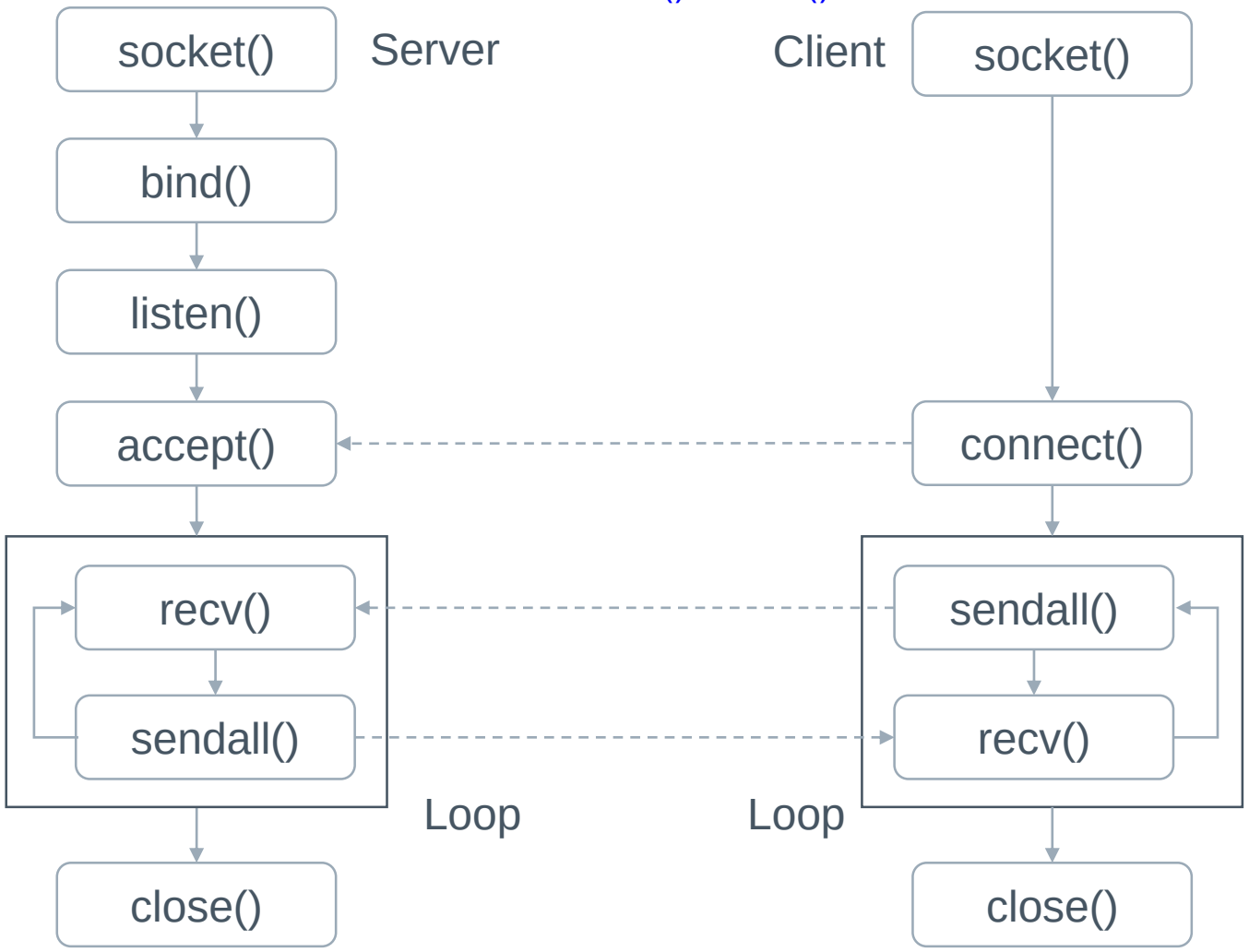


Esercizio: echo server TCP

- Scrivere un echo server con relativo client sfruttando i socket su protocollo TCP.
- Un echo server è un server che riceve dati da un client e li restituisce identici.
- Il server deve accettare la connessione dal client, attendere che il client invii i dati, rispondere, ed infine terminare.
- Il client invece, una volta connesso al server, invia i dati, attende la risposta, e poi termina.

Schema generale TCP

send() e recv() sono della connessione



Nota

- In realtà il nostro server non appena risponde al client esce immediatamente.
- Non è dunque necessario implementare il loop `recv()` / `sendall()` come lo schema generale suggerirebbe.

Modulo `socket`

https://docs.python.org/3/library/socket.html

In molte API di sistema operativo, l'interfaccia di programmazione per quanto riguarda i socket è quella dei socket UNIX.

Il modulo `socket` non fa altro che esporre la stessa interfaccia, tradotta per le strutture dati di python.

socket.socket

`socket.socket`(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

Crea un socket. La sua creazione non causa uso di risorse di comunicazione.

family indica la famiglia di indirizzi che possono essere associati al socket. I valori più comuni sono `AF_INET` per IPv4 e `AF_INET6` per IPv6.

type specifica il tipo di socket. I valori più comuni sono `SOCK_STREAM` per creare socket a connessione e `SOCK_DGRAM` per creare socket senza connessioni.

proto e *fileno* (quest'ultimo presente solo in python 3) sono parametri che servono per usi avanzati dei socket.

socket.socket

```
>>> import socket as sck
```

› Crea un socket TCP/IPv4

```
>>> s = sck.socket(sck.AF_INET, sck.SOCK_STREAM)
```

› Crea un socket UDP/IPv4

```
>>> s = sck.socket(sck.AF_INET, sck.SOCK_DGRAM)
```

› Crea un socket TCP/IPv6

```
>>> s = sck.socket(sck.AF_INET6, sck.SOCK_STREAM)
```

› Crea un socket UDP/IPv6

```
>>> s = sck.socket(sck.AF_INET6, sck.SOCK_DGRAM)
```

socket.bind

`socket.`**`bind`**(*address*)

Associa l'oggetto socket appena creato all'indirizzo *address*. Il socket non deve essere già stato associato ad altri indirizzi.

Il formato di *address* dipende dalla famiglia di indirizzi, per `AF_INET` o `AF_INET6` è una tupla (*hostname*, *port*).

```
...
>>> s.bind(('localhost', 80))
```

socket.listen

socket.**listen**([*backlog*])

SOCK_STREAM

Abilita il socket a ricevere connessioni. *backlog* specifica quanti tentativi di connessione falliti tollerare prima di rifiutare nuove connessioni da uno specifico client. Se assente, viene scelto un valore automaticamente.

```
...
>>> s.listen()
```

socket.accept

`socket.accept()`

`SOCK_STREAM`

Mette il socket in attesa di nuove connessioni.

conn = connessione

Appena la connessione è stabilita, restituisce una tupla (*conn*, *address*) dove *conn* è un oggetto socket da utilizzare per la comunicazione con il client, e *address* è il suo indirizzo.

```
...
>>> conn, addr = s.accept()
```

socket.connect

socket.**connect**(*address*)

SOCK_STREAM

Tenta di connettersi all'indirizzo specificato.

Il formato di *address* dipende dalla famiglia di indirizzi, per AF_INET o AF_INET6 è una tupla (*hostname*, *port*).

```
...
>>> s.connect(('localhost', 80))
```

socket.sendall

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

SOCK_STREAM

Invia dati all'indirizzo cui il socket è connesso. *bytes* è una stringa binaria che contiene i dati da inviare.

flags sono delle flag platform-dependent che modificano il comportamento della funzione.

```
...
>>> s.sendall(b'Hello...')
>>> s.sendall('...world!'.encode())
```

! In Python 2 non è necessario codificare la stringa in quanto viene fatto automaticamente.

socket.sendto

socket.**sendto**(*bytes*, *address*)

socket.**sendto**(*bytes*, *flags, address*)

SOCK_DGRAM

Invia dati all'indirizzo specificato. *bytes* è una stringa binaria che contiene i dati da inviare, mentre *address*, che dipende dalla famiglia degli indirizzi, è l'indirizzo di destinazione.

flags sono delle flag platform-dependent che modificano il comportamento della funzione.

```
...
>>> s.sendto(b'foo', ('localhost', 80))
```

socket.recv

socket.**recv**(*bufsize*[, *flags*])

SOCK_STREAM

Riceve dati dal socket connesso, restituendo una stringa binaria. *bufsize* specifica la dimensione del buffer di ricezione. Valori consigliati di *bufsize* sono potenze di 2 relativamente basse, come 4096.

flags sono delle flag platform-dependent che modificano il comportamento della funzione.

```
...
>>> data = s.recv(4096)
```

socket.recvfrom

socket.**recvfrom**(*bufsize*[, *flags*])

SOCK_DGRAM

Riceve dati dal socket, restituendo una tupla (*bytes*, *address*), ove *bytes* è una stringa binaria e *address* è l'indirizzo del client da cui provengono i dati. *bufsize* specifica la dimensione del buffer di ricezione. Valori consigliati di *bufsize* sono potenze di 2 relativamente basse, come 4096.

flags sono delle flag platform-dependent che modificano il comportamento della funzione.

```
...
>>> data, addr = s.recvfrom(4096)
```

socket.close

`socket.close()`

Chiude un socket e ne rilascia le risorse allocate dal sistema operativo.

```
s = sck.socket(...)
# do stuff
s.close()
```

si può invocare anche sulla "connection"
è meglio chiudere anche la "connection"

In alternativa a chiamare `close()` esplicitamente, è possibile usare `with`:

```
with sck.socket(...) as s:
    # do stuff
# close() è chiamato automaticamente
```

Note finali

- I metodi di invio e ricezione dati del socket operano di norma con stringhe binarie (bytes).
- Dichiarare una stringa binaria:
  - `ss = "Hello, world!"`
  - `sb = b"Hello, world!"`
- Metodi di conversione:
  - `ss.encode()`, da stringa (`ss`) a stringa binaria
  - `sb.decode()`, da stringa binaria (`sb`) a stringa

! In Python 3 la distinzione tra stringa e stringa binaria è più marcata rispetto a Python 2, quindi bisogna fare particolare attenzione alle conversioni.

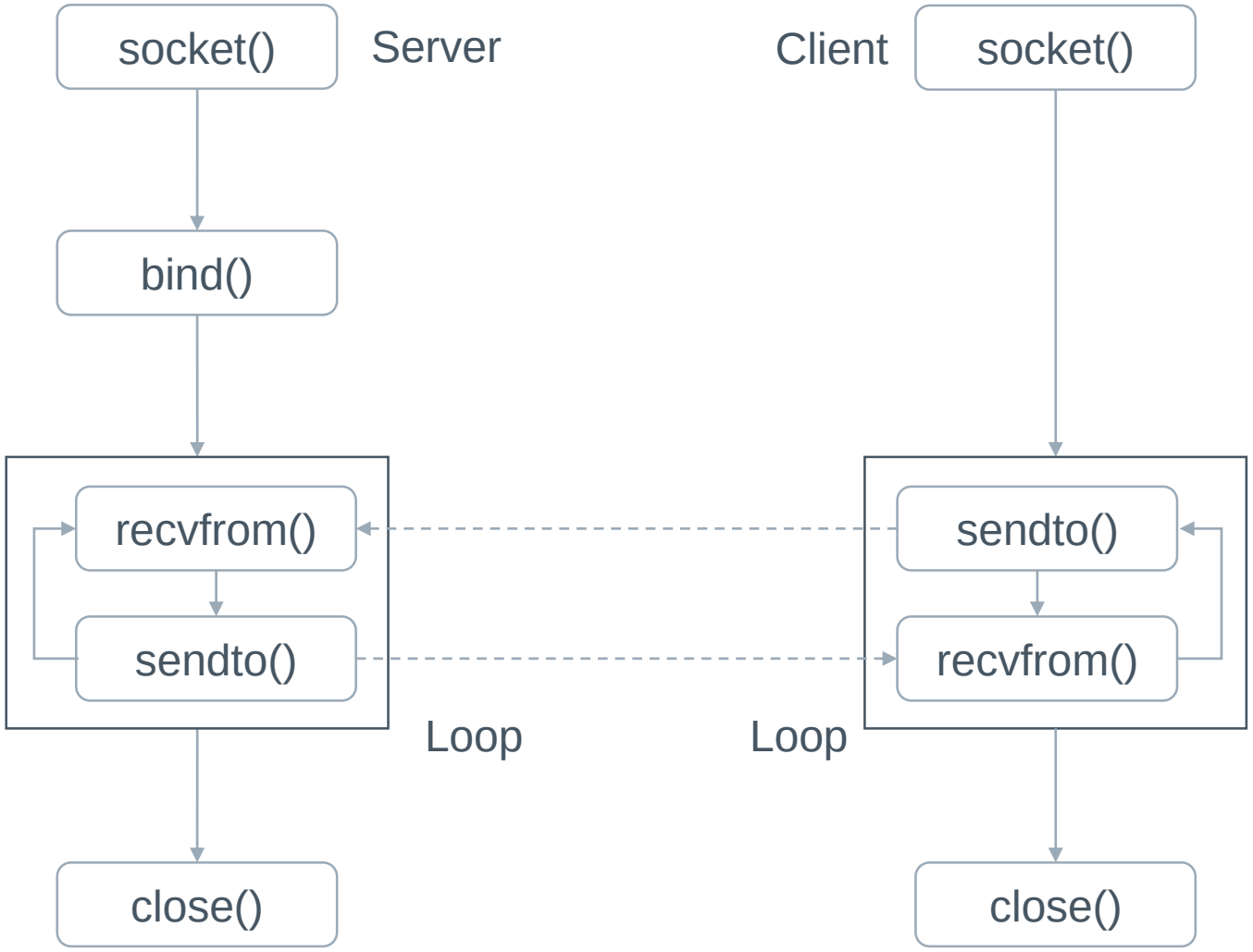
Note finali

- `socket.setsockopt()` è un metodo che permette di modificare il comportamento del socket a basso livello.
- In particolare, la chiamata `s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)` prima di `bind()` costringe il sistema operativo ad usare l'indirizzo passato a `bind()` anche se esso è già occupato.
- Normalmente non sempre è una buona idea utilizzare `SO_REUSEADDR`, in quanto in alcune piattaforme non è specificato cosa accade ai socket già attivi su quell'indirizzo.

Esercizio: echo server UDP

- Convertire l'echo server TCP scritto in precedenza per TCP in un echo server UDP.
- Questo comprende la modifica sia dello script server che dello script client.

Schema generale UDP



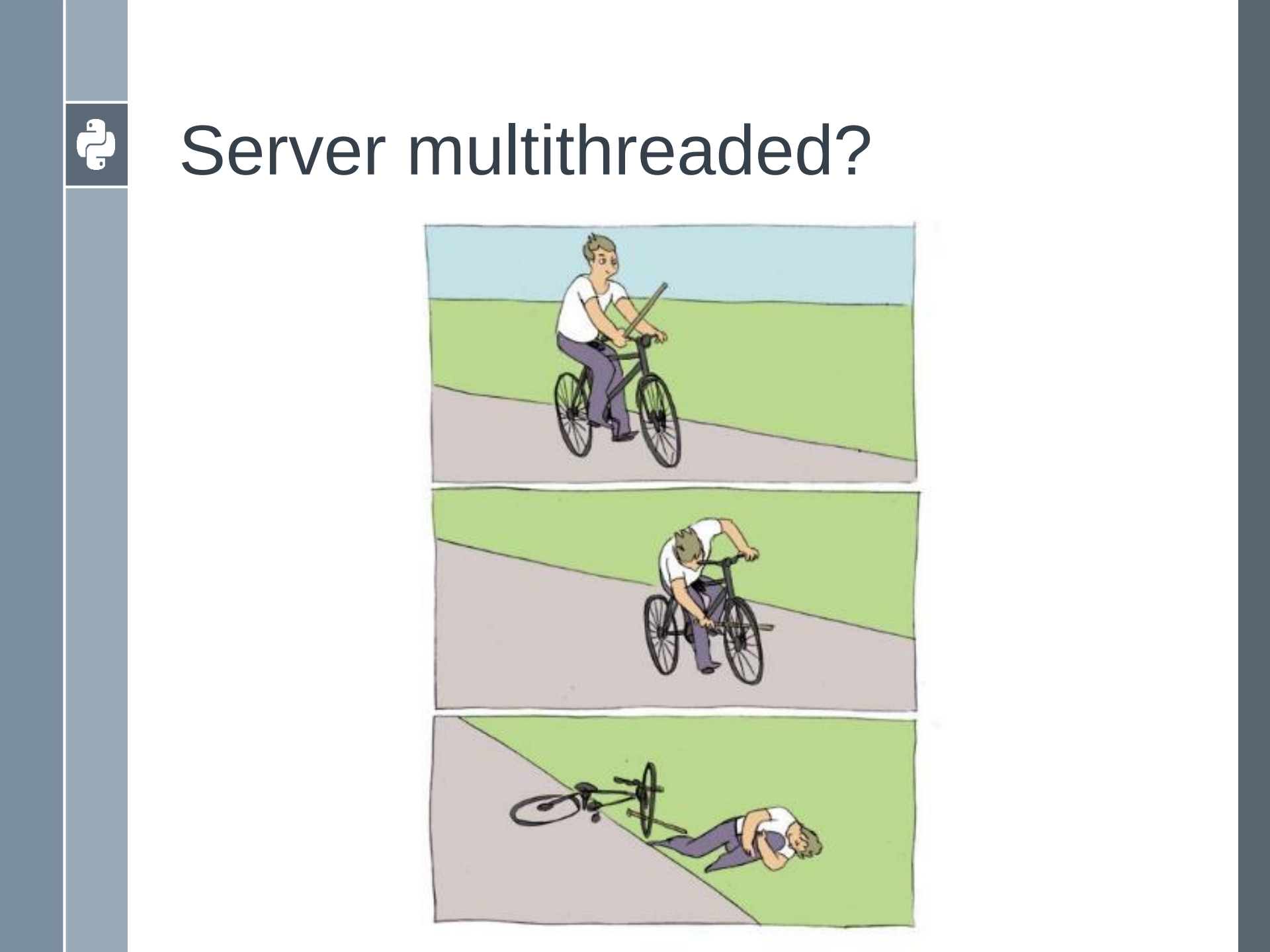
Riferimenti

`socket.`**`recvfrom`**(*bufsize*[, *flags*])

Riceve dati dal socket, restituendo una tupla (*bytes*, *address*), ove *bytes* è una stringa binaria e *address* è l'indirizzo del client da cui provengono i dati.

`socket.`**`sendto`**(*bytes*, *address*)

Invia dati all'indirizzo specificato. *bytes* è una stringa binaria che contiene i dati da inviare, mentre *address*, è l'indirizzo di destinazione.



Server multithreaded?

Modulo `threading`

https://docs.python.org/3/library/threading.html

Contiene un'interfaccia di alto livello per la creazione e la gestione di thread.

`threading.Thread`

`threading.`**`Thread`**(*group*=None, *target*=None, *name*=None, *args*=*(), kwargs*={})

Crea un nuovo thread che, quando avviato, invocherà la funzione *target*, passandogli come parametri *args* (una sequenza) e/o *kwargs* (un dizionario).

group è attualmente ignorato dal costruttore e riservato ad usi futuri.

name, se specificato, assegna un nome al thread, che altrimenti viene generato automaticamente.

Questo costruttore andrebbe chiamato *specificando sempre i nomi dei parametri*.

thread.start

`thread.start()`

Avvia il thread precedentemente creato. Può essere chiamato solo una volta.

```
import threading as thr
def say(something):
    print("I am saying ", something)
t = thr.Thread(target=say, \
    args=('qualcosa',))
t.start()
```

`threading.Thread`

- Python mette a disposizione anche un meccanismo object oriented per la creazione di un nuovo thread.
- Per utilizzarlo, bisogna subclassare `threading.Thread` e fare l'override del metodo `run()`.
- Il metodo `run()` è quello che determinerà cosa deve fare il thread appena creato.
- Non bisogna mai invocare `run()` in modo esplicito: viene utilizzato da `start()` internamente.

Esempio

```
import threading as thr
class TalkingThread(thr.Thread):
    def __init__(self, something):
        thr.Thread.__init__(self)
        self.something = something
    def run(self):
        print("I am saying ", \
            self.something)
t = TalkingThread('qualcosa')
t.start()
```

thread.join

`thread.`**`join`**(*timeout=None*)

Aspetta che il thread termini, o finchè non trascorrono *timeout* secondi. *timeout* può essere un valore in virgola mobile oppure `None`: in tal caso l'attesa è indefinita.

```
t = threading.Thread(...)
t.start()
# do something while t is running
t.join()
```

Miscellanea

`thread.is_alive()`

Restituisce `True` se il thread è ancora attivo.

`thread.name`

Contiene il nome del thread. Può essere utilizzato anche per cambiare il nome attuale.

`threading.current_thread()`

Chiamata in qualsiasi punto del programma restituisce l'oggetto `thread` che sta eseguendo quel frammento di codice in quel momento.

threading.Lock

`threading.Lock`()

Crea un nuovo lock.

`lock.acquire`(*blocking=True*, *timeout=-1*)

Tenta di acquisire il possesso del lock. Restituisce `True` se l'operazione ha avuto successo, `False` altrimenti.

Se *blocking* è `True`, si ferma inoltre per *timeout* secondi al massimo in attesa che il lock si liberi. Se *timeout* è -1 attende indefinitamente.

! *timeout* non è disponibile in Python 2

`lock.release`()

Rilascia il lock.

threading.Condition

`threading.`**`Condition`**(*lock=None*)

Crea una condition variable, utilizzando *lock* come lock associato. Se *lock* è `None`, viene creato un lock internamente.

`condition.`**`acquire`**(**args*)

Tenta di acquisire il possesso del lock associato. Stessa semantica e parametri di `lock.acquire()`.

`condition.`**`release`**()

Rilascia il lock associato. Stessa semantica di `lock.release()`.

threading.Condition

`condition.`**`wait`**(*timeout=None*)

Attende finché qualche altro thread non chiama `notify()` o `notify_all()`, o finché non trascorrano *timeout* secondi. Se *timeout* è `None`, attende indefinitamente.

Questo metodo rilascia il lock associato alla condition variable quando viene chiamato, e tenta di riacquisirlo quando viene risvegliato da un `notify()`.

`condition.`**`notify`**(*n=1*)

Risveglia fino a *n* thread in attesa tramite `wait()`.

`condition.`**`notify_all`**()

Risveglia tutti i thread in attesa tramite `wait()`.

threading.Timer

`threading.`**`Timer`**(*interval*, *function*, *args=[]*, *kwargs={}*)

Crea un timer, ossia una sottoclasse di `Thread` che esegue *function* dopo *interval* secondi. *args* e *kwargs* hanno lo stesso significato che hanno nel costruttore di `Thread`.

`timer.`**`cancel`**()

Interrompe il timer e annulla l'esecuzione programmata di *function*.

```
def spoiler():
    print("Noooooooooooooooooooo")
Timer(3, spoiler).start()
```

time.sleep

Benchè non faccia parte modulo `threading`, è spesso usata insieme ad esso.

`time.`**`sleep`**(*secs*)

Sospende il thread corrente per *secs* secondi. *secs* può essere un numero in virgola mobile, e può essere minore di `1.0`.

Il sistema operativo può decidere di risvegliare il thread prima o dopo il timeout specificato, dipendentemente dalla situazione (e dalla versione di python installata).

Esercizio: produttori e consumatori

- Su di una lista condivisa, più thread concorrono per effettuare delle operazioni:
  - 1 thread tenta di appendere alla lista un numero generato casualmente (produttore).
  - 3 thread tentano di rimuovere l'ultimo numero dalla lista per stamparlo a schermo (consumatori).
- La lista non è limitata superiormente, quindi il produttore potrà sempre produrre. Al contrario, i consumatori possono consumare solo se la lista non è vuota.
- Scrivere in python un programma che simuli il problema.

Modulo `socketserver`

https://docs.python.org/3/library/socketserver.html

Contiene delle classi che semplificano la scrittura di server di rete.

! In python 2 il modulo si chiama `SocketServer` (con le maiuscole).

Server preconfigurati

- Nel modulo vi sono diverse classi che generano server preconfigurati:
  - `TCPServer`, che implementa un server TCP,
  - `ThreadingTCPServer`, che implementa un server TCP multithreaded,
  - `UDPServer`, che implementa un server UDP,
  - `ThreadingUDPServer`, che implementa un server UDP multithreaded.
- Tutte queste classi hanno la stessa interfaccia, che è quella di `BaseServer`.

socketserver.BaseServer

socketserver.**BaseServer**(*server_address*, *RequestHandlerClass*)

Costruisce un oggetto server.

server_address è l'indirizzo a cui fare il binding del socket utilizzato internamente. Nel caso di TCP o UDP è una tupla `(host, port)`.

RequestHandlerClass è il nome della classe che gestirà la richiesta in arrivo dal client.

Questa classe non va istanziata direttamente, ma attraverso una delle sottoclassi presenti nel modulo (`TCPServer`, `ThreadingUDPServer`, etc).

Avvio e interruzione

`baseserver.`**`serve_forever`**(*poll_interval=0.5*)

Gestisce le richieste finchè il programma non termina o finchè 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.

`baseserver.`**`shutdown`**()

Interrompe l'esecuzione del server.

allow_reuse_address

- Questo attributo *di classe* può essere cambiato nelle sottoclassi.
- Si comporta esattamente come se avessimo chiamato `setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)` sul socket usato internamente.

```
import socketserver as ss

class EchoServer(ss.TCPServer):
    allow_reuse_address = True
    # ...
```

socketserver.BaseRequestHandler

socketserver.**BaseRequestHandler**()

È la classe che definisce come il server risponde ad una richiesta da parte di un client.

Viene istanziata automaticamente dal server quando riceve una richiesta.

Per definire il comportamento di un server, è necessario sottoclassarla e fare l'override del metodo `handle()`.

La sottoclasse con `handle()` modificato dovrà essere passata al costruttore dell'oggetto server.

`socketserver.BaseRequestHandler`

`baserequesthandler.server`

Contiene il riferimento all'oggetto server che ha creato quest'istanza di `BaseRequestHandler`.

`baserequesthandler.client_address`

Contiene l'indirizzo del client.

`baserequesthandler.request`

Per i server TCP è un oggetto socket che può essere usato per la comunicazione con il client.

Per i server UDP è una tupla `(data, socket)` ove *data* sono i dati ricevuti, e *socket* è il socket che può essere usato per la comunicazione con il client.

Non è necessario chiudere esplicitamente i socket.

Esempio

```
import socketserver as ss

class EchoHandlr(ss.BaseRequestHandler):
  def handle(self):
    data, sock = self.request
    sock.sendto(data, \
      self.client_address)

ss.UDPServer(('localhost', 12345), \
    EchoHandlr).serve_forever()
```

Esercizio: fileserver

- Un fileserver è un server che consente ai client di scaricare file.
- Tramite appositi comandi, il client deve:
  - Ottenere la lista dei file scaricabili
  - Scaricare un file
  - Uscire e terminare la connessione
- Il server deve essere multithreaded e deve permettere ai client di scaricare solo i file della directory in cui risiede (non delle sottocartelle).
- Scrivere in python sia server che client.

Funzioni di filesystem

`os.listdir`(*path*)

Restituisce la lista di file e cartelle presenti in *path*. La lista non è ordinata. Non include il contenuto delle eventuali sottocartelle (non è ricorsiva).

`os.path.isfile`(*path*)

Restituisce `True` se *path* è un file, false altrimenti. Segue automaticamente i link simbolici.