# Programska potpora komunikacijskim sustavima

Dr. sc. Adrian Satja Kurdija


## Programski jezik Python - 6. predavanje

Sveučilište u Zagrebu
Fakultet elektrotehnike i računarstva

# Sadržaj predavanja

- Paralelizam
  - Višedretvenost
  - Višeprocesnost

- Mrežno programiranje: *socket API*

# Paralelizam: višedretvenost

# Procesi vs. dretve

- Proces
  - Program koji se izvršava
  - Ima vlastiti adresni prostor, memoriju, stog podataka…
  - Visoka izolacija
- Dretve (engl. *threads*)
  - Izvršavaju se unutar istog procesa
  - Niska izolacija
    - Dretve dijele isti kontekst
    - Programeri se moraju brinuti za sinkronizaciju na zajedničkim podatcima

# Višedretvenost

- Python GIL: *Global Interpreter Lock*
  - Samo jedna dretva može istodobno pristupiti Python interpreteru
    - jer većina interpretera nije thread-safe
  - Ako dretve sadrže čisti Python kod (tj. ako su CPU-bound), onda nema smisla koristiti dretve za ubrzanje
- Kada koristiti dretve u Pythonu?
  - Za ubrzanje koda koji koristi vanjske resurse ili poziva npr. C kod
  - Responzivna sučelja
    - Čekanje na neki događaj delegira se dretvi
  - Delegacija zadataka
    - Više dretvi + red poruka
  - Višekorisničke aplikacije
    - Npr. web poslužitelji

# Višedretvenost

```
import threading
import time

# Kod koji se izvršava u neovisnoj dretvi
def countdown(n):
    while n > 0:
        print('t-minus', n)
        n -= 1
        time.sleep(1)

# Stvori i pokreni dretvu
t = threading.Thread(target=countdown, args=(10,))
t.start()  # eksplicitni početak
```

# Višedretvenost

- Upravljanje stanjem dretve

```
# provjeri je li dretva živa
t.is_alive()
# blokiraj trenutnu dretvu dok se dretva t ne završi
t.join()
```

- Kritični odsječci (za dijeljeni pristup)

```
lock = threading.Lock()
lock.acquire()
lock.release()
```

... ili jednostavnije:

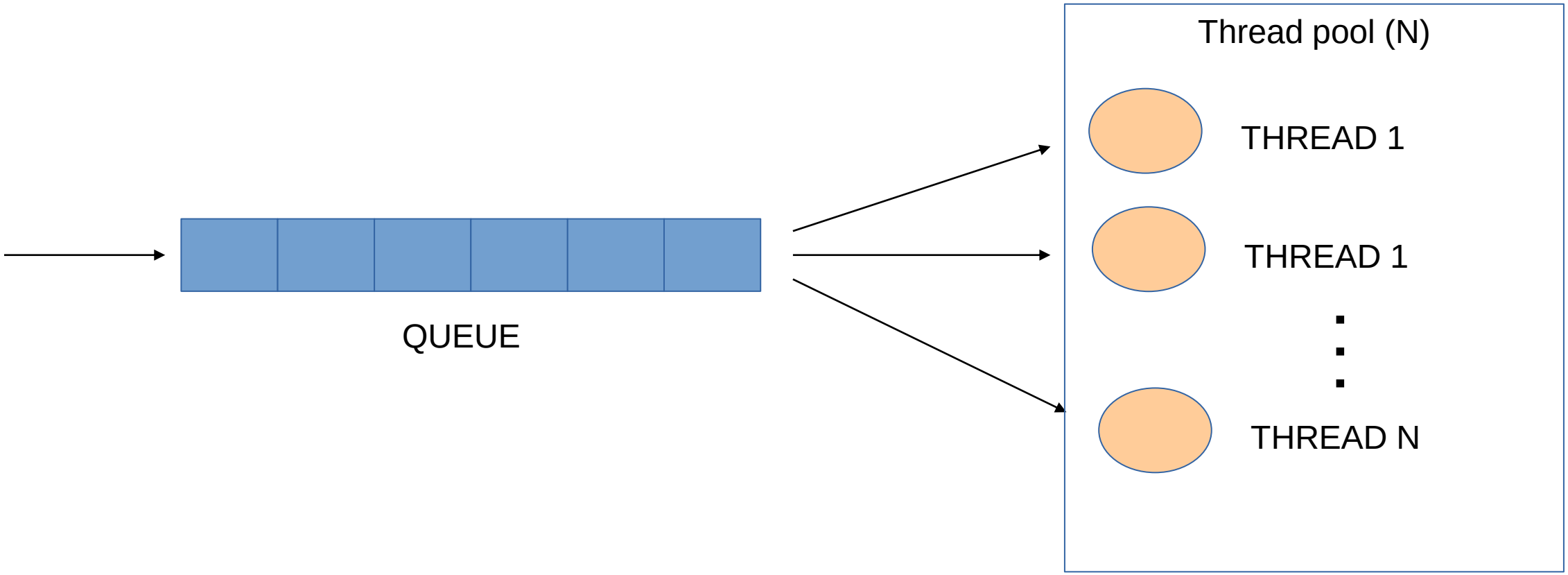
```
with lock:
    # kritični odsječak...
```

# Višedretvenost: *Queue*

- Dretve mogu komunicirati koristeći strukturu `Queue` iz modula `queue`
  - To je *thread-safe* struktura što znači da sama brine o sinkronizaciji (dijeljenom pristupu)
- Instanciranje: `red = queue.Queue()`
- Ubacivanje podatka u red: `red.put(value)`
- Vađenje (čekanje) podatka iz reda: `value = red.get()`

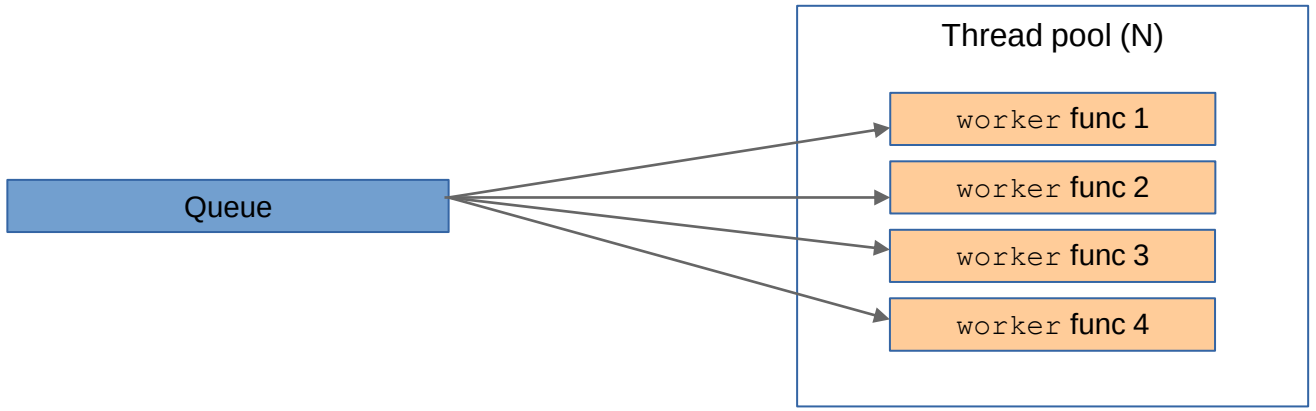
# Višedretvenost - primjer

- Bazen dretvi + red podataka



# Višedretvenost - primjer

```
import threading
import queue
work_queue = queue.Queue()
```




- Stvaranje dretvi:

```
NUM_THREADS = 4
threads = [
        threading.Thread(target=worker, args=(work_queue,))
        for i in range(NUM_THREADS)
    ]
```

- Pokreni dretve

```
for thread in threads:
    thread.start()
```

# Višedretvenost - primjer

- Funkcija koju izvodi pojedina dretva:

```
def worker(work_queue):
    while True:
        item = work_queue.get()
        # ... obradi item i ispisi rezultat ...
        work_queue.task_done()
```

- Glavna dretva ubacuje podatke za obradu:

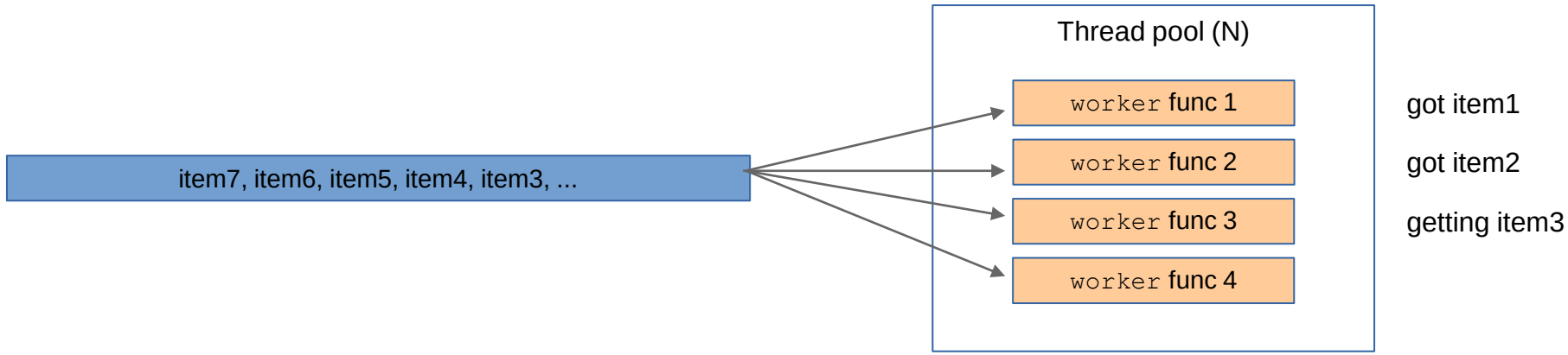
```
for i in items:
    work_queue.put(i)
```

# Višedretvenost - primjer

- Čišćenje:

```
# čekaj dok se ne dobiju i ne obrade svi elementi reda
work_queue.join()

# čekaj da sve dretve završe
while threads:
    threads.pop().join()
```

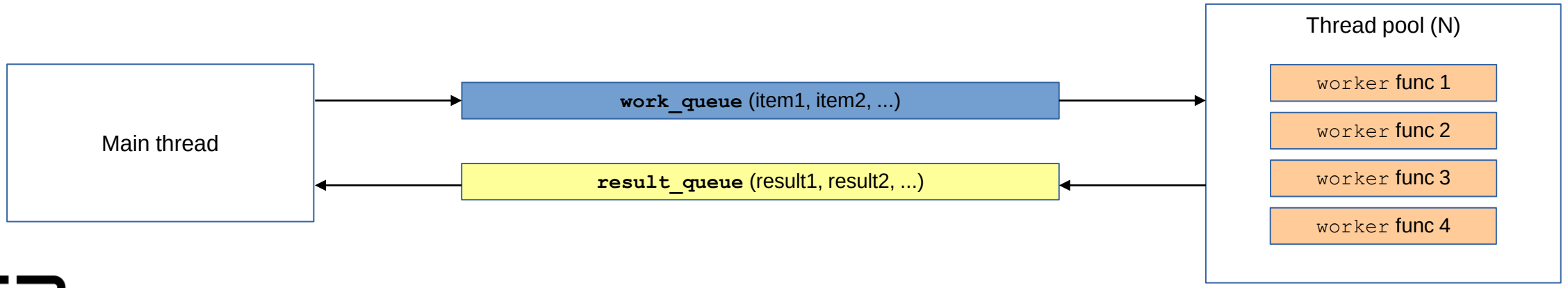


# Višedretvenost - vježba

- Problem: dretve se izvršavaju beskonačno dugo jer u petlji čekaju na novi element (`work_queue.get()`) iako ih više nema
- Zadatak: prepraviti kod tako da dretve prepoznaju kada su svi elementi obrađeni i završe.

# Višedretvenost - primjer

- Poboljšanje:
  - Ispis rezultata unutar dretvi nije praktičan (npr. nemamo kontrolu nad redoslijedom)
  - Najbolja je praksa tu odgovornost ostaviti glavnoj dretvi
  - Rješenje: dodaj još jedan red za rezultate



# Višedretvenost - primjer

- Stvori još jedan red:

```
results_queue = queue.Queue()
```

- Kreiraj bazen dretvi koje kao argumente primaju oba reda:

```
Thread(target=worker, args=(work_queue, results_queue))
```

- U worker funkciji, dodaj rezultat u red za rezultate:

```
results_queue.put(rezultat obrade itema)
```

- Ispiši sve rezultate:

```
while not results_queue.empty():
    print(results_queue.get())
```

# Višedretvenost - vježba

- Ispisati HTML sadržaj jedne od web stranica Google i Bing, one koja prva odgovori na zahtjev.
  - Svaka od dvije dretve paralelno dohvaća jedan URL i rezultat sprema u red
  - Glavna dretva ispisuje prvi element iz reda rezultata čim se on pojavi (`red.get()`)
  - Dohvaćanje HTML-a u worker funkciji dretve:

```
import urllib.request
sadrzaj = urllib.request.urlopen(url).read()
```

  - Dohvaćeni HTML treba ispisati u datoteku
  - Na kraju pozvati naredbu (web browser) koja će ga otvoriti

# Paralelizam: višeprocesnost

# Višeprocesnost

- Za ubrzanje čistog (CPU-bound) Python koda
  - Nema GIL ograničenja na jednu jezgru
- Sustavski poziv `fork()`
  - Stvara se novi (izolirani) process
  - Nema dijeljenja konteksta

```
import os
child_pid = os.fork()
if child_pid == 0:
    print('Child Process: PID', os.getpid())
else:
    print('Parent Process: PID', os.getpid())
```

# Modul `multiprocessing`

- Elegantniji način stvaranja i pokretanja novog procesa (interno se poziva *fork*)

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()  # pokretanje
    p.join()   # čeka završetak
```

# Primjer - **multiprocessing**

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())  # roditeljev ID
    print('process id:', os.getpid())       # moj ID

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

# Sinkronizacija (Lock)

```
from multiprocessing import Process, Lock

def f(lock, i):
    with lock:
        print('hello world’)
        print(i)

if __name__ == '__main__':
    l = Lock()
    for num in range(10):
        Process(target=f, args=(l, num)).start()
```

# Komunikacija (`Queue`)

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # ispis: [42, None, 'hello']
    p.join()
```

# Vježba - zadatak

- Kreirajte dva procesa: jedan računa zbroj svih vrijednosti *sin(x)* za x = 1, 2, ..., $10^7$, a drugi zbroj svih *cos(x)* za x = 1, 2, ..., $10^7$. Glavni proces treba ispisati rezultat koji prije završi.

# multiprocessing.Pool

- Primjenu iste funkcije na više objekata paraleliziramo među procesima
- Smisleno je definirati onoliko procesa koliko ima jezgri procesora (`os.cpu_count()`)

```
from multiprocessing import Pool

def f(x):
    return x * x

if __name__ == '__main__':
    with Pool(4) as p:
        argumenti = [1, 2, 3, 4, 5, 6, 7, , 9]
        rezultati = p.map(f, argumenti)
```

# Višeprocesnost - vježba

- Napisati program koji što brže računa sumu kvadrata svih brojeva od 1 do N. Posao treba podijeliti tako da interval od 1 do N podijelimo na četiri dijela – manja intervala.
- Testirati program za velike N (npr. $10^7$) te izmjeriti vrijeme izvođenja za rješenja s K = 1, 2 i 4 procesa.

# Mrežno programiranje: *socket API*

# Socket API

- Sučelje za mrežnu komunikaciju (*Berkeley sockets)*
- Socket je „utičnica” za slanje i primanje podataka
- Stvaramo je modulom *socket* standardne biblioteke:

  ```
  s = socket.socket(address family, socket type)
  ```

  - Za IPv4 adrese upotrebljavamo `socket.AF_INET`
  - Za TCP protokol upotrebljavamo `socket.SOCK_STREAM`
- Povezujemo je na određeno sučelje metodom:

  ```
  s.bind((HOST, PORT))
  ```
- Omogućujemo joj prihvat konekcija metodom:

  ```
  s.listen()
  ```
- Na klijentskoj strani:

  ```
  s.connect((SERVER_HOST, SERVER_PORT))
  ```

# Socket API: primanje i slanje podataka

- Čekanje na konekciju: `s.accept()` - blokira izvođenje
- Vraća uređeni par `conn, addr`: novi *socket* koji odgovara dolaznoj konekciji, te adresu spojenog klijenta (*host, port*)
- Komunikaciju ostvarujemo preko novog socketa metodama:

  `conn.recv(`*`max_bytes`*`)`

  - prima podatke kao niz bajtova
  - ako vrati prazan *bytes* objekt, klijent je zatvorio konekciju

  `conn.send(`*`bytes`*`)`

  - vraća broj uspješno poslanih bajtova

    → možda je potrebno ponovno pozvati
  - opetovano pozivanje može se automatizirati:

    `conn.sendall(`*`bytes`*`)`

# Socket API: primanje i slanje podataka

- Podatci se šalju i primaju kao niz bajtova – objekt tipa `bytes`
- Pretvorba stringa u bajtove:
  ```
  my_bytes = str.encode(my_str)
  ```
- Pretvorba bajtova u string:
  ```
  my_str = my_bytes.decode()
  ```
- Za proizvoljne Python objekte možemo koristiti:
  ```
  pickle.dumps(obj)
  ```
  - vraća niz bajtova
  ```
  pickle.loads(bytes)
  ```
  - vraća originalni objekt

# Socket API: server

- `s.close()` zatvara konekciju i socket
- Automatsko zatvaranje na kraju bloka:

```
with socket.socket(...) as s:
    ...
```

- Primjer jednostavnog *echo* servera:

```
import socket
HOST = "127.0.0.1"  # localhost
PORT = 65432
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

# Socket API: klijent

- Odgovarajući klijent:

```
import socket

HOST = "127.0.0.1"  # server's hostname/IP
PORT = 65432        # server’s port

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b"Hello, world")
    data = s.recv(1024)

print(f"Received {data!r}")
```

- Zadatak: simulirati dulju komunikaciju slanjem 10 poruka uz čekanje od 1 sekunde nakon svake poruke.

# Socket API: višestruke konekcije

- Kako omogućiti posluživanje većeg broja aktivnih klijenata?
- Svakog klijenta obrađuje zasebna dretva:

```
def on_new_client(conn, addr):
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    while True:
        conn, addr = s.accept()
        t = Thread(target=on_new_client, args=(conn, addr))
        t.start()
```

# Socket API: vježba (*chatroom*)

- Napišimo *chat-server.py* i *chat-client.py*
- Port zadajemo kao argument pri pokretanju iz komandne linije:

```
$ python chat-server.py 12000
$ python chat-client.py 12000
```

- U programu ga čitamo iz liste argumenata:

```
port = int(sys.argv[1])
```

- Klijent treba imati dvije dretve: jednu za upisivanje i slanje, drugu za primanje poruka
- Server održava globalnu listu konekcija da bi mogao proslijediti poruku (*broadcast*)