Generalitat de Catalunya
Departament d'Educació
Institut Thos i Codina

El mòdul 'socket' de Python

Domini Públic (CC0)

Índex

1. Conceptes previs **1**
Famílies d'adreces (AF_*) . . . 1
Tipus de socket (SOCK_*) . . . 1
TCP vs. UDP . . . 1
Adreça especial . . . 2

2. Un client TCP senzill **2**
`send()` vs. `sendall()` . . . 2
Forma curta: `socket.create_connection()` . . . 3

3. Un servidor TCP senzill (eco) **3**
Les quatre crides del servidor . . . 3
Per què SO_REUSEADDR? . . . 4
Forma curta: `socket.create_server()` (Python 3.8+) . . . 4
Client per provar l'eco . . . 4

4. TCP és un flux, no missatges (el concepte més important) **5**
a) Longitud fixa . . . 5
b) Delimitador (com HTTP o SMTP, amb `\r\n`) . . . 5
c) Prefix de longitud (el més robust) . . . 6

5. Atendre diversos clients alhora **6**
a) Un fil per client (`threading`) . . . 6
b) `socketserver` (alt nivell, a la biblioteca estàndard) . . . 7
c) Multiplexació amb `selectors` (un sol fil, molts clients) . . . 7
d) `asyncio` . . . 8

6. UDP: datagrames **8**
Difusió (*broadcast*) . . . 9

7. Temps màxims i mode no bloquejant **9**

8. Gestió d'errors **10**
`connect_ex()` . . . 10

9. Resolució de noms i funcions auxiliars **11**
`getaddrinfo()`: la forma moderna (IPv4 + IPv6) . . . 11
Cerca inversa . . . 11
Serveis i conversió d'adreces . . . 11
Ordre de bytes de xarxa . . . 11

10. Opcions de socket: `setsockopt` **12**

11. Xifrat: TLS amb `ssl` **12**

12. Sockets de domini Unix (comunicació local) **13**

13. Exemples complets **13**
13.1. Servidor de xat per línies (multiclient) . . . 13
13.2. Escàner de ports TCP (només per a xarxes pròpies) . . . 14

13.3. Comprovar si un servei respon (*health check*) 15
13.4. Protocol JSON amb prefix de longitud 15
13.5. Descobriment per UDP a la xarxa local 16

14. Seguretat i bones pràctiques **16**

15. Eines per depurar **17**

16. Errors habituals (llista de comprovació) **17**

17. Exercicis proposats **17**

18. Resum **18**

19. Referències **18**

El mòdul `socket` dona accés a la interfície de **sockets de xarxa** del sistema operatiu (l'API BSD sockets). És la base sobre la qual es construeixen quasi tots els protocols de xarxa (HTTP, SSH, SMTP, DNS…) i permet escriure clients i servidors a baix nivell, amb control total sobre què s'envia i com.

```
import socket
```

NOTA

Què és un socket? És un **punt final de comunicació**: un objecte que el sistema operatiu proporciona al programa per enviar i rebre dades per la xarxa. Una connexió TCP queda identificada per la quíntupla *(protocol, IP local, port local, IP remota, port remota)*. El socket és el "descriptor de fitxer" que representa un d'aquests extrems.

1. Conceptes previs

Famílies d'adreces (`AF_*`)

Constant	Significat	Format d'adreça
`socket.AF_INET`	IPv4	`(host, port)`
`socket.AF_INET6`	IPv6	`(host, port, flowinfo, scope_id)`
`socket.AF_UNIX`	Sockets locals (només Unix)	Camí d'un fitxer: `"/tmp/meu.sock"`

Tipus de socket (`SOCK_*`)

Constant	Protocol	Característiques
`socket.SOCK_STREAM`	**TCP**	Orientat a connexió, flux de bytes fiable i ordenat
`socket.SOCK_DGRAM`	**UDP**	Sense connexió, datagrames independents, sense garantia d'entrega ni d'ordre

TCP vs. UDP

	TCP	UDP
Connexió	Cal establir-la (*three-way handshake*)	No
Fiabilitat	Retransmet, ordena, detecta pèrdues	Cap garantia
Límits de missatge	**No existeixen** (és un flux continu)	Cada datagrama és un missatge
Ús típic	HTTP, SSH, SMTP, bases de dades	DNS, streaming, jocs, DHCP

Adreça especial

- `"127.0.0.1"` (o `"localhost"`): només es pot accedir des de la mateixa màquina.
- `"0.0.0.0"` (o `""`): escolta a **totes** les interfícies IPv4 (accessible des de la xarxa).
- Els ports `1--1023` són **privilegiats**: a Unix, cal ser root per fer-hi `bind`.

2. Un client TCP senzill

```
import socket

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect(("example.com", 80))
    s.sendall(b"GET / HTTP/1.1\r\nHost: example.com\r\nConnection:
     ↪  close\r\n\r\n")

    resposta = b""
    while True:
        dades = s.recv(4096)
        if not dades:          # b"" = l'altre extrem ha tancat la
         ↪  connexió
            break
        resposta += dades

print(resposta.decode("utf-8", errors="replace")[:500])
```

Punts importants:

1. **Els sockets envien i reben `bytes`**, no `str`. Cal fer `.encode()` en enviar i `.decode()` en rebre.
2. `connect()` rep una **tupla** `(host, port)`, no dos arguments.
3. `recv(n)` retorna **com a màxim** n bytes (pot retornar-ne menys) i **bloqueja** fins que arriba alguna cosa.
4. Quan `recv()` retorna `b""` (bytes buits), vol dir que l'altre extrem **ha tancat la connexió ordenadament**. No és un error, és el final del flux.
5. El `with` tanca el socket automàticament en sortir.

`send()` vs. `sendall()`

- `send(dades)` pot enviar només una part i retorna quants bytes ha enviat realment; cal reintentar amb la resta.
- `sendall(dades)` **continua fins a enviar-ho tot** (o llança una excepció). És el que es fa servir en la pràctica amb TCP.

Forma curta: `socket.create_connection()`

```
with socket.create_connection(("example.com", 80), timeout=5) as s:
    s.sendall(b"HEAD / HTTP/1.0\r\nHost: example.com\r\n\r\n")
    print(s.recv(1024).decode())
```

Resol el nom (amb `getaddrinfo`), prova totes les adreces retornades (IPv6 i IPv4) fins que una connecta, i admet un `timeout`. **És preferible a `connect` manual** quan es connecta a un nom de host.

3. Un servidor TCP senzill (eco)

```
import socket

HOST = "0.0.0.0"
PORT = 5000

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as servidor:
    servidor.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    servidor.bind((HOST, PORT))
    servidor.listen()
    print(f"Escoltant a {HOST}:{PORT}")

    conn, adreca = servidor.accept()       # bloqueja fins que arriba
     ↪  un client
    with conn:
        print("Connectat:", adreca)
        while True:
            dades = conn.recv(1024)
            if not dades:
                break
            conn.sendall(dades)            # retorna el mateix (eco)
```

Les quatre crides del servidor

Crida	Què fa
`bind((host, port))`	Associa el socket a una adreça i port locals
`listen([backlog])`	Marca el socket com a **passiu** (acceptarà connexions); `backlog` és la mida de la cua de connexions pendents
`accept()`	Bloqueja i retorna `(nou_socket, adreça_client)`
`recv/sendall`	Es fan sobre el **socket nou** retornat per `accept`, no sobre el d'escolta

IMPORTANT

El socket d'escolta i el socket de cada connexió són **objectes diferents**. El primer només serveix per acceptar; el segon parla amb un client concret.

Per què SO_REUSEADDR?

Quan un servidor TCP es tanca, el sistema operatiu manté la connexió en estat **`TIME_WAIT`** durant un temps (típicament 60 s) per assegurar-se que no arriben paquets endarrerits. Durant aquest temps, un nou `bind()` al mateix port falla amb:

```
OSError: [Errno 98] Address already in use
```

`SO_REUSEADDR` indica al sistema que permeti reutilitzar l'adreça igualment. És gairebé obligatori en servidors que es reinicien sovint.

NOTA

Windows: `SO_REUSEADDR` té una semàntica diferent (permet que un altre procés es vinculi al mateix port a la vegada). Allà és preferible `SO_EXCLUSIVEADDRUSE`.

Forma curta: `socket.create_server()` (Python 3.8+)

```
with socket.create_server(("0.0.0.0", 5000)) as servidor:
    conn, adreca = servidor.accept()
```

Fa `socket` + `SO_REUSEADDR` (a POSIX) + `bind` + `listen` en una sola crida. Amb `family=socket.AF_INET6, dualstack_ipv6=True` accepta IPv4 i IPv6 alhora.

Client per provar l'eco

```
import socket

with socket.create_connection(("localhost", 5000)) as s:
    s.sendall("Hola, món!".encode("utf-8"))
    print(s.recv(1024).decode("utf-8"))
```

També es pot provar des de la terminal amb `nc localhost 5000` o `telnet localhost 5000`.

4. TCP és un flux, no missatges (el concepte més important)

TCP **no conserva els límits dels missatges**. Si el client fa:

```
s.sendall(b"Hola")
s.sendall(b"Adeu")
```

el servidor pot rebre `b"HolaAdeu"` en un sol `recv`, o `b"Ho"` i després `b"laAdeu"`, o de qualsevol altra manera. El que és garantit és **l'ordre i que no es perd cap byte**, però no com queden agrupats.

Per això, qualsevol protocol sobre TCP necessita **delimitar els missatges**. Hi ha tres estratègies habituals:

a) Longitud fixa

Tots els missatges mesuren el mateix. Simple, però poc flexible.

b) Delimitador (com HTTP o SMTP, amb `\r\n`)

```
def llegeix_linia(conn):
    buf = b""
    while not buf.endswith(b"\n"):
        trosset = conn.recv(1)
        if not trosset:
            break
        buf += trosset
    return buf
```

Una alternativa molt més còmoda és `makefile()`, que embolcalla el socket com un fitxer:

```
with conn:
    f = conn.makefile("rwb")
    for linia in f:                 # itera línia a línia
        f.write(linia.upper())
        f.flush()
```

c) Prefix de longitud (el més robust)

Cada missatge s'envia precedit d'un enter de 4 bytes amb la seva mida:

```
import struct

def envia_missatge(sock, dades: bytes):
    sock.sendall(struct.pack("!I", len(dades)) + dades)

def rep_exacte(sock, n):
    """Llegeix exactament n bytes (recv pot retornar-ne menys)."""
    buf = bytearray()
    while len(buf) < n:
        trosset = sock.recv(n - len(buf))
        if not trosset:
            raise ConnectionError("Connexió tancada abans d'hora")
        buf.extend(trosset)
    return bytes(buf)

def rep_missatge(sock) -> bytes:
    (longitud,) = struct.unpack("!I", rep_exacte(sock, 4))
    return rep_exacte(sock, longitud)
```

`"!I"` vol dir *network byte order* (gran *endian*) i enter sense signe de 4 bytes. Això permet enviar qualsevol contingut (inclòs binari) i saber sempre on acaba cada missatge.

PRECAUCIÓ

Error típic de principiant: suposar que un `sendall` correspon a exactament un `recv`. Funciona a la xarxa local amb missatges petits, i falla de manera intermitent en producció.

5. Atendre diversos clients alhora

El servidor d'eco anterior atén **un sol client** i bloqueja els altres. Hi ha diverses solucions.

a) Un fil per client (`threading`)

```
import socket
import threading

def atén(conn, adreca):
    with conn:
        print("Connectat:", adreca)
        while True:
            dades = conn.recv(1024)
            if not dades:
                break
            conn.sendall(dades)
```

```
    print("Desconnectat:", adreca)

with socket.create_server(("0.0.0.0", 5000)) as servidor:
    while True:
        conn, adreca = servidor.accept()
        threading.Thread(target=atén, args=(conn, adreca),
        ↪  daemon=True).start()
```

Senzill i suficient per a poques desenes de connexions.

b) `socketserver` (alt nivell, a la biblioteca estàndard)

```
import socketserver

class GestorEco(socketserver.BaseRequestHandler):
    def handle(self):
        while True:
            dades = self.request.recv(1024)
            if not dades:
                break
            self.request.sendall(dades)

with socketserver.ThreadingTCPServer(("0.0.0.0", 5000), GestorEco) as
↪   srv:
    srv.allow_reuse_address = True
    srv.serve_forever()
```

c) Multiplexació amb `selectors` (un sol fil, molts clients)

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accepta(servidor):
    conn, adreca = servidor.accept()
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, llegeix)

def llegeix(conn):
    dades = conn.recv(1024)
    if dades:
        conn.sendall(dades)        # simplificació: en producció
        ↪  caldria gestionar buffers d'enviament
    else:
        sel.unregister(conn)
        conn.close()
```

```
servidor = socket.create_server(("0.0.0.0", 5000))
servidor.setblocking(False)
sel.register(servidor, selectors.EVENT_READ, accepta)

while True:
    for clau, _ in sel.select():
        clau.data(clau.fileobj)    # crida a accepta() o llegeix()
```

`selectors` usa `epoll` (Linux), `kqueue` (macOS/BSD) o `select` (Windows) segons el sistema. Escala molt millor que un fil per client.

d) `asyncio`

Per a servidors amb molta concurrència, `asyncio.start_server()` ofereix una API més còmoda amb `async/await` sense haver de gestionar els sockets a mà.

6. UDP: datagrames

```
# --- servidor UDP ---
import socket

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.bind(("0.0.0.0", 6000))
    while True:
        dades, adreca = s.recvfrom(1024)
        print(f"{adreca}: {dades.decode()}")
        s.sendto(b"rebut", adreca)
```

```
# --- client UDP ---
import socket

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.settimeout(2)
    s.sendto("Hola".encode(), ("localhost", 6000))
    try:
        resposta, _ = s.recvfrom(1024)
        print(resposta.decode())
    except socket.timeout:
        print("Sense resposta (el datagrama o la resposta s'ha pogut
        ↪  perdre)")
```

Diferències respecte a TCP:

- **No hi ha `listen`, `accept` ni `connect`** (tot i que `connect` és opcional i serveix només per fixar l'adreça per defecte).
- Es fa servir `sendto(dades, adreça)` i `recvfrom(n)`, que retorna `(dades, adreça_remitent)`.

- **Cada `sendto` és un datagrama independent** i es conserven els límits: un `recvfrom` retorna exactament un datagrama (si n és prou gran; si no, es trunca).
- No hi ha garanties: els datagrames es poden perdre, duplicar o arribar desordenats. Si necessites fiabilitat, l'aplicació l'ha d'implementar (o usar TCP).
- Mida pràctica: evita datagrames de més d'uns ~1400 bytes per no fragmentar-los (MTU típica d'Ethernet 1500).

Difusió (*broadcast*)

```
with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    s.sendto(b"Hi ha algú?", ("255.255.255.255", 6000))
```

7. Temps màxims i mode no bloquejant

Per defecte, les operacions de socket **bloquegen indefinidament**. Un servidor que mai respon et pot deixar el programa penjat per sempre.

```
s = socket.socket()
s.settimeout(5.0)               # 5 segons per a cada operació

try:
    s.connect(("example.com", 80))
    s.recv(1024)
except socket.timeout:          # des de Python 3.10 és un àlies de
 ↪  TimeoutError
    print("Temps esgotat")
```

Mode	Crida	Comportament
Bloquejant	`settimeout(None)` (per defecte)	Espera indefinidament
Amb temps màxim	`settimeout(5.0)`	Llança `TimeoutError` després de 5 s
No bloquejant	`setblocking(False)` o `settimeout(0)`	Si no pot fer res ara, llança `BlockingIOError`

El mode no bloquejant s'usa amb `selectors`/`select` o `asyncio`, no s'hi fa polling manual en un bucle.

8. Gestió d'errors

Gairebé tots els errors de xarxa són subclasses d'`OSError`:

Excepció	Causa típica
`ConnectionRefusedError`	No hi ha res escoltant a aquest port (o un tallafocs rebutja)
`TimeoutError` / `socket.timeout`	L'operació ha superat el temps màxim
`ConnectionResetError`	L'altre extrem ha tancat de cop (RST)
`BrokenPipeError`	S'escriu en una connexió que l'altre extrem ja ha tancat
`socket.gaierror`	No es pot resoldre el nom (error de DNS)
`OSError: [Errno 98] Address already in use`	El port ja està ocupat (o és en `TIME_WAIT`)
`OSError: [Errno 13] Permission denied`	`bind` a un port privilegiat sense permisos
`OSError: [Errno 113] No route to host`	No hi ha ruta a la destinació

```
import socket

try:
    with socket.create_connection(("srv-dc01", 389), timeout=3):
        print("Port 389 obert")
except socket.gaierror:
    print("No es resol el nom")
except ConnectionRefusedError:
    print("Port tancat")
except TimeoutError:
    print("Sense resposta (probablement un tallafocs descarta els
    ↪  paquets)")
except OSError as e:
    print("Error:", e)
```

`connect_ex()`

Variant de `connect` que **retorna un codi d'error en lloc de llançar una excepció** (0 si ha anat bé):

```
s = socket.socket()
s.settimeout(1)
codi = s.connect_ex(("192.168.1.10", 22))
print("obert" if codi == 0 else f"tancat o filtrat (errno {codi})")
s.close()
```

9. Resolució de noms i funcions auxiliars

```
socket.gethostname()                # nom d'aquesta màquina
socket.getfqdn()                    # nom de domini complet
socket.gethostbyname("example.com")  # '93.184.216.34' (només IPv4)
```

`getaddrinfo()`: la forma moderna (IPv4 + IPv6)

```
for familia, tipus, proto, nom_canonic, adreca in socket.getaddrinfo(
        "example.com", 443, type=socket.SOCK_STREAM):
    print(familia.name, adreca)
```

Retorna totes les adreces del nom (A i AAAA). És el que fa servir internament `create_connection`. `gethostbyname` només retorna IPv4 i s'ha de considerar de l'època anterior.

Cerca inversa

```
nom, alias, adreces = socket.gethostbyaddr("8.8.8.8")
print(nom)    # dns.google
```

Serveis i conversió d'adreces

```
socket.getservbyname("https")        # 443
socket.getservbyport(22)             # 'ssh'

socket.inet_aton("192.168.1.1")      # b'\xc0\xa8\x01\x01' (4 bytes)
socket.inet_ntoa(b"\xc0\xa8\x01\x01")# '192.168.1.1'
socket.inet_pton(socket.AF_INET6, "::1")   # IPv6 binari
```

Per treballar amb adreces i xarxes (comprovar si una IP pertany a una subxarxa, calcular màscares...) és molt més còmode el mòdul `ipaddress`.

Ordre de bytes de xarxa

La xarxa fa servir *big-endian*. `socket.htons()`/`ntohs()` (16 bits) i `htonl()`/`ntohl()` (32 bits) converteixen entre l'ordre de la màquina i el de xarxa. En Python moderni és més pràctic `struct.pack("!...")`.

10. Opcions de socket: `setsockopt`

```
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)     #
 ↪  reutilitzar adreça
s.setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)     # sondes
 ↪  per detectar connexions mortes
s.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)     #
 ↪  desactiva l'algorisme de Nagle
s.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, 262144)   # mida del
 ↪  buffer de recepció
```

- **`TCP_NODELAY`:** per defecte, TCP agrupa dades petites abans d'enviar-les (algorisme de Nagle). Per a protocols interactius (peticions petites i resposta ràpida) conviu desactivar-ho.
- **`SO_KEEPALIVE`:** sense això, una connexió pot quedar "oberta" indefinidament després que l'altra màquina s'hagi apagat sense tancar-la.

Es llegeixen amb `getsockopt(...)`. Per consultar adreces: `s.getsockname()` (local) i `s.getpeername()` (remota).

11. Xifrat: TLS amb `ssl`

Un socket normal envia les dades **en clar**. Per xifrar-les s'embolcalla el socket amb `ssl`:

```
import socket
import ssl

context = ssl.create_default_context()      # valida certificats i nom
 ↪  de host

with socket.create_connection(("example.com", 443), timeout=5) as
 ↪  sock:
    with context.wrap_socket(sock, server_hostname="example.com") as
     ↪  ssock:
        print("Versió TLS:", ssock.version())
        ssock.sendall(b"HEAD / HTTP/1.1\r\nHost:
         ↪  example.com\r\nConnection: close\r\n\r\n")
        print(ssock.recv(1024).decode(errors="replace"))
```

`server_hostname` és imprescindible: serveix per a SNI i per **verificar que el certificat correspon al servidor**. Desactivar la verificació (`CERT_NONE`) elimina la protecció contra atacs *man-in-the-middle*; només és acceptable en proves de laboratori.

12. Sockets de domini Unix (comunicació local)

Per a processos de la mateixa màquina, els sockets Unix són més ràpids i es controlen amb permisos de fitxer:

```
import os, socket

RUTA = "/tmp/meu.sock"
if os.path.exists(RUTA):
    os.remove(RUTA)       # un bind falla si el fitxer ja existeix

with socket.socket(socket.AF_UNIX, socket.SOCK_STREAM) as srv:
    srv.bind(RUTA)
    srv.listen()
    conn, _ = srv.accept()
    with conn:
        print(conn.recv(1024))
```

El client fa `connect("/tmp/meu.sock")`. Així funcionen, per exemple, Docker (`/var/run/docker.sock`) o PostgreSQL en local.

13. Exemples complets

13.1. Servidor de xat per línies (multiclient)

```
import socket
import threading

clients = {}
bloqueig = threading.Lock()

def difon(missatge, excepte=None):
    with bloqueig:
        for conn in list(clients):
            if conn is not excepte:
                try:
                    conn.sendall(missatge)
                except OSError:
                    pass

def atén(conn, adreca):
    f = conn.makefile("rb")
    try:
        conn.sendall(b"Com et dius? ")
        nom = f.readline().decode().strip() or str(adreca)
        with bloqueig:
            clients[conn] = nom
        difon(f"* {nom} s'ha connectat\n".encode(), excepte=conn)
        for linia in f:
            difon(f"{nom}: {linia.decode()}".encode(), excepte=conn)
```

```
    finally:
        with bloqueig:
            clients.pop(conn, None)
        difon(f"* {nom} ha marxat\n".encode())
        conn.close()

with socket.create_server(("0.0.0.0", 5000)) as srv:
    print("Xat a 0.0.0.0:5000")
    while True:
        conn, adreca = srv.accept()
        threading.Thread(target=atén, args=(conn, adreca),
        ↪  daemon=True).start()
```

Es prova obrint dues terminals amb `nc localhost 5000`.

13.2. Escàner de ports TCP (només per a xarxes pròpies)

```
import socket
from concurrent.futures import ThreadPoolExecutor

def prova(host, port):
    with socket.socket() as s:
        s.settimeout(0.5)
        return port if s.connect_ex((host, port)) == 0 else None

def escaneja(host, ports):
    with ThreadPoolExecutor(max_workers=100) as pool:
        resultats = pool.map(lambda p: prova(host, p), ports)
    return [p for p in resultats if p]

obert = escaneja("127.0.0.1", range(1, 1025))
for port in obert:
    try:
        servei = socket.getservbyport(port)
    except OSError:
        servei = "desconegut"
    print(f"{port}/tcp obert ({servei})")
```

NOTA

Escaneja només màquines i xarxes que administris o per a les quals tinguis autorització explícita. En entorns d'aula, fes-ho en la xarxa de laboratori.

13.3. Comprovar si un servei respon (*health check*)

```
import socket

def servei_actiu(host, port, timeout=2):
    try:
        with socket.create_connection((host, port), timeout=timeout):
            return True
    except OSError:
        return False

for nom, host, port in [("DNS", "8.8.8.8", 53), ("HTTP",
 ↪  "example.com", 80), ("SSH", "localhost", 22)]:
    print(f"{nom:5} {host}:{port:<5} {'▫' if servei_actiu(host, port)
     ↪  else '▫'}")
```

13.4. Protocol JSON amb prefix de longitud

```
import json
import socket
import struct

def envia_json(sock, objecte):
    dades = json.dumps(objecte).encode("utf-8")
    sock.sendall(struct.pack("!I", len(dades)) + dades)

def rep_exacte(sock, n):
    buf = bytearray()
    while len(buf) < n:
        t = sock.recv(n - len(buf))
        if not t:
            raise ConnectionError("Connexió tancada")
        buf.extend(t)
    return bytes(buf)

def rep_json(sock, maxim=1_000_000):
    (n,) = struct.unpack("!I", rep_exacte(sock, 4))
    if n > maxim:                       # defensa davant de mides
     ↪  absurdes
        raise ValueError("Missatge massa gran")
    return json.loads(rep_exacte(sock, n))
```

El límit màxim impedeix que un client malintencionat sol·liciti un buffer de gigabytes.

13.5. Descobriment per UDP a la xarxa local

```
# Servidor: respon a "DISCOVER"
import socket

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.bind(("", 9999))
    while True:
        dades, adreca = s.recvfrom(1024)
        if dades == b"DISCOVER":
            s.sendto(f"{socket.gethostname()}".encode(), adreca)
```

```
# Client: envia per broadcast i recull respostes durant 2 s
import socket

with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    s.settimeout(2)
    s.sendto(b"DISCOVER", ("255.255.255.255", 9999))
    try:
        while True:
            nom, adreca = s.recvfrom(1024)
            print(adreca[0], nom.decode())
    except socket.timeout:
        pass
```

14. Seguretat i bones pràctiques

1. **Valida i limita tot el que arriba de la xarxa.** Mida màxima de missatge, nombre de connexions, temps màxim per connexió. Un servidor sense límits és fàcil de tombar.
2. **Posa sempre un `timeout`** als clients; un servidor mort no ha de bloquejar el teu programa.
3. **No facis `bind` a `0.0.0.0` si no cal.** `127.0.0.1` evita exposar el servei a la xarxa.
4. **No facis servir `eval()`, `pickle.loads()` ni similars** sobre dades rebudes: permeten execució de codi remot.
5. **Xifra amb TLS** qualsevol dada delicada i no desactivis la verificació de certificats.
6. **Tanca sempre els sockets** (`with` o `try/finally`) per no esgotar descriptors de fitxer.
7. **Gestiona `b""` a `recv`** com a fi de connexió, i `ConnectionResetError` com a cas normal.
8. **Per a protocols reals, usa biblioteques fetes** (`http.client`, `smtplib`, `ftplib`, `paramiko`, `requests`…). Els sockets crus són per aprendre, per a protocols propis o per a eines de diagnòstic.

15. Eines per depurar

Eina	Ús
`ss -tlnp` (Linux)	Llista els ports TCP en escolta i el procés propietari
`ss -tanp`	Connexions TCP actives
`nc host port` / `nc -l port`	Client i servidor TCP ràpids per a proves
`nc -u`	El mateix amb UDP
`tcpdump -i any port 5000`	Capturar el trànsit d'un port
Wireshark	Anàlisi gràfica de paquets
`telnet host port`	Prova de connectivitat bàsica
`netstat -an` (Windows)	Estat de connexions i ports

16. Errors habituals (llista de comprovació)

1. **Enviar `str` en lloc de `bytes`** → `TypeError: a bytes-like object is required`. Usa `.encode()`.
2. **Suposar que un send = un `recv`** en TCP. Usa un protocol de delimitació.
3. **Usar send en lloc de `sendall`** i perdre dades en silenci.
4. **No gestionar `b""`** a `recv` i entrar en un bucle infinit consumint CPU.
5. **`Address already in use`** en reiniciar el servidor: falta `SO_REUSEADDR` o `create_server`.
6. **`ConnectionRefusedError`**: servidor no iniciat, port equivocat o tallafocs.
7. **Cap `timeout`** i el programa penjat per sempre.
8. **`bind` a `localhost` i provar des d'una altra màquina**: no hi arriba; cal `0.0.0.0` o la IP de la interfície.
9. **Oblidar el tallafocs** (`ufw`, `firewalld`, Windows Defender Firewall) que bloqueja el port entrant.
10. **Confondre el socket d'escolta amb el de connexió** (`accept` retorna un socket nou).

17. Exercicis proposats

1. Escriu un servidor d'eco TCP i prova'l amb `nc`. Després fes-lo atendre diversos clients amb fils.
2. Modifica el servidor perquè converteixi a majúscules cada línia rebuda, usant `makefile()`.
3. Implementa el protocol amb prefix de longitud i comprova que funciona enviant un missatge d'1 MiB (que no cabrà en un sol `recv`).
4. Escriu un client que descarregui la capçalera HTTP d'una web amb sockets crus i mostri només la primera línia (codi d'estat).
5. Fes un servidor UDP que respongui amb l'hora actual a cada datagrama `TIME`, i un client amb `timeout`.
6. Escriu una funció `port_obert(host, port)` i usa-la per comprovar els serveis del teu laboratori (`thos.local`): DNS (53), LDAP (389), SMB (445), RDP (3389).
7. Converteix el servidor de l'exercici 1 en un servidor `selectors` d'un sol fil i compara'n el comportament amb 50 connexions simultànies.

8. Fes que el client TLS de la secció 11 mostri el certificat del servidor (`ssock.getpeercert()`) i comprova què passa si es connecta per IP en lloc de pel nom.

18. Resum

Vull…	Eina
Client TCP	`socket.create_connection((host, port), timeout=...)`
Servidor TCP	`socket.create_server((host, port))` + `accept()`
Enviar tot el que cal	`sock.sendall(bytes)`
Rebre dades	`sock.recv(n)` (retorna `b""` en tancar)
Delimitar missatges	Prefix de longitud (`struct`) o delimitador / `makefile()`
Datagrama UDP	`sendto(dades, adreça)` / `recvfrom(n)`
Temps màxim	`sock.settimeout(segons)`
Resoldre noms (IPv4+IPv6)	`socket.getaddrinfo(host, port)`
Provar un port sense excepcions	`sock.connect_ex((host, port))`
Reutilitzar el port ràpidament	`SO_REUSEADDR`
Molts clients	`threading`, `selectors`, `socketserver` o `asyncio`
Xifrar	`ssl.create_default_context().wrap_socket(...)`
Comunicació local ràpida	`AF_UNIX`

19. Referències

- Documentació oficial de `socket`: https://docs.python.org/es/3.15/library/socket.html
- HOWTO de programació amb sockets: https://docs.python.org/es/3.15/howto/sockets.html
- `selectors`: https://docs.python.org/es/3.15/library/selectors.html
- `socketserver`: https://docs.python.org/es/3.15/library/socketserver.html
- `ssl`: https://docs.python.org/es/3.15/library/ssl.html
- `ipaddress`: https://docs.python.org/es/3.15/library/ipaddress.html
- `asyncio` (streams): https://docs.python.org/es/3.15/library/asyncio-stream.html

Versions d'aquest document

- HTML
- PDF
- ODT
- MD

Domini Públic (CC0)