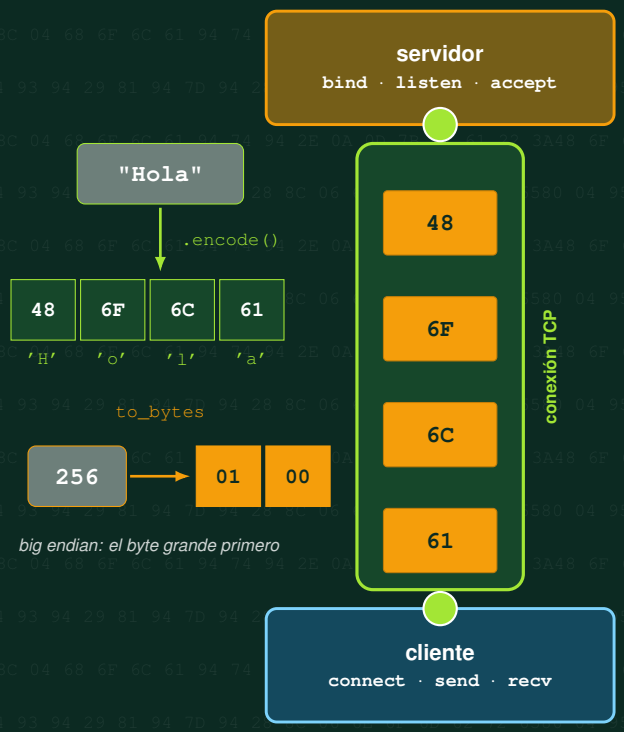
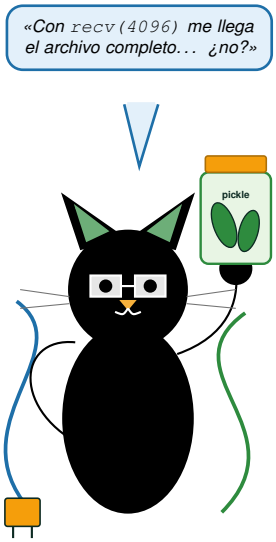
IIC2233 • PROGRAMACIÓN AVANZADA

SOCKETS BYTES Y PICKLE

`bytes` • `pickle` • TCP/UDP • hilos • P2P

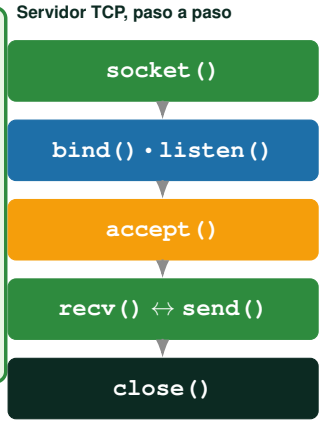
de un objeto en memoria a bytes, y de bytes a otro computador





Felipín quiso hacer un chat. Abrió un socket y nunca lo cerró, leyó un archivo de 3 MB con un solo `recv(4096)` y se preguntó por qué llegó cortado, y guardó los números como texto sin saber en qué *orden* iban sus bytes. Para peor, cargó con `pickle.loads` un archivo que le mandó un desconocido…

No seas como Felipín. En este resumen verás cómo se representan los datos como **bytes**, cómo **serializarlos** con `pickle` (y cuándo *no* hacerlo), y cómo construir **clientes y servidores** con **sockets**.



Lo que viaja por la red: `pickle.dumps(("a", 1, 3, "hola"))`

80	04	95	13	00	00	00	00	00	00	00	28	··· (30 bytes)



Resumen de Contenidos

IIC2233 — Programación Avanzada • Edición Salmuera

Resumen elaborado por el equipo de ayudantía, en base al material oficial del curso

Tabla de Contenidos

1 Introducción 2

2 Serialización con `pickle` 3

3 Manejo de bytes (*Input*/*Output*) 6

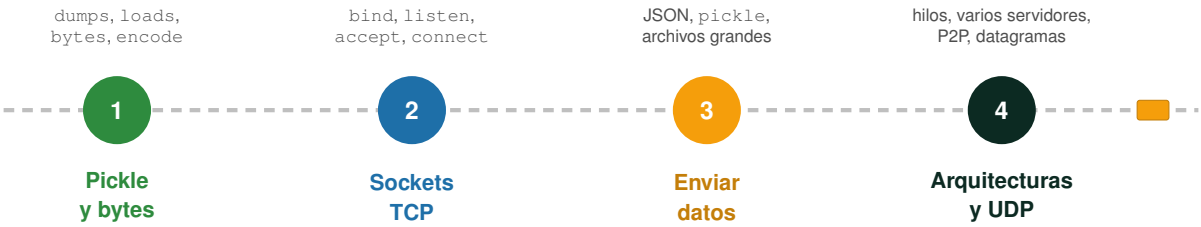
4 Sockets y comunicación TCP 11

5 Envío de datos por la red 16

6 Arquitecturas con múltiples conexiones 21

7 Bonus: sockets UDP 28

8 Hoja de referencia 30



1 Introducción

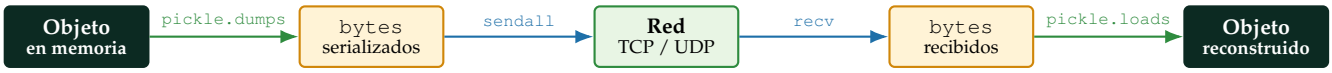
Este resumen reúne el contenido de **Networking con sockets** de IIC2233: cómo se representan los datos como **bytes**, cómo se **serializan** objetos con `pickle` y cómo dos programas (en el mismo computador o en computadores distintos) se comunican con **sockets**, tanto del lado del **cliente** como del **servidor**. Cerramos con arquitecturas más completas: un servidor con muchos clientes, un cliente con varios servidores y comunicación *peer-to-peer*.

■ Los cuatro bloques de este resumen

- **Serialización con `pickle` y manejo de bytes** (Secciones 2 y 3): guardar y cargar casi cualquier objeto de Python, personalizar el proceso con `__getstate__` y `__setstate__`, y entender `bytes`, `bytearray`, *encoding*, *chunks* y *endianness*.
- **Sockets TCP** (Sección 4): los pasos `socket`, `bind`, `listen`, `accept`, `connect`, `send`/`recv` y `close`, para construir un cliente y un servidor.
- **Envío de datos** (Sección 5): enviar JSON y `pickle` por un *socket*, y cómo transmitir mensajes grandes con un **protocolo** de largo + contenido.
- **Arquitecturas y UDP** (Secciones 6 y 7): servidor concurrente con hilos, cliente de varios servidores, P2P, y el protocolo UDP (bonus).

★ Cómo leer este resumen

Cada bloque de código va acompañado de una explicación *"¿Qué hace este código?"*. Los recuadros **rojos** marcan errores típicos y trampas; los **ámbar** son definiciones importantes; los **azules** son **analogías telefónicas** para fijar ideas. Una idea central se repite todo el resumen: *un socket solo transmite bytes, así que ambos lados deben ponerse de acuerdo en cómo interpretarlos*.



el **ciclo completo** de un dato que viaja de un programa a otro

2 Serialización con `pickle`

2.1 Otra forma de serializar

Ya vimos que **serializar** es transformar un objeto a una representación que se puede guardar o enviar, y **deserializar** es reconstruirlo. Con JSON la representación era *texto*. Python incluye además el módulo `pickle`, que serializa **directamente a bytes** y permite guardar y cargar **casi cualquier objeto de Python**, incluidas instancias de nuestras propias clases.

Característica	JSON	`pickle`
Representación	Texto (`str`)	Bytes (`bytes`)
Legible por personas	Sí	No
Compatible con otros lenguajes	Sí, es un estándar	No, es específico de Python
Qué puede guardar	Tipos básicos (y clases propias con trabajo extra)	Casi cualquier objeto de Python
Seguridad al cargar datos ajenos	Segura (solo datos)	**No es segura**

2.2 `dumps` y `loads`: serializar en memoria

El módulo ofrece dos funciones fundamentales:

- `pickle.dumps(objeto)`: **serializa**, es decir, *guarda* el objeto como una secuencia de bytes. Retorna un objeto `bytes`.
- `pickle.loads(bytes)`: **deserializa**, es decir, *carga* el objeto y lo devuelve a su estado original.

Una vez serializado, el objeto es **persistente**: está listo para ser usado de nuevo por el mismo u otro programa.

```python
import pickle

tupla = ("a", 1, 3, "hola")
tupla_serializada = pickle.dumps(tupla)

print(f"Resultado serialización: {tupla_serializada}")
print(f"Tipo de versión serializada: {type(tupla_serializada)}")

tupla_deserializada = pickle.loads(tupla_serializada)
print(f"Resultado deserialización: {tupla_deserializada}")

print(f"¿Las tuplas son iguales? {tupla == tupla_deserializada}")
print(f"¿Las tuplas son el mismo objeto? {tupla is tupla_deserializada}")
```

Salida

```
Resultado serialización: b'\x80\x04\x95\x13\x00\x00\x00\x00\x00\x00\x00(\x8c\x01a\x94K\x01K\x03
    \x8c\x04hola\x94t\x94.'
Tipo de versión serializada: <class 'bytes'>
Resultado deserialización: ('a', 1, 3, 'hola')
¿Las tuplas son iguales? True
¿Las tuplas son el mismo objeto? False
```

¿Qué hace este código? `dumps` entrega un objeto de tipo `bytes`; el contenido no es legible para una persona, pero Python sí sabe interpretarlo (los primeros bytes, `\x80\x04`, indican la versión del formato). Al llamar `loads` obtenemos una tupla **igual** a la original (`==` compara contenido), pero es un **objeto nuevo** en memoria (`is` compara identidad). A diferencia de JSON, aquí la tupla *sí vuelve como tupla*.

2.3 `dump` y `load`: serializar a archivos

Existen también `dump` y `load` (los mismos nombres, **sin la s**). Hacen lo mismo, pero **a través de archivos**:

Función	Dirección	Trabaja con	Retorna
`dumps(obj)`	serializa	memoria	un objeto `bytes`
`loads(bytes)`	deserializa	memoria	el objeto original
`dump(obj, archivo)`	serializa	archivo abierto	nada: escribe en el archivo
`load(archivo)`	deserializa	archivo abierto	el objeto original

Como trabajamos con **bytes** y no con *strings*, los archivos deben abrirse en **modo binario**: `"wb"` para escribir y `"rb"` para leer (en lugar del modo texto, que es el predeterminado).

```python
from os import path

lista = [1, 2, 3, 7, 8, 3]

with open(path.join("data", "mi_lista.bin"), "wb") as file:
    pickle.dump(lista, file)

with open(path.join("data", "mi_lista.bin"), "rb") as file:
    lista_cargada = pickle.load(file)

print(f"Lista original: {lista}")
print(f"Lista cargada : {lista_cargada}")
print(f"¿Las listas son iguales? {lista == lista_cargada}")        # True
print(f"¿Las listas son el mismo objeto? {lista is lista_cargada}")  # False
```

¿Qué hace este código? Guarda la lista en el archivo `mi_lista.bin` (la extensión es solo una convención) y luego la lee de vuelta. Usamos `with` para que el archivo se cierre solo, y `path.join` para construir la ruta sin depender del sistema operativo.

! `pickle` NO es seguro: nunca cargues datos desconocidos

Nunca cargues (`load`/`loads`) un *pickle* cuya procedencia no conoces. El formato permite indicar *"para reconstruir este objeto, ejecuta tal función con estos argumentos"*, así que un archivo (o un mensaje de red) malicioso puede **ejecutar código arbitrario en tu computador** apenas lo deserializas. Para datos que vienen de clientes o de internet, usa un formato de solo datos como **JSON**. En el apunte original hay un enlace a una demostración de este riesgo; no es necesario entrar en detalles.

★ `pickle` guarda el estado, no el código

Al serializar una instancia, `pickle` guarda el **nombre de la clase** y el **estado** del objeto (sus atributos), pero **no el código de la clase**. Por eso, para deserializar, la clase debe estar definida (importable) en el programa que hace el `loads`. Si el receptor no tiene la clase `Persona`, no podrá reconstruir una `Persona`.

2.4 Personalizar la serialización: `__getstate__`

Cuando `pickle` intenta serializar un objeto, primero revisa si su clase tiene implementado el método `__getstate__`. Este método debe **retornar el diccionario** con los atributos que se quieren serializar. Si no está implementado, `pickle` guarda directamente el atributo `__dict__` del objeto.

■ `__dict__`: dónde viven los atributos

`__dict__` es un diccionario que guarda los atributos de un objeto. En otras palabras, `objeto.atributo` equivale a `objeto.__dict__["atributo"]` y `objeto.atributo = 42` equivale a `objeto.__dict__["atributo"] = 42`.

Implementar `__getstate__` nos permite construir un diccionario con **solo la información que deseamos guardar**, o modificarla justo antes de serializar.

```python
class Persona:

    def __init__(self, nombre: str, edad: int) -> None:
        self.nombre = nombre
```

```
        self.edad = edad
        self.mensaje = "No pasa nada"

    def __getstate__(self) -> dict:
        # Copiamos el diccionario para alterar solo la copia, no el objeto original
        nueva = self.__dict__.copy()
        nueva.update({"mensaje": "¡Me están serializando!"})
        # Lo que retornemos es lo que será serializado por pickle
        return nueva
```

```
original = Persona("Juan", 30)
print(f"Mensaje original: {original.mensaje}")
serializado = pickle.dumps(original)
deserializado = pickle.loads(serializado)

print(f"Mensaje original: {original.mensaje}")             # sigue igual
print(f"Mensaje deserializado: {deserializado.mensaje}")
```

Salida

```
Mensaje original: No pasa nada
Mensaje original: No pasa nada
Mensaje deserializado: ¡Me están serializando!
```

¿Qué hace este código? Al llamar `dumps`, `pickle` invoca `__getstate__`, que devuelve una **copia** del diccionario con el atributo `mensaje` cambiado. Gracias a la copia, el objeto `original` no se altera; el objeto que *sale* de la deserialización sí trae el mensaje modificado.

2.5 Personalizar la deserialización: `__setstate__`

De forma análoga se puede personalizar la **deserialización**, implementando `__setstate__`, que se ejecuta cada vez que llamamos a `load` o `loads`. Recibe como argumento el **diccionario con el estado** que fue serializado y debe asignarlo al objeto: `self.__dict__ = estado`. Antes o después de esa asignación podemos hacer otras acciones, como modificar el estado. Si `__setstate__` no existe, simplemente se asigna el estado deserializado al `__dict__` del objeto.



serializar activa `__getstate__`; **deserializar** activa `__setstate__`

```
class Persona:

    def __init__(self, nombre: str, edad: int) -> None:
        self.nombre = nombre
        self.edad = edad
        self.mensaje = "No pasa nada"

    def __getstate__(self) -> dict:
        nueva = self.__dict__.copy()
        print(f"[__getstate__] Serializando a {nueva['nombre']}")
        nueva.update({"mensaje": "¡Me están serializando!"})
        return nueva

    def __setstate__(self, state) -> None:
        print("[__setstate__] Objeto recién deserializado, actualizando su estado")
        state.update({"nombre": f"{state['nombre']} deserializado"})
        self.__dict__ = state
```

```
original = Persona("Juan", 30)
print(f"Nombre original: {original.nombre}")
serializado = pickle.dumps(original)            # se ejecuta __getstate__

print("¡Ejecutar loads -> deserializar!")
deserializado = pickle.loads(serializado)       # se ejecuta __setstate__
print(f"Nombre deserializado: {deserializado.nombre}, y su mensaje: {deserializado.mensaje}")
```

Salida

```
Nombre original: Juan
[__getstate__] Serializando a Juan
¡Ejecutar loads -> deserializar!
[__setstate__] Objeto recién deserializado, actualizando su estado
Nombre deserializado: Juan deserializado, y su mensaje: ¡Me están serializando!
```

¿Qué hace este código? Los `print` permiten *ver* cuándo se ejecuta cada método: `__getstate__` durante `dumps` y `__setstate__` durante `loads`. En la deserialización el nombre pasa de `Juan` a `Juan deserializado`, prueba de que el método se ejecutó y modificó el estado antes de asignarlo.

2.6 ¿Para qué sirve esto en la práctica?

Un caso real: un objeto que tiene un **atributo que depende de las condiciones actuales del programa**, como las conexiones con los usuarios que están conectados *ahora*. Al guardar ese objeto, tiene sentido **eliminar** las conexiones (al cargarlo en otra ejecución del programa, no podremos comunicarnos con los usuarios de la ejecución anterior). Eso se logra con `__getstate__`. Y al volver a cargarlo, hay que **recrear** las conexiones con las condiciones del programa nuevo: eso se hace con `__setstate__`.

```python
class Servidor:
    """Ejemplo ilustrativo de la idea anterior."""

    def __init__(self, nombre: str) -> None:
        self.nombre = nombre
        self.usuarios = []        # datos que sí queremos conservar
        self.conexiones = {}      # sockets abiertos: dependen del programa actual

    def __getstate__(self) -> dict:
        estado = self.__dict__.copy()
        estado["conexiones"] = {}     # no tiene sentido guardar conexiones abiertas
        return estado

    def __setstate__(self, estado: dict) -> None:
        self.__dict__ = estado
        # Aquí podríamos volver a abrir las conexiones que correspondan
```

! Los *sockets* no se pueden serializar

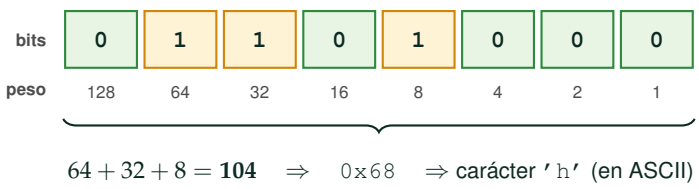
Un objeto `socket` representa un recurso del sistema operativo, así que `pickle` **no puede serializarlo** (lanza un `TypeError`). Si una clase guarda un *socket* como atributo, hay que sacarlo del estado en `__getstate__`, como en el ejemplo anterior.

3 Manejo de bytes (*Input/Output*)

Para trabajar con *sockets* necesitamos entender los **bytes**: no solo almacenamos *strings* como caracteres que vemos en pantalla, también podemos manipular directamente la representación de más bajo nivel de los datos. En esta sección estudiamos los bytes, los arreglos de bytes, la lectura por *chunks* y la conversión de números.

3.1 *Bytes* y *encoding*

Internamente, un computador solo almacena ceros y unos. Un **bit** es un valor 0 o 1, y un **byte** es una secuencia de **8 bits**, por lo que puede tomar $2^8 = 256$ combinaciones distintas: los números del 0 al 255 en binario.



Para almacenar *caracteres*, lo que hacemos es **interpretar** el número guardado en un byte como un carácter. Esa asociación entre números y caracteres se llama **codificación** o ***encoding***.

Encoding

Un *encoding* es la regla que asocia números (bytes) con caracteres. Para interpretar correctamente un conjunto de bytes **hay que conocer el *encoding* con el que fueron codificados**: los mismos bytes pueden significar cosas distintas con reglas distintas.

Una codificación clásica es **ASCII** (de 1963). Define los códigos del 0 al 127: las mayúsculas van del **65 al 90**, las minúsculas del **97 al 122** y los dígitos del **48 al 57**; además hay códigos para puntuación, espacios, saltos de línea y tabulaciones. Carece de caracteres de otros idiomas (como la `ñ` o las tildes), pero es un excelente ejemplo para entender cómo funciona un *encoding*.

Decimal	Hexadecimal	Carácter	Observación
48	0x30	'0'	primer dígito
65	0x41	'A'	primera mayúscula
97	0x61	'a'	primera minúscula
32	0x20	' '	espacio
10	0x0A	'\n'	salto de línea

Python ofrece dos funciones para pasar de código a carácter y viceversa: `chr(codigo)` y `ord(caracter)`. Funcionan con más valores que los de la tabla ASCII.

```
print(chr(105), chr(99), chr(50))    # i c 2
print(ord('B'), ord('y'), ord('t'), ord('e'))    # 66 121 116 101
print(ord('á'), ord('ñ'))             # 225 241  (fuera de ASCII)
```

3.1.1 Hexadecimal

Aunque `ord` muestra el valor decimal, para *representar bytes* es común usar **dos dígitos hexadecimales**. Un dígito hexadecimal es uno de `0 1 2 3 4 5 6 7 8 9 A B C D E F`, que valen de 0 a 15. Como dos dígitos hexadecimales cubren $16 \times 16 = 256$ valores, **un byte se escribe exactamente con dos dígitos hexadecimales**. Se antepone `0x` para indicar que es hexadecimal. Por ejemplo, $\texttt{0xFF} = 15 \cdot 16 + 15 = 255$.

```
for i in [0, 8, 12, 15, 16, 42, 100, 255]:
    print(f"Decimal: {i}, Hexadecimal: {hex(i)}")
# 0 -> 0x0, 8 -> 0x8, 12 -> 0xc, 15 -> 0xf, 16 -> 0x10, 42 -> 0x2a, 100 -> 0x64, 255 -> 0xff
```

3.1.2 Unicode y UTF-8

Existen muchas codificaciones modernas, pero la más extendida hoy es **Unicode**, y será el *encoding* principal del curso. Unicode asigna un número a una enorme cantidad de caracteres de todos los idiomas, y **UTF-8** es la forma más común de convertir esos números en bytes (usa entre 1 y 4 bytes por carácter). Lo importante: **los primeros 128 caracteres de UTF-8 coinciden con ASCII**, así que UTF-8 es *compatible hacia atrás* con ASCII. La mayoría de las aplicaciones modernas lo usan por defecto.

3.2 El objeto `bytes`

En Python, los bytes se representan con el tipo `bytes`: una secuencia **inmutable** (igual que `str`). Se declara anteponiendo una `b` a las comillas.

```
# La notación \x63 indica el valor hexadecimal 63 (1 byte)
caracteres = b"\x63\x6c\x69\x63\x68\xe9"    # c, l, i, c, h, é (en latin-1)
print(caracteres)          # b'clich\xe9'
print(type(caracteres))    # <class 'bytes'>
```

Cada secuencia `\x63` representa **1 byte** escrito en hexadecimal. Los bytes que coinciden con ASCII se muestran como el carácter (`clich`); los demás (`\xe9`) se muestran en hexadecimal. La `b` inicial recuerda que es un `bytes` y no un `str`.

3.2.1 `decode`: de bytes a `str`

Los bytes pueden representar cualquier cosa (desde caracteres hasta píxeles de una imagen). Para interpretarlos como texto hay que decirle a Python **con qué *encoding*** fueron codificados, usando el método `decode`, que retorna un `str` normal.

```
caracteres = b"\x63\x6c\x69\x63\x68\xe9"
print(caracteres.decode("latin-1"))      # cliché

print(b"\x61\x62".decode("ascii"))        # ab    (0x61 y 0x62 son 'a' y 'b')
print(bytes((97, 98)))                    # b'ab' (también se pueden crear desde enteros)
```

El **mismo** byte `\xe9` se interpreta distinto según el *encoding*:

Encoding	Resultado al decodificar `b"clich\xe9"`
`latin-1`	`cliché` (la `é` esperada)
`iso8859-5`	termina en una letra del alfabeto cirílico
`cp437`	termina en una letra griega
`utf-16`	lee los bytes de a pares y entrega texto sin sentido

! Dos errores clásicos con `bytes`

- `b"áb"` produce un **`SyntaxError`**: en un literal `bytes` solo se permiten caracteres **ASCII**. Para tildes o `ñ` hay que codificar un `str` con `encode`.
- Decodificar con el *encoding* equivocado puede fallar: `b"clich\xe9".decode("utf-8")` lanza **`UnicodeDecodeError`**, porque ese byte no es UTF-8 válido por sí solo.

3.2.2 `encode`: de `str` a bytes

Para convertir un `str` en bytes usamos el método `encode` de `str`, indicando el *encoding*.

```
caracteres = "estación"
print(caracteres.encode("UTF-8"))      # b'estaci\xc3\xb3n'   (la ó ocupa 2 bytes)
print(caracteres.encode("latin-1"))    # b'estaci\xf3n'       (la ó ocupa 1 byte)
print(caracteres.encode("CP437"))      # b'estaci\xa2n'

print(caracteres.encode("ascii"))      # UnicodeEncodeError: la 'ó' no existe en ASCII
```

¿Qué hace este código? Un mismo `str` se convierte en bytes **distintos** según el *encoding*: la `ó` ocupa 2 bytes en UTF-8 pero solo 1 en latin-1. Con ASCII falla porque no tiene ese carácter. Para controlar qué hacer en ese caso, `encode` acepta el argumento opcional `errors`:

`errors=`	Qué hace con el carácter imposible	Resultado con `"estación"`
`strict`	lanza un error (valor por defecto)	(excepción)
`replace`	lo reemplaza por `?`	`b'estaci?n'`
`ignore`	lo omite	`b'estacin'`
`xmlcharrefreplace`	lo escribe como entidad XML	`b'estación'`

★ ¿Qué *encoding* uso si no sé cuál?

Usa **UTF-8**: es compatible con ASCII y cubre todos los idiomas. Y recuerda siempre que los objetos `bytes` son **inmutables**.

3.3 El objeto `bytearray`

Un `bytearray` es un **arreglo (lista) de bytes** que, a diferencia de `bytes`, **es mutable**. Se comporta como una lista: se puede indexar, hacer *slicing* y agregar elementos. Se construye a partir de un objeto `bytes`.

	`str`	`bytes`	`bytearray`
Contiene	caracteres	bytes	bytes
¿Mutable?	No	No	**Sí**
Elemento al indexar	un `str`	un `int` (0–255)	un `int` (0–255)
Ejemplo	`"hola"`	`b"hola"`	`bytearray(b"hola")`

```
ba = bytearray(b"holamundo")
print(ba)                   # bytearray(b'holamundo')

print(ba[3:7])              # bytearray(b'amun')     (slicing)
ba[4:6] = b"\x15\xa3"       # reemplazamos 2 bytes
print(ba)                   # bytearray(b'hola\x15\xa3ndo')

ba.extend(b"programa")      # agregamos bytes al final
print(ba)                   # bytearray(b'hola\x15\xa3ndoprograma')
```

¿Qué hace este código? Muestra que un `bytearray` se puede modificar *en el lugar*: reemplazamos un trozo con asignación sobre un *slice* y agregamos más bytes con `extend`. Con un `bytes` normal nada de esto sería posible.

Al **indexar** un byte obtenemos un **entero**, no un carácter. Podemos verlo en binario con `bin`, rellenando con ceros a la izquierda con `zfill`:

```
print(ba[0])                        # 104  (el ASCII de 'h')
print(bin(ba[0]))                   # 0b1101000
print(bin(ba[0])[2:].zfill(8))      # 01101000
```

El `[2:]` descarta el prefijo `0b` (que solo indica "binario") y `zfill(8)` completa con ceros hasta tener los 8 bits del byte (`bin` omite los ceros de la izquierda: por eso salen 7 dígitos). Para pasar un carácter de un byte a entero se usa `ord`, y podemos asignar usando enteros o `ord`, indistintamente:

```
print(ord(b"a"))                # 97

b = bytearray(b'abcdef')
b[3] = ord(b'g')                # 'g' tiene código 103
b[4] = 68                       # 68 es 'D' (equivale a ord(b'D'))
print(b)                        # bytearray(b'abcgDf')
```

Como se comporta como lista, admite las operaciones que ya conocemos:

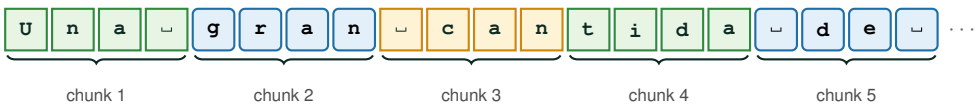
```
mi_bytearray = bytearray()

mi_bytearray.append(2)                  # append SOLO funciona con un int
mi_bytearray.extend(b'\xff')            # extend funciona con bytes...
mi_bytearray.extend(bytearray(b'\xff\x12'))  # ...o con otro bytearray
print(mi_bytearray)                     # bytearray(b'\x02\xff\xff\x12')

print(mi_bytearray[0])          # 2 (indexar entrega el int, no un byte)
print(mi_bytearray[0:2])        # bytearray(b'\x02\xff')  (slicing entrega un bytearray)
print(max(mi_bytearray))        # 255 (el int más grande)
for x in mi_bytearray:
    print(x)                    # x es cada int del bytearray
```

3.4 *Chunks*

Un ***chunk*** es un grupo de bytes. Cuando manejamos una cantidad importante de bytes, no conviene leerlos todos a la vez (pueden no caber en memoria) ni de uno en uno (son demasiadas operaciones y es lento). El **punto medio** es leerlos en *chunks* de cierto tamaño.




```
muchos_bytes = bytearray(b"Una gran cantidad de bytes que quiero leer de a poquito")
TAMANO_CHUNK = 4

for i in range(0, len(muchos_bytes), TAMANO_CHUNK):
    chunk = bytearray(muchos_bytes[i:i + TAMANO_CHUNK])   # aquí obtenemos nuestro chunk
    print(chunk)
# bytearray(b'Una '), bytearray(b'gran'), bytearray(b' can'), bytearray(b'tida'), ...
```

¿Qué hace este código? Recorre el `bytearray` con un `range` que avanza de a `TAMANO_CHUNK` posiciones y, en cada vuelta, extrae el trozo `[i:i+4]`. El último *chunk* puede ser más corto que los demás (aquí, `b'ito'`).

También podemos leer **archivos** por *chunks*, abriéndolos en modo `"rb"`. Si a `file.read()` le pasamos un número, lee a lo más esa cantidad de bytes:

```
from os.path import join
```

```
TAMANO_CHUNK = 4
with open(join('data', 'ejemplo_chunk.bin'), 'rb') as file:
    while chunk := file.read(TAMANO_CHUNK):
        print(chunk)
```

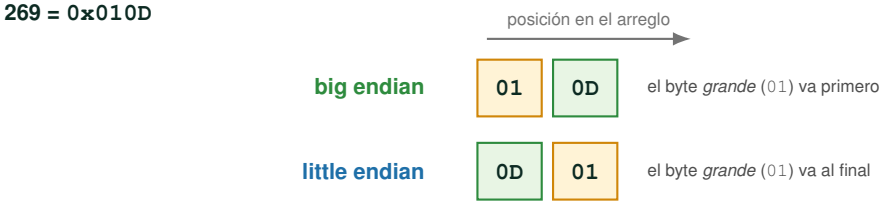
El *walrus operator* `:=`

`chunk := file.read(TAMANO_CHUNK)` hace dos cosas a la vez: **guarda** en `chunk` lo leído y **usa ese valor** como condición del `while`. Cuando se termina el archivo, `read` retorna `b''` (vacío, que es *falso*) y el ciclo termina. Evita escribir la lectura dos veces, pero úsalo con moderación: abusar de él deja el código menos legible.

3.5 Transformar números

Cuando trabajamos con datos binarios o con protocolos de red a menudo necesitamos convertir **enteros a bytes y viceversa**. Un solo byte llega solo hasta 255 (`0xFF`); entonces, ¿cómo se guarda un número más grande? Hay que decidir en **qué orden** se colocan sus bytes. Ese orden se llama `byteorder` y tiene dos variantes:

- ***Big endian*** (`'big'`): el byte *más significativo* (de mayor peso) queda **al inicio**. El 256 es `\x01\x00`.
- ***Little endian*** (`'little'`): el byte más significativo queda **al final**. El 256 es `\x00\x01`.



Número decimal	Big endian	Little endian
269	`\x01\x0d`	`\x0d\x01`
68866	`\x01\x0d\x02`	`\x02\x0d\x01`
1157698	`\x11\xAA\x42`	`\x42\xAA\x11`

3.5.1 `int.to_bytes`: entero a bytes

`numero.to_bytes(length, byteorder)` retorna el entero como un arreglo de bytes. Recibe como mínimo **dos argumentos**: el **largo** del arreglo y el **orden**. Si `length` es mayor al necesario, rellena con ceros (como escribir `001235`, que no cambia el valor).

```
(65).to_bytes(1, 'little')      # b'A'
(256).to_bytes(2, 'big')        # b'\x01\x00'
(256).to_bytes(5, 'big')        # b'\x00\x00\x00\x01\x00'   (relleno a la izquierda)
(256).to_bytes(2, 'little')     # b'\x00\x01'
(256).to_bytes(5, 'little')     # b'\x00\x01\x00\x00\x00'   (relleno a la derecha)

i = 1024
i.to_bytes(4, byteorder='big')  # b'\x00\x00\x04\x00'  (prueba con 'little')
```

! `length` debe alcanzar para el número

Si el largo pedido es insuficiente, Python lanza un `OverflowError`: por ejemplo, `(256).to_bytes(1, 'big')` falla, porque 256 no cabe en un solo byte. Con `length=4` podemos representar hasta $2^{32} - 1$.

3.5.2 `int.from_bytes`: bytes a entero

`int.from_bytes(bytes, byteorder)` hace el camino inverso. Recibe un objeto de tipo `bytes` (o un iterable de bytes) y el orden.

```
int.from_bytes(b'\x00\x00\x04\x00', byteorder='big')      # 1024
int.from_bytes(b'\x00\x00\x04\x00', byteorder='little')   # 262144
int.from_bytes(b'A', byteorder='little')                  # 65
int.from_bytes(b'A', byteorder='big')                     # 65
```

¿Qué hace este código? Los **mismos bytes** `\x00\x00\x04\x00` dan 1024 en *big endian* y 262144 en *little endian*: cambiar el orden puede cambiar radicalmente el valor. En cambio, cuando el arreglo tiene **un solo byte** (como `b'A'`) el orden no importa.

★ ¿Qué orden uso en redes?

Ambos lados de una comunicación deben usar **el mismo** `byteorder`. Por convención, los protocolos de red usan *big endian* (se le llama *network byte order*); por eso en los ejemplos de *sockets* de este resumen siempre usaremos `'big'`.

3.6 No confíes en el `print` de bytes

Un mensaje muy importante: **no confíes en los `print` de bytes**. Al imprimir un byte, Python puede **mostrar visualmente** otro carácter equivalente. Eso no significa que la operación esté mal; es solo la forma que tiene Python de representar ciertos valores.

```
int.from_bytes(b'\x09', byteorder='big')    # 9
(9).to_bytes(1, 'big')                      # b'\t'   <- ¡no es \x09!... pero sí vale 9
int.from_bytes(b'\t', byteorder='big')      # 9

(13).to_bytes(1, 'big')                     # b'\r'   (lo mismo con el 13)
int.from_bytes(b'\r', byteorder='big')      # 13
```

¿Qué hace este código? Convertimos el 9 a bytes y Python lo muestra como `b'\t'` (un tabulador) en vez de `b'\x09'`. Pero al volver a entero con `from_bytes` comprobamos que sigue siendo 9: ambas representaciones son **el mismo byte**.

Valor	Se ve como	Significado
9	`b'\t'`	`tabulador`
10	`b'\n'`	`salto de línea`
13	`b'\r'`	`retorno de carro`

★ Cómo verificar de verdad un byte

En vez de fiarte de cómo se ve, **transfórmalo a entero** con `int.from_bytes` o mira sus valores con `list(mis_bytes)` (por ejemplo, `list(b'\t')` entrega `[9]`). Con eso confirmas que sea el número esperado.

4 Sockets y comunicación TCP

4.1 ¿Qué es un *socket*?

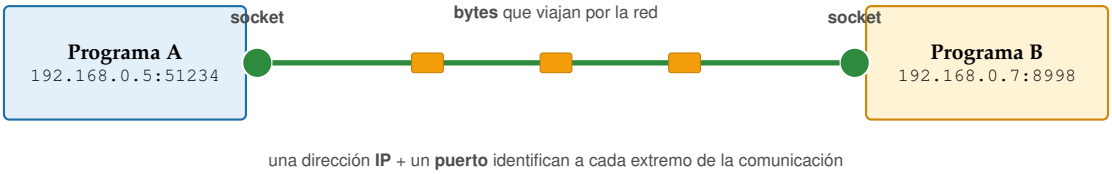
Hasta ahora nuestros programas se han ejecutado de forma aislada. En esta sección veremos cómo implementar programas de **bajo nivel** que se comunican con otro programa, ya sea en el mismo computador o en otro.

■ *Socket*

Un ***socket*** es un objeto del sistema operativo que permite a un programa **enviar y recibir datos** hacia y desde otro programa (que corre en otra máquina, o en la misma máquina pero en otro puerto). Es el *"punto de entrada y salida a la red"* de nuestro código.

★ Analogía telefónica

Un *socket* es como un **teléfono**: para comunicarte necesitas uno, y la otra persona también. Cada teléfono tiene un número (la **dirección IP**) y una *extensión* (el **puerto**) que identifican exactamente a quién estás llamando. Lo que viaja por la línea son **sonidos**; en un *socket*, son **bytes**.



Para obtener un *socket* necesitamos especificar **(1)** qué tipo de dirección IP usaremos (IPv4 o IPv6) y **(2)** qué protocolo de transporte usaremos (TCP o UDP). En Python se usa el módulo `socket`: creamos una instancia de `socket(family, type)`, donde `family` es el tipo de dirección y `type` el protocolo de transporte.

Valor	Parámetro	Significado
`AF_INET`	`family`	direcciones IPv4
`AF_INET6`	`family`	direcciones IPv6
`SOCK_STREAM`	`type`	protocolo **TCP**
`SOCK_DGRAM`	`type`	protocolo **UDP**

```python
import socket

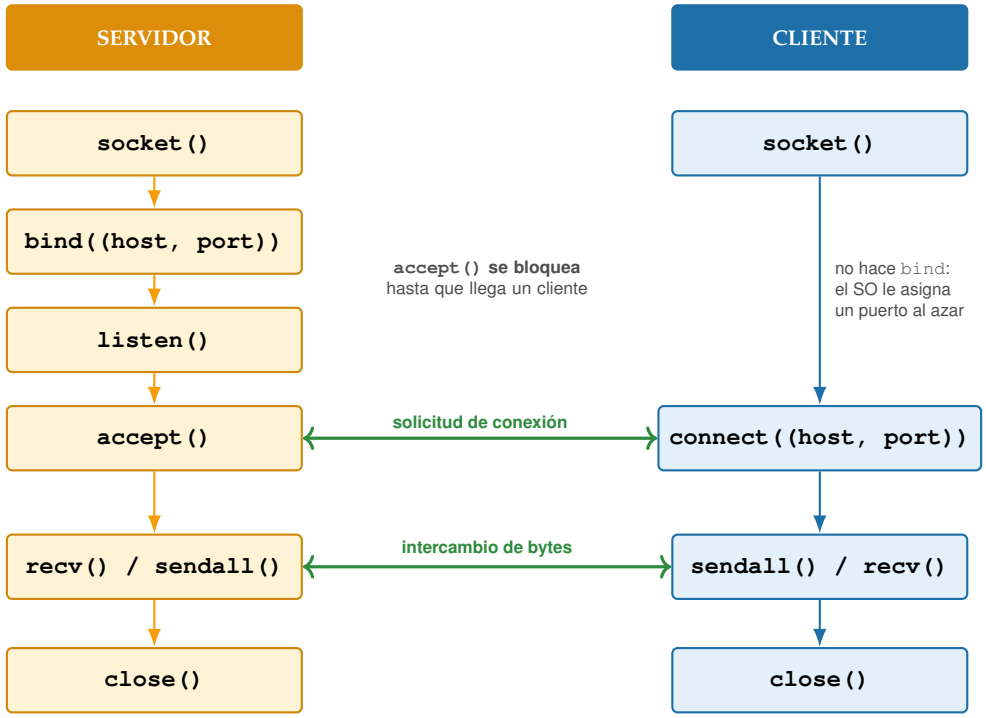
# Un socket TCP con direcciones IPv4
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

★ En este resumen: TCP primero

Por simplicidad, primero implementamos la comunicación con **TCP**. El protocolo **UDP** se revisa como bonus en la Sección 7.

4.2 Los pasos de una comunicación TCP

Para comunicarnos con *sockets* hay que seguir una serie de pasos definidos, tanto en el servidor como en el cliente. Este es el mapa completo; en las siguientes secciones lo recorremos paso a paso.



★ Analogía telefónica

Servidor = una central telefónica: `socket` es *conseguir el teléfono*, `bind` es *asignarle su número*, `listen` es *encenderlo y quedar atento* y `accept` es *contestar* cuando suena. **Cliente** = quien *marca* (`connect`), conversa (`send`/`recv`) y *cuelga* (`close`).

4.3 Cliente TCP

4.3.1 `connect`: conectarse a un servidor

Para abrir una conexión TCP necesitamos **(1)** la dirección IP del *host* destinatario y **(2)** su puerto. Usamos `connect((host, port))`: recibe **una tupla** cuyo primer elemento es la dirección (acorde a lo definido al crear el *socket*) y el segundo, el puerto. Opcionalmente, el primer elemento puede ser un *hostname* como `python.org` o `github.com`; en ese caso, el sistema usa **DNS** para encontrar la IP correspondiente.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('104.18.32.7', 80))          # por dirección IP

# o bien, con un hostname (se resuelve con DNS)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('stackoverflow.com', 80))
```

4.3.2 `send` y `sendall`: enviar bytes

Los *sockets* **solo transmiten bytes**, así que todo lo que enviemos debe estar codificado (con `encode`, `pickle.dumps`, `to_bytes`, etc.). Hay dos métodos para enviar:

Método	Qué hace	Cuidado con...
`send(bytes)`	Intenta enviar el mensaje, pero **puede enviar solo una parte**. **Retorna cuántos bytes envió**.	Es **responsabilidad del programador** volver a llamar `send` con lo que falta, hasta completar el mensaje.
`sendall(bytes)`	Se asegura de enviar **todo** el mensaje (llama a `send` por debajo hasta terminar).	Si hay un error, solo lanza una excepción: **no sabremos cuántos bytes alcanzaron a salir**.

```
# Una solicitud HTTP (HTTP Request), que un servidor web sí entiende
solicitud = 'GET / HTTP/1.1\nHost: stackoverflow.com\n\n'
sock.sendall(solicitud.encode('ascii'))
```

4.3.3 `recv`: recibir bytes

`recv(buffer)` retorna un objeto `bytes`. El parámetro es la cantidad **máxima** de bytes que se leerá; la documentación sugiere una potencia de 2 pequeña, como `4096`.

! `recv(4096)` NO significa "recibir 4096 bytes"...

Es la cantidad **máxima**: el método *saca los bytes que hayan llegado al* socket *en ese instante*. Si al llamar `sock.recv(4096)` habían llegado solo 20 bytes, entrega **solo 20**. Por eso, para mensajes grandes hay que llamar `recv` varias veces (lo veremos en la Sección 5). Además, `recv` **se bloquea** hasta que haya algo que leer, y retorna `b''` (vacío) cuando la otra parte **cerró la conexión**.

```
data = sock.recv(4096)
print(data.decode('ascii'))
```

4.3.4 `close`: cerrar la conexión

El último paso, pero no el menos importante: **hay que cerrar la conexión**. Si no, y nuestro programa continúa, seguirá ocupando recursos que no utiliza.

```
sock.close()
```

4.3.5 Ejemplo completo con manejo de errores

En *networking* es muy probable que ocurran errores, así que conviene usar `try/except/finally`. La clase `ConnectionError` es la **clase base** de varios errores de conexión, por lo que capturarla permite responder a cualquiera de ellos.

```python
import socket

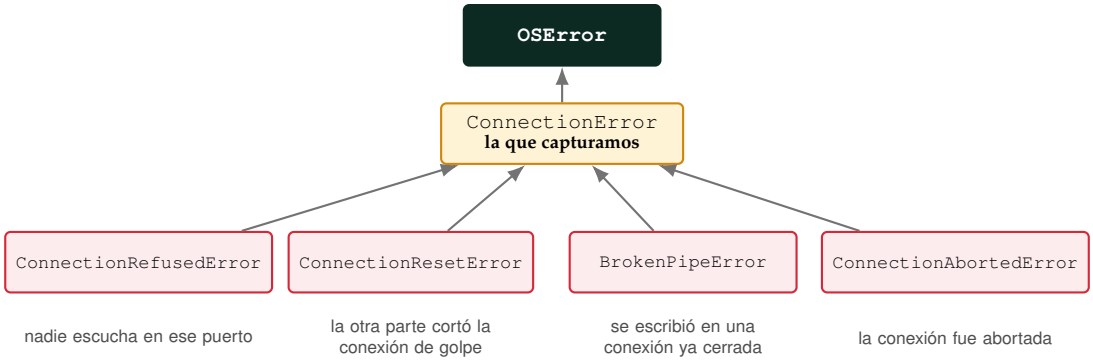
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = "stackoverflow.com"
port = 80

try:
    sock.connect((host, port))
    sock.sendall('GET / HTTP/1.1\nHost: stackoverflow.com\n\n'.encode('utf-8'))
    data = sock.recv(4096)
    print(data.decode('utf-8'))
except ConnectionError as e:
    print("Ocurrió un error.")
finally:
    # ¡No olvidemos cerrar la conexión!
    sock.close()
```

¿Qué hace este código? Se conecta al puerto 80 de `stackoverflow.com`, envía una solicitud HTTP, lee a lo más 4096 bytes de la respuesta y la imprime. El bloque `finally` **siempre** se ejecuta, así que el *socket* se cierra incluso si hubo un error.

La respuesta que recibimos es una **respuesta HTTP** (*HTTP Response*) de tipo **301 (*moved permanently*)**: indica que, para acceder a StackOverflow, hay que usar otra dirección con **HTTPS**. El puerto 80 es un servicio de **texto plano** (sin encriptación); la mayoría de los servidores web modernos usan HTTPS en el puerto 443, que exige operaciones de encriptación sobre el *socket* que complicarían este ejemplo.



4.4 Servidor TCP

★ Los fragmentos de servidor no se ejecutan en Jupyter

En el material del curso, algunos códigos de servidor no están pensados para ejecutarse en un *notebook*. Para probarlos, guarda cada uno en un archivo `.py` y ejecútalo desde una **terminal** (el apunte incluye ejemplos similares en la carpeta `codigo/TCP`). Para el cliente y el servidor necesitas **terminales distintas**.

Un servidor debe **escuchar** las conexiones que le llegan a un puerto específico. Los pasos son:

4.4.1 `bind`: asociar el *socket* a un puerto

`bind((host, port))` *"enlaza"* el *socket* a un *host* y un puerto. El primer elemento es el *hostname* del computador donde corre el servidor (usar `'localhost'` indica *nuestro propio computador*) y el segundo es el puerto donde queremos escuchar.

```python
import socket

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

host = 'localhost'   # el servidor corre en nuestro computador
port = 8998

sock.bind((host, port))
```

El cliente no necesita `bind`

En el cliente **no** hace falta hacer `bind`: el sistema operativo lo hace implícitamente al llamar `connect`, asignándole un **puerto aleatorio**. Solo si el servidor exigiera que los clientes usen un puerto específico tendríamos que enlazar el cliente también.

4.4.2 `listen`: empezar a escuchar

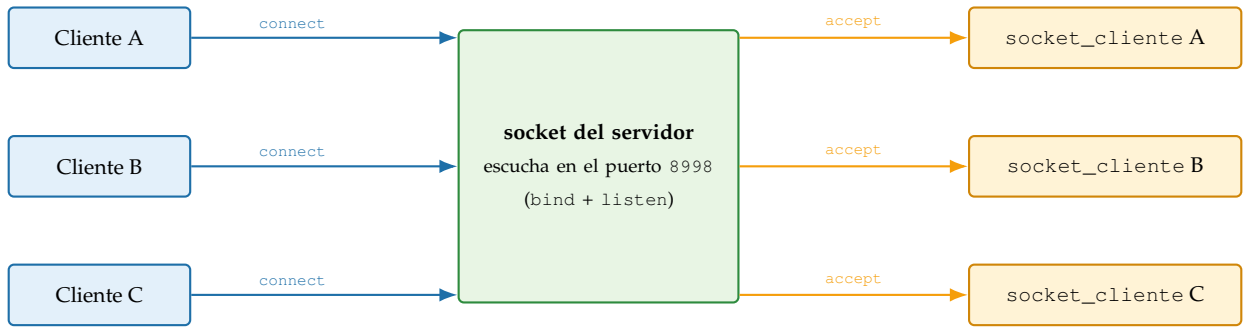
`listen(conexiones_pendientes)` comienza a escuchar. El parámetro indica **cuántas conexiones pendientes (todavía sin aceptar)** pueden esperar en paralelo a que el servidor las acepte; sobre ese máximo, las nuevas conexiones se rechazan.

```
sock.listen(3)   # hasta 3 clientes pueden esperar a ser aceptados al mismo tiempo
```

4.4.3 `accept`: aceptar clientes

`accept()` espera a que llegue una conexión y retorna **dos cosas**: un ***socket* nuevo** para comunicarse con *ese cliente en particular*, y la **dirección del cliente** (una tupla con su IP y su puerto). Es **bloqueante**: si nadie se conecta, el programa queda congelado esperando.

```
while True:
    socket_cliente, address = sock.accept()
    print("Conexión aceptada desde", address)
    socket_cliente.sendall("Gracias por conectarte\n".encode("utf-8"))
    socket_cliente.close()
```



`accept()` crea un socket nuevo por cada cliente; el original sigue escuchando

★ Analogía telefónica: ¿por qué `accept` devuelve un socket nuevo?

El teléfono principal de la central *solo sirve para recibir llamadas*. Cada vez que alguien llama y tú contestas (`accept`), se abre una **línea privada** (el `socket_cliente`) por la que conversan solo ustedes dos, mientras el teléfono principal queda libre para atender a la siguiente llamada. Por eso hablamos con el cliente usando el *socket nuevo*, **no** el original.

Cuando queramos terminar el servidor, cerramos el *socket* que aceptaba conexiones:

```
sock.close()
```

! Error común: "Address already in use"

Si intentas levantar un servidor en un puerto que **ya está ocupado** (por otra copia de tu programa, o porque lo acabas de cerrar), `bind` lanza un **`OSError`**. Revisa que no haya otro servidor corriendo, espera unos instantes o usa otro puerto. Y recuerda: **dos servidores no pueden escuchar en el mismo puerto** del mismo computador.

4.5 Ejemplo base: servidor y cliente

Armemos el ejercicio completo. Guarda cada código en un archivo (por ejemplo `servidor.py` y `cliente.py`), **abre varias terminales**, ejecuta primero el servidor y luego el cliente en las otras. Este servidor recibe a un cliente, le manda un mensaje y cierra la conexión; la particularidad es que **se cierra después de aceptar cinco clientes**.

```python
## Código servidor
import socket

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

host = 'localhost'
port = 8998

sock.bind((host, port))
sock.listen()
print("[Servidor] Aceptando conexiones en el puerto", port)

counter = 0
while counter < 5:
    try:
        socket_cliente, address = sock.accept()
        print("[Servidor] Conexión aceptada desde", address)
        socket_cliente.sendall("Gracias por conectarte\n".encode("utf-8"))
        socket_cliente.close()
        counter += 1
    except ConnectionError:
        print("[Servidor] Ocurrió un error.")

print("[Servidor] Se han recibido 5 clientes, por lo que se cerrará la conexión")
sock.close()
```

El cliente intenta conectarse, recibe el mensaje y cierra la conexión (también si ocurre un error):

```python
## Código cliente
import socket

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

host = 'localhost'
port = 8998

try:
    sock.connect((host, port))
    data = sock.recv(4096)
    print(data.decode("utf-8"))
except ConnectionError as e:
    print("[Cliente] Ocurrió un error")
finally:
    sock.close()
```

¿Qué hace este código? El servidor hace `bind` y `listen` en el puerto `8998` y entra a un ciclo: cada `accept` entrega un *socket* para un cliente, le envía el saludo y lo cierra, aumentando el contador. Al llegar a 5 clientes cierra el *socket* del servidor. El cliente, por su lado, se conecta, lee el saludo y termina. Si ejecutas el cliente **sin** que el servidor esté corriendo, obtendrás un **`ConnectionRefusedError`**, que el `except` captura.

Salida esperada en cada terminal

```
Servidor:  [Servidor] Aceptando conexiones en el puerto 8998
           [Servidor] Conexión aceptada desde ('127.0.0.1', 52814)     (puerto del cliente:
    aleatorio)
Cliente:   Gracias por conectarte
```

5 Envío de datos por la red

Veamos ahora variaciones sobre qué enviamos y cómo: datos serializados con JSON o `pickle`, y mensajes de tamaño arbitrario.

5.1 Envío de datos con JSON

Para probarlo, usa **dos terminales**: en una ejecuta el servidor y en la otra el cliente. El primer ejemplo es un ***echo server***: un servidor que recibe datos y los envía de vuelta tal cual.

```python
# Echo server: recibe datos y los envía de vuelta
import socket
```

```python
host = 'localhost'   # usar nuestro computador como host
port = 12345

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind((host, port))
print("Listo para recibir. Ahora voy a esperar que me hablen...")
sock.listen()

sock_cliente, (host_cliente, puerto_cliente) = sock.accept()
print("Conexión desde", host_cliente, puerto_cliente)

while True:
    data = sock_cliente.recv(4096)
    print(f"Recibí estos bytes: {data}")
    if not data:
        break
    print("Los enviaré de vuelta")
    sock_cliente.sendall(data)

sock_cliente.close()
sock.close()
```

¿Qué hace este código? Acepta a **un** cliente y entra a un ciclo: lee bytes y los reenvía. La línea `if not data: break` es clave: cuando el cliente cierra la conexión, `recv` retorna `b''` (vacío, *falso*) y el servidor sale del ciclo. Fíjate que **aceptamos con una tupla**: `(host_cliente, puerto_cliente)` se desempaqueta directamente.

El cliente serializa un diccionario con JSON, lo codifica a bytes, lo envía y decodifica la respuesta:

```python
# Cliente que envía datos en formato JSON
import json
import socket

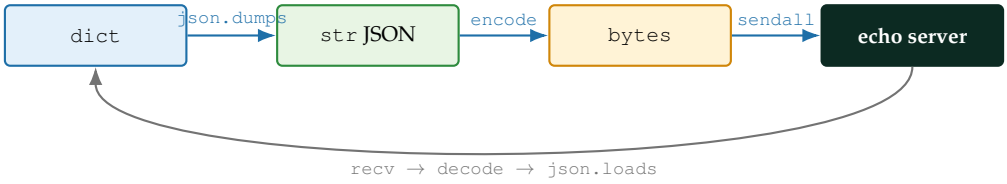
server_host = ''    # aquí va la IP del servidor; vacío = el mismo computador
server_port = 12345

data = {1: "Hola", 2: "Chao"}
mensaje = json.dumps(data)

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_host, server_port))

# Debemos enviar bytes, así que codificamos el mensaje
sock.sendall(mensaje.encode('utf-8'))

# Decodificamos los bytes de la respuesta y los deserializamos con JSON
data = json.loads(sock.recv(4096).decode('utf-8'))
print(data)
input("ENTER para terminar.")
sock.close()
```



Cuando el cliente termina, la conexión se cierra, el servidor deja de recibir datos y (como está construido) también termina. Si quisiéramos que quedara disponible para otras conexiones, tendríamos que volver a llamar `accept`, como en el ciclo del servidor del ejemplo base.

5.2 Envío de datos con `pickle`

De la misma forma podemos enviar **cualquier objeto de Python** serializado con `pickle`. Este cliente se conecta al *mismo echo server* anterior: el servidor no se preocupa de qué tipo de datos recibe, solo recibe bytes y los devuelve.

```python
# Cliente que envía datos en formato pickle
import pickle
import socket
```

```python
server_host = ''
port = 12345


class Persona:
    def __init__(self, nombre, correo):
        self.nombre = nombre
        self.correo = correo


persona = Persona("Juan Pérez", "jp@ejemplo.com")
mensaje = pickle.dumps(persona)

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((server_host, port))
sock.sendall(mensaje)

data = pickle.loads(sock.recv(4096))
print(f"He recibido bytes de vuelta y los deserialicé a {type(data)}")
print(data.nombre)
sock.close()
```

¿Qué hace este código? Serializa una instancia de `Persona`, la envía, recibe los mismos bytes de vuelta y los deserializa: obtenemos nuevamente un objeto `Persona`. A diferencia de JSON, **no necesitamos `encode`/`decode`**, porque `pickle.dumps` ya entrega bytes. Podríamos modificar el servidor para que haga algo con los datos que recibe (por ejemplo, verificar las credenciales de un inicio de sesión) y envíe el resultado de vuelta.

! Dos advertencias para un servidor real

- **Seguridad**: aquí el *echo server* nunca deserializa nada, y solo el cliente hace `pickle.loads` de lo que él mismo envió. Un servidor que haga `pickle.loads` de lo que envían **clientes desconocidos** puede ser atacado (Sección 2). Para esos casos usa JSON.
- **Tamaño**: `recv(4096)` lee *a lo más* 4096 bytes. Si el objeto serializado es más grande, `pickle.loads` recibiría un mensaje **incompleto** y fallaría. La solución es el protocolo de la Sección 5.3.

★ Pruébalo con dos computadores

Se recomienda fuertemente conectar **dos computadores** y probar el envío hacia y desde ambos. En los **clientes** debes escribir la **dirección IP del servidor** como *host*; el servidor no necesita modificaciones. La dirección IP se consulta en la configuración de red de cada sistema operativo. Ojo: sitios como `whatismyipaddress.com` muestran tu dirección *pública*; si el otro computador está en tu misma casa (misma subred), necesitas la dirección *interna* de tu red.

5.3 Envío de muchos datos

Supongamos que queremos transmitir un archivo relativamente grande entre dos computadores. Parece simple: leer el archivo en bytes y enviarlo con un *socket*. Implementemos un receptor y un emisor.

```python
# Receptor del archivo (versión ingenua)
import socket

host = 'localhost'
port = 12345

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind((host, port))
sock.listen()
print("Escuchando...")

sock_cliente, (host_cliente, port_cliente) = sock.accept()
print("Conexión entrante aceptada.")

datos = sock_cliente.recv(4096)
with open('files/recibido.bin', 'wb') as binfile:
    binfile.write(datos)

print("¡Archivo recibido!")
sock_cliente.sendall("Gracias.".encode('utf-8'))

sock_cliente.close()
sock.close()
```

```python
# Emisor del archivo (versión ingenua)
import socket

host_receptor = 'localhost'   # en otro computador: su dirección IP
puerto_receptor = 12345

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host_receptor, puerto_receptor))
print("Conexión establecida.")

with open('files/enviar.bin', 'rb') as binfile:
    datos = binfile.read()
    sock.sendall(datos)

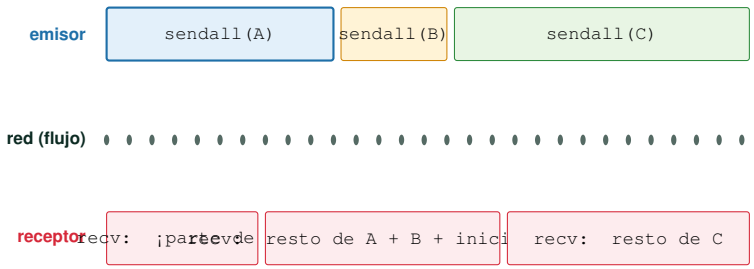
print("¡Archivo enviado!")
print("Respuesta:", sock.recv(4096).decode('utf-8'))
sock.close()
```

Todo parece ir bien… pero, si comparamos los archivos, **el recibido es mucho más pequeño**. ¡Leímos muy pocos bytes! Podríamos subir el valor de `recv(buffer)`, pero seguiríamos teniendo problemas:

- ¿Cómo saber **de antemano** el tamaño de lo que voy a recibir?
- E incluso sabiéndolo, ¿qué ocurre si hay que recibir **megabytes o gigabytes**?

■ TCP es un flujo de bytes, no una lista de mensajes

TCP entrega los bytes **en orden**, pero **no conserva los límites entre mensajes**. Dos `sendall` seguidos pueden llegar *pegados* en un solo `recv`, y un solo mensaje grande puede llegar *partido* en varios. El receptor **no puede adivinar** dónde termina cada mensaje: necesitamos un acuerdo entre ambos lados.



5.3.1 La solución: un protocolo con encabezado de largo

La solución tiene **dos partes**. Primero, definir un **protocolo** (un esquema de comunicación acordado por ambos lados). Por ahora, solo necesitamos una forma de indicar el **tamaño** del mensaje: enviar, *antes* del contenido, un número de **4 bytes** con su largo. Con 4 bytes de encabezado podemos enviar mensajes de hasta 2^{32} bytes (4 GiB), una cantidad razonable para la mayoría de los casos.



La segunda parte: ya sabiendo cuántos bytes hay que leer, **leer por *chunks*** hasta completarlos. Así, volvemos a implementar el emisor y el receptor.

```python
# Receptor mejorado del archivo
import socket

host = 'localhost'
puerto = 12345

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind((host, puerto))
sock.listen()
print('Escuchando...')
```

```python
sock_cliente, (host_cliente, puerto_cliente) = sock.accept()
print("Conexión entrante aceptada.")

# Leemos primero los 4 bytes del tamaño; los transformamos en un int
largo_archivo = int.from_bytes(sock_cliente.recv(4), byteorder='big')
datos = bytearray()
bytes_leidos = 0
print(f"OK. Ahora sé que debo recibir {largo_archivo} bytes")

# Ahora leemos el archivo por chunks, de máximo 4096 bytes
while len(datos) < largo_archivo:
    # El último recv será probablemente más chico que 4096
    bytes_leer = min(4096, largo_archivo - len(datos))
    datos_recibidos = sock_cliente.recv(bytes_leer)
    bytes_leidos += len(datos_recibidos)
    print(f"He recibido {len(datos_recibidos)} bytes en el último recv. Van {bytes_leidos} en
    total.")
    datos.extend(datos_recibidos)

print(f"¡Listo! He recibido {len(datos)} bytes")

with open('files/recibido.bin', 'wb') as binfile:
    binfile.write(datos)

print("¡Archivo guardado!")
sock_cliente.sendall("Gracias.".encode('utf-8'))

sock_cliente.close()
sock.close()
```

¿Qué hace este código? Primero lee el encabezado (4 bytes → `int`) para saber el `largo_archivo`. Luego, mientras no haya completado ese largo, llama a `recv` pidiendo `min(4096, lo que falta)`: así **nunca se pasa** del final del mensaje (que podría pertenecer al siguiente). Y como `recv` **no asegura entregar todo lo pedido**, el progreso se mide con `len(datos_recibidos)`, **no** con el número que le pasamos como parámetro.

```python
# Emisor mejorado del archivo
import socket

host_receptor = 'localhost'
puerto_receptor = 12345

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host_receptor, puerto_receptor))
print("Conexión establecida.")

with open('files/enviar.bin', 'rb') as binfile:
    datos = binfile.read()
    largo_archivo = len(datos)
    print(f"Voy a enviar {largo_archivo} bytes")

    # 1. Enviar el tamaño del archivo (4 bytes)
    sock.sendall(largo_archivo.to_bytes(4, byteorder='big'))

    # 2. Enviar el archivo, con send() en un ciclo para asegurar que se envíe todo
    bytes_enviados = 0
    while bytes_enviados < largo_archivo:
        # Enviamos desde el último punto efectivamente enviado
        bytes_enviados += sock.send(datos[bytes_enviados:])

print("¡Archivo enviado!")
print("Respuesta:", sock.recv(4096).decode('utf-8'))
sock.close()
```

¿Qué hace este código? Envía primero el largo en 4 bytes y luego el contenido. Para el contenido usa `send` **dentro de un ciclo**: cada llamada retorna cuántos bytes salieron y la siguiente parte desde ahí (`datos[bytes_enviados:]`), hasta completar `largo_archivo`. Es el uso manual de `send` que mencionamos antes (equivale a lo que hace `sendall` internamente).

★ Versión reutilizable (extra): tres funciones que puedes copiar

Esta versión no está en el apunte, pero resume la idea y además corrige un detalle: si el otro lado cierra la conexión a mitad de camino, `recv` retorna `b''` y un ciclo ingenuo **no terminaría nunca**. Aquí lo detectamos.

También lee el encabezado de forma segura (un `recv(4)` podría, en teoría, entregar menos de 4 bytes).

```python
def recibir_exacto(sock, n: int) -> bytes:
    datos = bytearray()
    while len(datos) < n:
        parte = sock.recv(min(4096, n - len(datos)))
        if not parte:    # b'' -> el otro lado cerró la conexión
            raise ConnectionError("Conexión cerrada antes de recibir todo el mensaje")
        datos.extend(parte)
    return bytes(datos)

def enviar_mensaje(sock, contenido: bytes) -> None:
    sock.sendall(len(contenido).to_bytes(4, byteorder="big") + contenido)

def recibir_mensaje(sock) -> bytes:
    largo = int.from_bytes(recibir_exacto(sock, 4), byteorder="big")
    return recibir_exacto(sock, largo)
```

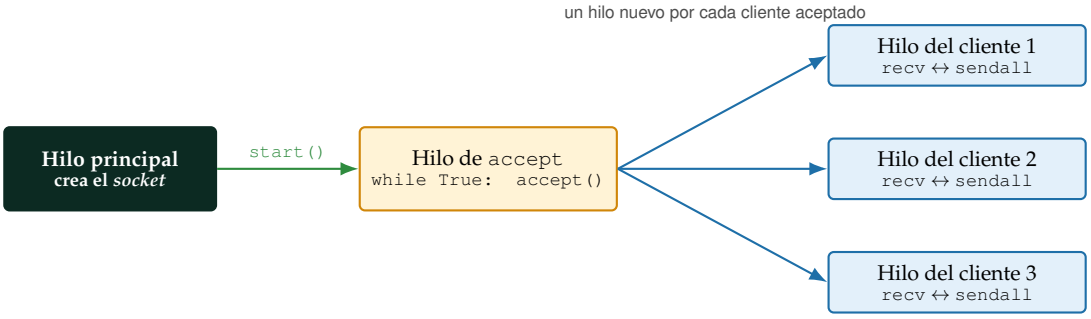
6 Arquitecturas con múltiples conexiones

En la práctica tendremos que manejar **varios usuarios a la vez** y enviar mensajes de tamaño arbitrario. En esta sección armamos tres arquitecturas: un servidor con muchos clientes, un cliente conectado a varios servidores, y comunicación *peer-to-peer* (P2P).

6.1 Servidor con múltiples clientes concurrentes

Los métodos `accept()` y `recv()` son **bloqueantes**: el hilo (*thread*) que los ejecuta queda detenido hasta que llegue una conexión o datos. Si todo ocurriera en un solo hilo, mientras esperamos a un cliente no podríamos atender a otro. La solución es usar ***threads***:

- Un *thread* dedicado a **aceptar nuevos clientes**.
- Y, cada vez que se acepta un cliente, **un *thread* nuevo** que escucha y responde a ese cliente.



Además, implementamos lo aprendido sobre mensajes de tamaño arbitrario: **los primeros 4 bytes de cada mensaje indican su largo**. Definimos primero la clase `Client`:

```python
import socket
import threading


class Client:
    """Maneja toda la comunicación desde el lado del cliente."""

    def __init__(self, port: int, host: str) -> None:
        print("Inicializando cliente...")
        self.host = host
        self.port = port
        self.socket_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        try:
            self.connect_to_server()
            self.listen()
            self.repl()
        except ConnectionError:
```

```
            print("Conexión terminada.")
            self.socket_client.close()
            exit()

    def connect_to_server(self) -> None:
        """Crea la conexión al servidor."""
        self.socket_client.connect((self.host, self.port))
        print("Cliente conectado exitosamente al servidor.")

    def listen(self) -> None:
        """Inicia el thread que escucha los mensajes del servidor, para que
        este pueda hablarnos sin que nosotros partamos la conversación."""
        thread = threading.Thread(target=self.listen_thread, daemon=True)
        thread.start()

    def send(self, msg: str) -> None:
        """Envía 4 bytes con el largo y luego el mensaje."""
        msg_bytes = msg.encode()
        msg_length = len(msg_bytes).to_bytes(4, byteorder='big')
        self.socket_client.sendall(msg_length + msg_bytes)

    def listen_thread(self) -> None:
        while True:
            response_bytes_length = self.socket_client.recv(4)
            response_length = int.from_bytes(response_bytes_length, byteorder='big')
            response = bytearray()

            # Recibimos hasta alcanzar el largo indicado en los primeros 4 bytes
            while len(response) < response_length:
                read_length = min(4096, response_length - len(response))
                response.extend(self.socket_client.recv(read_length))

            print(f"{response.decode()}\n>>> ", end='')

    def repl(self) -> None:
        """Lee mensajes desde la terminal y los pasa a self.send()."""
        print("------ Consola ------\n>>> ", end='')
        while True:
            msg = input()
            self.send(msg)


if __name__ == "__main__":
    port = 8080
    host = 'localhost'
    client = Client(port, host)
```

¿Qué hace este código? El cliente usa **dos hilos**: el principal ejecuta `repl` (lee lo que escribe el usuario con `input`, que también bloquea) y un hilo aparte (`listen_thread`) queda esperando mensajes del servidor. Así la comunicación es **asíncrona**: el servidor puede escribirnos en cualquier momento. Con `daemon=True`, ese hilo termina automáticamente cuando termina el programa principal.

Ahora la clase `Server`, que controla la lógica del lado del servidor:

```
import socket
import threading


class Server:
    def __init__(self, port: int, host: str) -> None:
        print("Inicializando servidor...")
        self.host = host
        self.port = port
        self.socket_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.bind_and_listen()
        self.accept_connections()

    def bind_and_listen(self) -> None:
        """Enlaza el socket al host y puerto, y queda esperando conexiones."""
        self.socket_server.bind((self.host, self.port))
        self.socket_server.listen()
        print(f"Servidor escuchando en {self.host}:{self.port}...")

    def accept_connections(self) -> None:
        """Inicia el thread que acepta clientes, para poder seguir haciendo
```

```
        otras cosas en el servidor sin dejar de aceptar."""
        thread = threading.Thread(target=self.accept_connections_thread)
        thread.start()

    def accept_connections_thread(self) -> None:
        """Por cada cliente aceptado, inicia un thread que lo atiende."""
        print("Servidor aceptando conexiones...")
        while True:
            client_socket, _ = self.socket_server.accept()
            listening_client_thread = threading.Thread(
                target=self.listen_client_thread,
                args=(client_socket, ),
                daemon=True)
            listening_client_thread.start()

    @staticmethod
    def send(value: object, sock: socket.socket) -> None:
        """Protocolo: los primeros 4 bytes indican el largo del mensaje."""
        stringified_value = str(value)
        msg_bytes = stringified_value.encode()
        msg_length = len(msg_bytes).to_bytes(4, byteorder='big')
        sock.sendall(msg_length + msg_bytes)

    def listen_client_thread(self, client_socket: socket.socket) -> None:
        """Thread que escucha a UN cliente en particular."""
        print(f"{client_socket.getpeername()} Nuevo cliente conectado al servidor")

        while True:
            response_bytes_length = client_socket.recv(4)
            response_length = int.from_bytes(response_bytes_length, byteorder='big')
            response = bytearray()

            while len(response) < response_length:
                read_length = min(4096, response_length - len(response))
                response.extend(client_socket.recv(read_length))

            received = response.decode()

            if received != "":
                # handle_command contiene la lógica asociada a cada mensaje
                # y retorna la respuesta que se enviará al cliente.
                response = self.handle_command(received, client_socket)
                self.send(response, client_socket)

    def handle_command(self, received: str, client_socket: socket.socket) -> str:
        print(f"{client_socket.getpeername()} Comando recibido:", received)
        return "Acción asociada a " + received


if __name__ == "__main__":
    port = 8080
    host = 'localhost'
    server = Server(port, host)
```

¿Qué hace este código? Se divide en piezas con una responsabilidad cada una:

Método	Responsabilidad
`bind_and_listen`	Crea el enlace y deja el servidor escuchando.
`accept_connections`	Lanza un hilo que repite `accept()` y crea un hilo por cliente.
`listen_client_thread`	Un hilo *por cliente*: lee el largo, lee el mensaje por *chunks* y llama a `handle_command`.
`send`	Aplica el protocolo (4 bytes de largo + contenido) al responder.
`handle_command`	Aquí va la *lógica*: recibe el texto y retorna la respuesta.

! Detalle importante: ¿qué pasa si un cliente se desconecta?

En este ejemplo, cuando un cliente cierra la conexión, `recv(4)` retorna `b''`: el largo sería 0 y `received` quedaría vacío, de modo que el ciclo `while True` del hilo **giraría para siempre sin hacer nada útil**. El ejemplo del apunte es didáctico; en código real hay que **detectar `b''`**, cerrar ese *socket* y terminar el hilo de ese

cliente (por ejemplo, con la función `recibir_exacto` de la Sección 5.3).

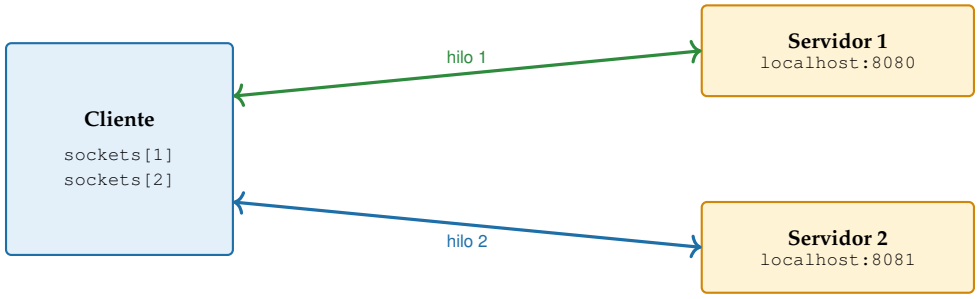
★ Separa la comunicación de la lógica

Es recomendable que las clases `Client` y `Server` se dediquen **solo a la comunicación** (enviar y recibir). Si les vamos agregando la lógica del programa, el código queda **muy difícil de leer y mantener**. Implementa la lógica en **otras clases (*backend*)** y deja a `Client`/`Server` únicamente como el canal de comunicación.



6.2 Cliente conectado a más de un servidor

Aunque es menos común que un servidor con muchos clientes, a veces un cliente necesita conectarse a **más de un servidor** al mismo tiempo (cada servidor pertenece a un servicio distinto, o encargan funcionalidades distintas de un mismo producto). La forma de trabajar es similar: **un *thread* por conexión**. La diferencia es que **no podemos usar el mismo *socket*** para ambas conexiones: el cliente crea **un *socket* por servidor** y debe decidir a cuál enviar cada mensaje, y cómo procesar lo recibido según qué servidor lo envió.



cada servidor tiene **su propio puerto**: así el cliente los distingue aunque tengan la misma IP

Para ejecutarlo, crea los archivos `servidor_1.py`, `servidor_2.py` y `cliente.py`. Los dos servidores son **el mismo código** (la clase `Server` de arriba, simplificada para responder con `sendall` directo), diferenciados solo por su **puerto** y por los `print`:

```python
# servidor_1.py  (servidor_2.py es igual, pero con port = 8081 y textos "servidor 2")
if __name__ == "__main__":
    # Cada servidor debe definir su propio puerto distinto.
    port = 8080
    host = 'localhost'
    server = Server(port, host)

# Dentro de la clase Server del servidor 1:
    def handle_command(self, received: str, client_socket: socket.socket) -> str:
        print("Comando recibido por el servidor 1:", received)
        return "Acción del servidor 1 asociada a " + received
```

El cliente, en cambio, genera **dos *sockets*** y **dos hilos** independientes entre sí:

```python
import socket
import threading
import time


class Client:
    def __init__(self, datos_servidor_1: dict, datos_servidor_2: dict) -> None:
        print("Inicializando cliente...")
        self.datos_servidor_1 = datos_servidor_1
        self.datos_servidor_2 = datos_servidor_2
        # Un socket por servidor (en un dict, para escalar mejor)
        self.sockets = {
            1: socket.socket(socket.AF_INET, socket.SOCK_STREAM),
            2: socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        }
        try:
            self.connect_to_server()
            self.listen()
```

```
            self.repl()
        except ConnectionError:
            print("Conexión terminada.")
            self.disconnect_servers()
            exit()

    def connect_to_server(self) -> None:
        """Repetimos el connect para cada servidor."""
        self.sockets[1].connect(
            (self.datos_servidor_1['host'], self.datos_servidor_1['port']))
        self.sockets[2].connect(
            (self.datos_servidor_2['host'], self.datos_servidor_2['port']))
        print("Cliente conectado exitosamente a ambos servidores.")

    def disconnect_servers(self) -> None:
        for sock in self.sockets.values():
            sock.close()

    def listen(self) -> None:
        """Un thread por conexión, para que los servidores no se bloqueen entre sí."""
        thread_1 = threading.Thread(target=self.listen_thread, args=(1,), daemon=True)
        thread_2 = threading.Thread(target=self.listen_thread, args=(2,), daemon=True)
        thread_1.start()
        thread_2.start()

    def send(self, server: int, msg: str) -> None:
        """Elegimos a qué servidor mandar el mensaje usando su socket."""
        self.sockets[server].sendall(msg.encode())

    def listen_thread(self, servidor: int) -> None:
        while True:
            response = self.sockets[servidor].recv(4096)
            print(f"El siguiente mensaje fue enviado por el servidor {servidor}:")
            print(f"{response.decode()}\n>>> ", end='')

    def repl(self) -> None:
        print("------ Consola ------\n>>> ", end='')
        while True:
            server = input("Selecciona a que servidor deseas escribir (1, 2): ")
            msg = input("Escribe tu mensaje: ")
            self.send(int(server), msg)
            time.sleep(1)    # solo para que los textos de la consola no se sobrepongan


if __name__ == "__main__":
    datos_servidor_1 = {'port': 8080, 'host': 'localhost'}
    datos_servidor_2 = {'port': 8081, 'host': 'localhost'}
    client = Client(datos_servidor_1, datos_servidor_2)
```

¿Qué hace este código? Guarda un *socket* por servidor en el diccionario `self.sockets`, se conecta a ambos y lanza `listen_thread` dos veces, indicando con el argumento `servidor` *a cuál* escuchar; así, cada mensaje recibido se identifica con el servidor que lo envió. Para enviar, `send` elige el *socket* correspondiente.

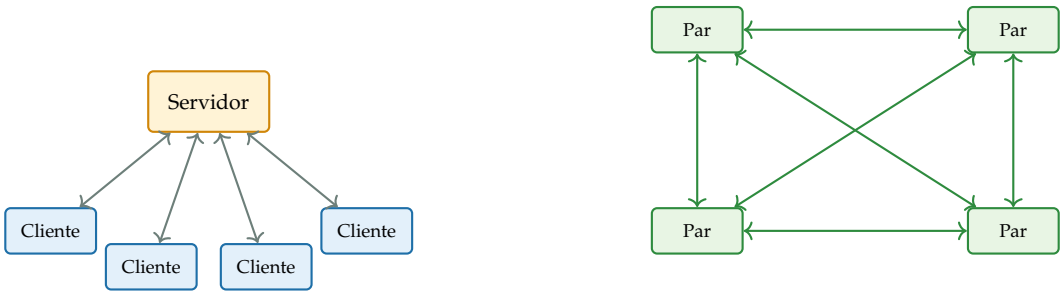
★ Orden de ejecución y simplificación

Ejecuta primero los dos servidores y luego el cliente. Además, estos ejemplos usan `recv(4096)` directo, **sin** el encabezado de largo, para concentrarse en la idea de las múltiples conexiones; en un programa real combinarías ambas técnicas.

6.3 Interacción entre clientes *Peer-to-Peer* (P2P)

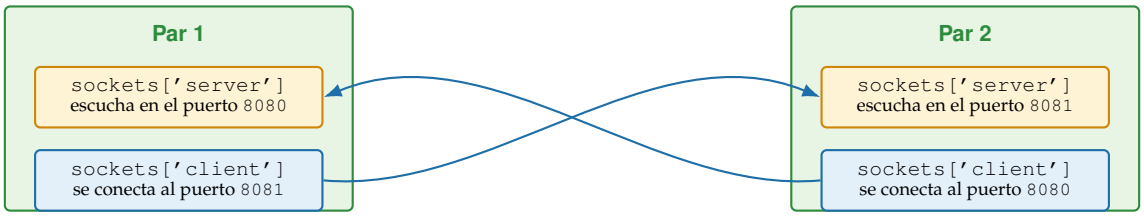
Hasta ahora usamos arquitecturas **centralizadas**: hay una diferencia clara entre el **servidor** (procesa solicitudes, responde y administra la red) y los **clientes** (piden o entregan información cuando la necesitan). Pero no es la única arquitectura. En una red ***Peer-to-Peer*** desaparece el servidor único: hay solo **clientes que se comunican entre sí**, repartiéndose la carga de trabajo de la red. Un ejemplo conocido son las redes de transferencia de archivos.

Centralizada ***Peer-to-Peer***



todos hablan con el servidor — todos hablan con todos; nadie es "el servidor"

Para implementarlo con *sockets* ya no tendremos un archivo de servidor y otro de cliente: **ambos son clientes** equivalentes en jerarquía. Cada programa es, a la vez, un **servidor** (crea un *socket* en cierto puerto para aceptar conexiones de otros) y un **cliente** (se conecta a los *sockets* de otros). Por eso nuestras clases tienen **métodos de servidor y de cliente**, y usamos *threads* para que no se bloqueen entre sí.



dos conexiones cruzadas: cada par es **servidor** del otro y **cliente** del otro

Por simplicidad, el ejemplo tiene dos archivos, `cliente_1.py` y `cliente_2.py`, que **deben ejecutarse en ese orden**: así evitamos intentar conectarnos a un *socket* de un programa que aún no está corriendo. Veamos el `cliente_2.py` (versión resumida del apunte, solo se omiten los comentarios):

```python
import socket
import threading
import time


class Client_2:
    def __init__(self, host: str, port: int, datos_otro_cliente: dict) -> None:
        print("Inicializando cliente P2P (2)...")
        self.host = host
        self.port = port
        self.datos_otro_cliente = datos_otro_cliente
        # Un socket para escuchar como servidor y otro para conectarnos como cliente
        self.sockets = {
            'server': socket.socket(socket.AF_INET, socket.SOCK_STREAM),
            'client': socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        }

        # 1) Abrimos nuestro socket como servidor
        self.bind_and_listen()
        self.accept_connections()

        # 2) Asumimos que el otro par ya está arriba: nos conectamos como clientes
        try:
            self.connect_to_server()
            self.listen()
            self.repl()
        except ConnectionError:
            print("Conexión terminada.")
            self.sockets['client'].close()
            exit()

    # ---------------- MÉTODOS DE SERVIDOR ----------------
    def bind_and_listen(self) -> None:
        self.sockets['server'].bind((self.host, self.port))
```

```
        self.sockets['server'].listen()
        print(f"Cliente P2P (2) escuchando en {self.host}:{self.port}...")

    def accept_connections(self) -> None:
        thread = threading.Thread(target=self.accept_connections_thread)
        thread.start()

    def accept_connections_thread(self) -> None:
        print("Cliente P2P (2) aceptando conexiones...")
        while True:
            client_socket, _ = self.sockets['server'].accept()
            listening_client_thread = threading.Thread(
                target=self.listen_client_thread,
                args=(client_socket, ),
                daemon=True)
            listening_client_thread.start()

    def listen_client_thread(self, client_socket: socket.socket) -> None:
        print("Cliente P2P (2) recibió a un nuevo cliente...")
        while True:
            received = client_socket.recv(4096).decode()
            if received != "":
                response = self.handle_command(received, client_socket)
                self.answer_client(response, client_socket)

    def handle_command(self, received: str, client_socket: socket.socket) -> str:
        print("Comando recibido por el Cliente P2P (2):", received)
        return "Acción del Cliente P2P (2) asociada a " + received

    @staticmethod
    def answer_client(value: object, sock: socket.socket) -> None:
        sock.sendall(str(value).encode())   # respondemos como un servidor

    # ---------------- MÉTODOS DE CLIENTE ----------------
    def connect_to_server(self) -> None:
        self.sockets['client'].connect(
            (self.datos_otro_cliente['host'], self.datos_otro_cliente['port']))
        print("Cliente conectado exitosamente al otro cliente.")

    def listen(self) -> None:
        thread = threading.Thread(target=self.listen_thread, daemon=True)
        thread.start()

    def listen_thread(self) -> None:
        while True:
            response = self.sockets['client'].recv(4096)
            print("El siguiente mensaje fue una respuesta por el otro cliente P2P:")
            print(f"{response.decode()}\n>>> ", end='')

    def send(self, msg: str) -> None:
        self.sockets['client'].sendall(msg.encode())   # enviamos como cliente

    def repl(self) -> None:
        time.sleep(1)
        print("------ Consola P2P (2) ------\n>>> ", end='')
        while True:
            msg = input("Escribe tu mensaje al otro cliente: ")
            self.send(msg)
            time.sleep(1)


if __name__ == "__main__":
    host = 'localhost'
    port = 8081
    datos_otro_cliente = {'port': 8080, 'host': 'localhost'}
    client = Client_2(host, port, datos_otro_cliente)
```

¿Qué hace este código? Primero abre su *socket* de servidor (puerto `8081`) y empieza a aceptar conexiones en un hilo. Luego, como asume que el otro par ya está corriendo, se conecta como *cliente* al puerto `8080`. Resultado: dos conexiones cruzadas (una iniciada por cada par), por las que ambos pueden escribir y responder.

El `cliente_1.py` es casi idéntico (puertos intercambiados: escucha en `8080` y apunta a `8081`), con **una diferencia clave**: como se ejecuta primero, **no se conecta de inmediato**; espera a que el otro par lo contacte y recién entonces se conecta a él:

```python
# En Client_1, dentro de accept_connections_thread:
        while True:
            client_socket, _ = self.sockets['server'].accept()
            listening_client_thread = threading.Thread(
                target=self.listen_client_thread, args=(client_socket, ), daemon=True)
            listening_client_thread.start()
            # Como ya aceptamos al otro cliente, sabemos que está online:
            # ahora podemos conectarnos a su servidor y volvernos sus clientes
            try:
                self.connect_to_server()
                self.listen()
                self.repl()
            except ConnectionError:
                print("Conexión terminada.")
                self.sockets['client'].close()
                exit()
```

! Limitación de este ejemplo

En `cliente_1.py`, `repl()` es un ciclo infinito que corre *dentro* del hilo que acepta conexiones: una vez que el primer par se conecta, ese hilo queda ocupado en el `repl` y **no vuelve a aceptar a nadie más**. Por eso el ejemplo funciona para **exactamente dos pares**. Si quieres más, tendrías que mover el `repl` a otro hilo. (El apunte lo deja como desafío: unificar ambos archivos en un solo `cliente.py` que maneje el orden de arranque.)

✓ Idea común a todos los ejemplos

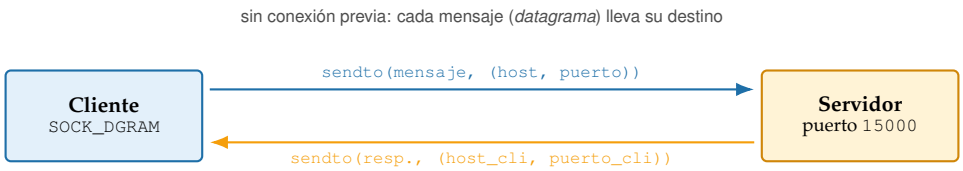
Todos los programas con *sockets* se resumen en:

- **Quién es servidor y quién es cliente** (recordando que puede haber varios de cada uno, y clases que sean ambos).
- **Un *thread* por cada *socket*** que instanciemos.
- **Un *thread* por cada cliente** que se conecte a nosotros.
- Evaluar la necesidad de ***threads* extra** (para interactuar con la terminal, por ejemplo, u otros procesos).

Con esto puedes armar distintas combinaciones de arquitecturas para tus programas.

7 Bonus: sockets UDP

Además de TCP, existe el protocolo **UDP**, que se usa con `SOCK_DGRAM`. Como **no establece una conexión**, su implementación es mucho más simple: no hay `connect`, `listen` ni `accept`. El apunte incluye los archivos de cliente y servidor UDP en la carpeta `codigo/UDP (Bonus)`.



7.1 Cliente UDP

Cada vez que enviamos un mensaje con `sendto(mensaje, (host_destino, puerto_destino))` **debemos especificar la dirección y el puerto** del servidor (a diferencia de `sendall` y `recv` de TCP, que ya tienen la conexión).

```python
import socket

server_host = socket.gethostname()
server_port = 15000

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
mensaje = "Hola, simplemente te estoy enviando un mensaje.".encode('utf-8')
sock.sendto(mensaje, (server_host, server_port))
```

Opcionalmente, podemos recibir una respuesta con `recvfrom(buffer)`. Además de los datos, retorna **la dirección y el puerto desde donde fueron enviados**. El `buffer` es la cantidad máxima de bytes a leer (una potencia de 2 pequeña). Al terminar, cerramos el *socket*.

```
data, (host_origen, puerto_origen) = sock.recvfrom(4096)
print(data.decode('utf-8'))

sock.close()
```

7.2 Servidor UDP

El servidor solo debe preocuparse de **responder a la misma dirección desde donde le escribieron**. Como no hay conexión previa, cada `recvfrom` entrega los datos y la IP y puerto del emisor.

```
## Corre este código desde servidor_udp en codigo/UDP (Bonus); no en Jupyter
import socket

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

# Un string vacío en el bind indica que el socket es alcanzable desde
# cualquier dirección que pueda tener el servidor
sock.bind(("", 15000))

while True:
    data, (host_cliente, puerto_cliente) = sock.recvfrom(4096)
    print(data.decode('utf-8'))
    respuesta = f"Aquí va mi respuesta para {host_cliente}."
    sock.sendto(respuesta.encode('utf-8'), (host_cliente, puerto_cliente))
```

¿Qué hace este código? Hace `bind` al puerto 15000 y queda en un ciclo infinito: cada datagrama que llega se imprime y se responde **al remitente** usando la dirección que entregó `recvfrom`. No hay `accept`: **un solo *socket*** atiende a todos los clientes.

★ Un detalle de *encoding*

El apunte decodifica el mensaje del cliente con `'ascii'` en el servidor, pero el cliente codifica con `'utf-8'`. Funciona porque el texto de ejemplo es ASCII; si el mensaje llevara tildes o `ñ`, `decode('ascii')` fallaría. Usa **UTF-8 en ambos lados** (como en el código de arriba).

7.3 TCP vs. UDP

	TCP (SOCK_STREAM)	UDP (SOCK_DGRAM)
Conexión	Sí: `connect` / `listen` / `accept`	No hay conexión previa
Enviar	`send` / `sendall`	`sendto(datos, (host, puerto))`
Recibir	`recv(buffer)`	`recvfrom(buffer)`: entrega datos y dirección del emisor
Servidor	Un *socket* nuevo por cliente (`accept`)	Un único *socket* para todos
Garantías	Entrega confiable y en orden	Sin garantía de entrega ni de orden
Se usa en	Web, transferencia de archivos, chats	Transmisión en vivo, juegos, consultas rápidas

8 Hoja de referencia

Herramienta	Qué hace	Detalle clave
`pickle.dumps(obj)`	Objeto → `bytes`	`__getstate__` personaliza el estado.
`pickle.loads(bytes)`	`bytes` → objeto	`__setstate__` personaliza la carga. **Inseguro con datos ajenos.**
`pickle.dump / load`	Igual, con archivos	Abrir en modo binario (`"wb"`, `"rb"`).
`str.encode(enc) / bytes.decode(enc)`	`str` ↔ `bytes`	Usa UTF-8; `errors=` controla los fallos.
`bytes / bytearray`	Secuencia inmutable / mutable	Indexar da un `int` (0–255).
`int.to_bytes(n, orden)`	Entero → bytes	`'big'` o `'little'`; falla si `n` no alcanza.
`int.from_bytes(b, orden)`	Bytes → entero	Usar el **mismo orden** que al codificar.
`socket(AF_INET, SOCK_STREAM)`	Crea un *socket* TCP	`SOCK_DGRAM` para UDP.
`bind((host, port))`	Asocia el *socket* a un puerto	Solo el servidor; puerto único por computador.
`listen(n)`	Empieza a escuchar	`n`: conexiones pendientes máximas.
`accept()`	Acepta un cliente	**Bloquea**; retorna `(socket_cliente, address)`.
`connect((host, port))`	Conecta con un servidor	*host* puede ser IP o *hostname* (DNS).
`send(b) / sendall(b)`	Envía bytes	`send` retorna cuántos envió; `sendall` envía todo.
`recv(n)`	Recibe a lo más `n` bytes	**Bloquea**; `b''` = conexión cerrada.
`sendto / recvfrom`	Envío y recepción en UDP	`recvfrom` entrega la dirección del emisor.
`close()`	Cierra el *socket*	Usar `finally` para no olvidarlo.
`threading.Thread(target=, args=, daemon=)`	Hilo de ejecución	Un hilo por *socket* y por cliente conectado.

! Errores frecuentes

- Confiar en el `print` de bytes (`b'\t'` es el byte 9) en vez de convertir a entero.
- Usar `b"áb"` (solo ASCII en literales), o decodificar con el *encoding* equivocado.
- Mezclar el `byteorder`: codificar en `'big'` y decodificar en `'little'`.
- Hacer `pickle.loads` de datos que vienen de **desconocidos** (ejecución de código).
- Intentar serializar un `socket` con `pickle`, o no definir la clase al deserializar una instancia.
- Creer que `recv(4096)` entrega 4096 bytes: **solo asegura un máximo**. Para mensajes grandes, usar encabezado de largo y leer por *chunks*.
- No cerrar los *sockets* (usar `finally`), o no detectar `b''` cuando el otro lado se desconecta.
- Usar el *socket* del servidor para hablar con un cliente (hay que usar el que retorna `accept`).
- Levantar dos servidores en el mismo puerto (`OSError`).
- Esperar a que `accept`/`recv`/`input` no bloqueen: sin *threads*, un cliente detiene a los demás.
- Poner lógica del programa dentro de las clases `Client`/`Server` en vez de un *backend* separado.

✓ Checklist de una comunicación sana

1. ¿Ambos lados usan el **mismo *encoding*** y el **mismo `byteorder`**?
2. ¿Mis mensajes llevan un **encabezado de largo** y leo por *chunks* hasta completarlos?
3. ¿Controlo el caso en que el otro lado se desconecta (`b''`, `ConnectionError`)?
4. ¿Cierro los *sockets* con `finally`? ¿Uso el *socket* correcto (el de `accept`, no el del servidor)?
5. ¿Tengo un ***thread*** por *socket* y por cliente, para que nada bloquee a los demás?
6. ¿Estoy deserializando con `pickle` datos de **fuentes confiables**? Si no, ¿puedo usar JSON?
7. ¿La **lógica** del programa vive fuera de las clases de red?



Fin del resumen • ¡Que tus bytes lleguen siempre completos y en orden!