Contenido

IPC

IPC Python documentación

IPC son las siglas de “Inter-Process Communication”(“Comunicación Interprocesos”), son mecanismos y métodos utilizados por los procesos en un sistema operativo para comunicarse entre sí. Los procesos son ejecuciones de programas que se encuentran en ejecución en un sistema y pueden ser independientes o relacionados.

La comunicación entre procesos es esencial en sistemas operativos multitarea y multiproceso, donde varios procesos pueden estar en ejecución simultáneamente y necesitan intercambiar información o coordinar sus actividades.

Algunos de los mecanismos comunes de IPC incluyen:

1. **Pipes (Tubos)**
   - Los pipes son canales de comunicación unidireccionales que permiten la transferencia de datos entre dos procesos. Un proceso puede escribir en un extremo del pipe, y otro proceso puede leer desde el otro extremo.
2. **Colas (Queues)**
   - Las colas son estructuras de datos que permiten la comunicación entre procesos a través de mensajes. Un proceso puede poner mensajes en la cola y otro proceso puede retirarlos. Esto permite la comunicación asíncrona.
3. **Memoria Compartida**
   - La memoria compartida permite que varios procesos compartan la misma área de memoria. Esto facilita la comunicación rápida ya que los datos pueden ser escritos y leídos directamente en la memoria compartida.
4. **Señales**
   - Las señales son mecanismos de IPC basados en interrupciones que permiten a un proceso notificar a otro proceso sobre eventos o solicitudes específicas.
5. **Sockets**
   - Los sockets permiten la comunicación entre procesos en diferentes nodos de una red. Se utilizan en aplicaciones cliente-servidor y en la comunicación entre procesos en distintos dispositivos.
6. **RPC (Remote Procedure Call)**
   - RPC es un protocolo que permite que un programa de un sistema se ejecute en otro sistema como si fuera un procedimiento local. Facilita la comunicación entre procesos distribuidos.

La elección del mecanismo de IPC depende de varios factores, incluyendo la naturaleza de la comunicación, la complejidad de la aplicación y los requisitos de rendimiento. Los sistemas operativos proporcionan API (Interfaz de Programación de Aplicaciones) para facilitar la implementación de la comunicación entre procesos.

Librería IPC Python incluye sockets, memoria compartida, señales, pipes anónimas y nombradas, y más. Algunas varían en portabilidad, complejidad y utilidad.

- *Signals* permiten a programas enviar una notificación simple de eventos a otros programas.

- *Anonimous pipes* permiten hilos y procesos relacionados para compartir descriptores de ficheros para pasar información, pero generalmente es asociado a un bifurcado del modelo Unix para procesos, el cual no es universalmente portable.
- *Named pipes* son mapas en el sistema de ficheros que permiten a programas sin relaciones conversar, pero en Python no está disponibles para todas las plataformas.
- *Sockets* mapa para todo el sistema de números de puertos, permite transferir data entre programas arbitrarios en ejecución en el mismo host, pero además permite comunicación entre máquinas en la red, esta es una opción más portable.

Otras herramientas como `mmap` proveen la forma de compartir memoria.

Módulo `multiprocessing` ofrece opciones adicionales y portables IPC, incluyendo memoria compartida, pipes, queues de objetos `pickle` Python.

Internet Protocol

El tercer protocolo de nivel de red es IP (Internet Protocol - Protocolo Internet), que proporciona la entrega de paquetes sin conexión no fiable para Internet.

IP no tiene conexiones porque trata cada paquete de información de forma independiente. No es fiable porque no garantiza la entrada, lo que significa que no necesita reconocimientos del sistema principal de envío, del sistema principal de recepción ni de los sistemas principales intermedios.

IP proporciona la interfaz en los protocolos de nivel de interfaz de red. Las conexiones físicas de una red transfieren la información de una trama con una cabecera y datos. La cabecera contiene la dirección de origen y la dirección de destino. IP utiliza un datagrama de Internet que contiene información similar a la trama física. El datagrama también tiene una cabecera que contiene direcciones de protocolo de Internet del origen y del destino de los datos.

IP define el formato de todos los datos enviados a través de Internet.

Identificando máquinas

- Nombre de máquina : string de números separados por puntos (ejemplo: 166.93.218.100) o mediante nombre de dominio (ejemplo: starship.python.net) que son automáticamente mapeado al servidor. localhost y 127.0.0.1 indica la propia máquina.
- Número de puerto : es el puerto asignado para la conversación, tanto el cliente y servidor debe tener el mismo puerto.

Capa de protocolo

Protocolo estándar de Internet define una forma de estructura para hablar sobre sockets, estandarizar formato de mensaje y número de puerto:

- Formato de mensajes provee estructura para intercambio bytes sobre sockets durante la conversación.
- Número de puerto son reservados mediante números adjuntados al socket por el que se intercambia el mensaje.

Sockets Raw todavía son usados en los sistemas, pero es más común y fácil usar los estándares de alto nivel de protocolo de Internet.

Número de puertos va entre `0` y `65535`, es un valor entero de 16-bit, siendo los puertos `0` a `1023` son reservados para el sistema.

Protocolo	Función	Puerto	Módulo
HTTP	Web pages	80	`http.client`, `http.server`
NNTP	Usenet news	119	`nntplib`
FTP data default	File transfers	20	`ftplib`
FTP control	File transfers	21	`ftplib`
SMTP	Send email	25	`smtplib`
POP3	Fetching email	110	`poplib`
IMAP4	Fetching email	143	`imaplib`
Finger	Informational	79	n/a
SSH	Command lines	22	n/a: third party
Telnet	Command lines	23	`telnetlib`

Intentar usar `socket.bind()` en un puerto reservado elevará un error de permiso `PermissionError`, no se pueden usar estos puertos sin tener privilegios de administrador o root.

Módulos relacionados internet

Internet protocol - Python

Python módulos	Utilidad
`socket`, `ssl`	Soporta comunicación de red y IPC (TCP/IP, UDP, etc) envuelto en socket seguros SSL.
`cgi`	Script Server-side CGI: analiza el stream entrada, analiza texto HTML, etc.
`urllib.request`	Obtiene páginas web desde URLs.
`urllib.parse`	Divide URL en componente, escapa el texto URL.
`http.client`, `ftplib`, `nntplib`	Módulos HTTP (web), FTP (file transfer), y NNTP (news).
`http.coockies`, `http.cookiejar`	Soporta cookies HTTP (almacena data en clientes por peticiones de websites, server-side y client-side).
`poplib`, `imaplib`, `smtplib`	Protocolo módulo POP, IMAP (fetch mail), SMTP (send mail).
`telnetlib`	Protocolo telnet.
`html.parser`, `xml.*`	Analiza contenido web (documento HTML y XML).
`xdrlib`, `socket`	Codifica data binaria para la transmisión.
`struct`, `pickle`	Codifica objetos Python en paquetes binarios o bytes serializados para transmisión.
`email.*`	Analiza y compone mensaje email, con headers, adjuntos, encodings.
`mailbox`	Procesa en disco mailboxes y sus mensajes.

Python módulos	Utilidad
`mimetypes`	Consulta el tipo del fichero desde nombres y extensiones.
`uu`, `binhex`, `base64`, `binascii`, `quopri`, `email.*`	Codifica y decodifica a binario (u otros) para transmitir texto (automático en paquete `email`).
`socketserver`	Framework para servidor Net.
`http.server`	Implementación básica HTTP, con manejador de request para un servidor simple CGI.

Pipes

Pipes, dispositivos de comunicación entre programas, son implementados por el sistema operativo y son disponibles en la librería estándar Python. Son canales unidimensionales que trabajan de forma similar a buffer de memoria compartida, pero con una interface re-ensamblando un fichero simple en cada dos finales.

Un ejemplo, un programa escribe data al final del pipe, y otro lee la data en el final del otro. Cada programa solamente ve el final y lo procesa usando llamados Python de ficheros normales.

Pipes hace mucho más dentro del sistema operativo como, llamado a leer un pipe normalmente bloqueará el llamador hasta que la data esté disponible en lugar de retornar el indicador de final de fichero. Leer llamados en un Pipe siempre retorna información vieja escrita en él por el modelo *first-in, first-out* implementado, la primera data escrita para ser leída. Pipes también poseen formas para sincronizar la ejecución de programas independientes.

Pipes tienen dos formas:

1. anonymous pipes
2. named pipes (fifos)

Anonymous pipes

Existen solamente dentro de procesos y son usados típicamente en conjunto con procesos de bifurcaciones para enlazar los procesos padres a los procesos hijos dentro de la aplicación. Padres e hijos comparten descriptores de fichero pipes que son heredados por procesos generados debido a que los hilos están en el mismo proceso y comparten la memoria global.

Anonymous pipes permite relacionar comunicar, pero directamente adecuados para lanzamientos independientes de programas.

Anonymous pipes son cargados en memoria, son temporales.

Para fines explicativos, el siguiente ejemplo, usa `os.fork` para crear una copia del llamado del proceso (una copia del pipe "servidor"), luego el proceso original del padre y su hijo copian la conversación usando los dos finales de un pipe creado con `os.pipe` creado para bifurcarlo, `os.pipe` retorna dos *descriptores de ficheros* (identificadores de fichero de bajo nivel) que representan la entrada y salida del pipe. Los hijos obtienen una copia de los descriptores de ficheros de los padres, escribiendo al final del pipe de salida del hijo y enviando de vuelta la data al padre en el pipe creado antes que el hijo apareciera.

binary data

```
import os, time

def child(pipeout):
    zzz = 0
    while True:
        time.sleep(zzz)   # make parent wait
        msg = ('Spam %03d' % zzz).encode()   # pipe binary bytes - utf-8
        os.write(pipeout, msg)   # send to parent
```

```
        zzz = (zzz + 1) % 5   # goto 0 to 4


def parent():
    pipein, pipeout = os.pipe()   # make 2-ended pipe
    if os.fork() == 0:   # copy this process
        child(pipeout)   # in copy, run child
    else:   # in parent, listen to pipe
        while True:
            line = os.read(pipein, 32)   # blocks until data sent, >= 32 bytes
            print('Parent %d got [%s] at %s' % (
                            os.getpid(),
                            line,
                            time.time()
                    )
            )


parent()
```

`os.fork` no está disponible para Windows, se debe usar `multiprocessing.Pipe`.

Se puede decir, que es un ejemplo básico del modelo cliente-servidor por el modelo de conversación similar empleado, siendo parent el servidor y el cliente es child, los hijos tienen distintos tiempos de ejecución y el padre espera por la data en binario de el o los hijo/s.

El cliente puede escribir dos mensajes distintos y en algunas plataformas o configuraciones, podrían intercalarse o procesarse lo suficientemente cerca en el tiempo para que el padre los recupere como una sola unidad. El padre pide leer ciegamente como máximo 32 bytes, pero recupera cualquier texto disponible en el pipe cuando esté disponible.

Pero, los named pipes son mejor para modelo cliente-servidor porque pueden ser accedidos por procesos sin relaciones y arbitrarios, no requieren que sean bifurcados.

`str` object

Usando `os.fdopen` permite retornar un `str`, normalmente los pipes lidian con string binary byte cuando sus descriptores son usados directamente con herramientas `os`.

Para distinguir mejor el mensaje se puede crear un separador de carácter en un pipe, un `\n` puede envolver un descriptor pipe en un objeto de fichero con `os.fdopen` y confiar en el método del fichero de objeto `readline()` para escanear el separador `\n` en el pipe.

```
import os, time

def child(pipeout):
    zzz = 0
    while True:
        time.sleep(zzz)   # make parent wait
        msg = ('Spam %03d\n' % zzz).encode()   # pipes are binary in 3.X
        os.write(pipeout, msg)   # send to parent
        zzz = (zzz + 1) % 5   # roll to 0 at 5
```

```
def parent():
    pipein, pipeout = os.pipe()   # make 2-ended pipe
    if os.fork() == 0:   # in child, write to pipe
        os.close(pipein)   # close input side here
        child(pipeout)
    else:   # in parent, listen to pipe
        os.close(pipeout)   # close output side here
        pipein = os.fdopen(pipein)   # make text mode input file object
        while True:
            line = pipein.readline()[:-1]   # blocks until data sent
            print(
                    'Parent %d got [%s] at %s' % (
                            os.getpid(),
                            line,
                            time.time()
                        ),
                        type(line)
                )


parent()
```

En este caso, el padre lee los mensajes usando `os.fdopen` el cual retorna un `str`. Además cierra los pipes sin usar.

Threads y anonymous pipes

Los hilos ejecutan en el mismo proceso y comparten los descriptores de ficheros y memoria global en general, crea pipes anónimos útiles como dispositivos de comunicación y sincronización por hilo, se podría decir que es un mecanismo inferior bajo nivel en comparación con queues o nombres compartidos y objetos, pero provee opciones IPC adicionales por hilo.

```
import os, time, threading


def child(pipeout):
    zzz = 0
    while True:
        time.sleep(zzz)
        msg = ('Spam %03d' % zzz).encode()
        os.write(pipeout, msg)
        zzz = (zzz + 1) % 5


def parent(pipein):
    while True:
        line = os.read(pipein, 32)
        print('Parent %d got [%s] at %s' % (os.getpid(), line, time.time()))
```

```
pipein, pipeout = os.pipe()
threading.Thread(target=child, args=(pipeout,)).start()
parent(pipein)
```

Es posible que se deba usar un gestor de tareas para poder terminar los hilos que no se cierren en Windows.

Bidireccional IPC y anonymous pipes

Pipes normalmente permiten el flujo de data en una sola dirección, un lado de entrada y un lado de salida.

Un pipe no puede manejar comunicación bidireccional, pero dos pipes si pueden, uno usado para pasar la petición al programa y otro para retornar la respuesta al solicitante.

Por ejemplo, crear una función que bifurcan programas hijos y conecten el stream *input* y *output* del padre al *input* y *output* del hijo, siendo:

- El padre lee la entrada estándar, estará leyendo el texto enviado por la salida estándar del hijo.
- El padre escribe a la salida estándar, estará enviando la data a la entrada estándar del hijo.

  El efecto de red que existe entre dos programas independientes, existen gracias que deben conversar usando streams estándar.

- *anon_bidirectional_pipe.py*

```
import os, sys

def spawn(prog, *args):
    # obtiene descriptores para streams, stdin=0, stdout=1
    stdinFd = sys.stdin.fileno()
    stdoutFd = sys.stdout.fileno()
    parentStdin, childStdout = os.pipe()
    childStdin, parentStdout = os.pipe()
    pid = os.fork()
    if pid:
        # en padre despues de fork
        os.close(childStdout)
        os.close(childStdin)
        os.dup2(parentStdin, stdinFd)
        os.dup2(parentStdout, stdoutFd)
    else:
        # en child despues de fork
        os.close(parentStdin)
        os.close(parentStdout)
        os.dup2(childStdin, stdinFd)
        os.dup2(childStdout, stdoutFd)
        args = (prog,) + args
        # nuevo programa en el proceso
        os.execvp(prog, args)
```

```
        assert False, 'execvp failed!'


if __name__ == '__main__':
    mypid = os.getpid()
    # fork child program
    spawn('python', 'anon_pipe_bidirectional_test_child.py', 'spam')
    # to child stdin
    print('Hello 1 from parent', mypid)
    sys.stdout.flush()
    # from child stdout
    reply = input()
    sys.stderr.write('Parent got: "%s"\n' % reply)# to child's stdin
    print('Hello 2 from parent', mypid)
    sys.stdout.flush()
    reply = sys.stdin.readline()
    sys.stderr.write('Parent got: "%s"\n' % reply[:-1])
```

`os.fork` (no funciona en Windows), copia el llamado del proceso y retorna el processID del hijo en padre solamente.

`os.execvp` solapa un nuevo programa en el proceso de llamado y toma una tupla o lista de comandos en forma de string (recogidos por `*args`), similar a `os.execlp`.

`os.pipe` retorna una tupla de descriptor de fichero representando el *input* y *output* del final del pipe.

`os.close(fd)` cierra el descriptor file.

`os.dup2(fd1, fd2)` copia toda la información del sistema asociado con el nombre del fichero del file descriptor *fd1* al fichero nombrado *fd2*. Esencialmente asigna el proceso padre fichero *stdin* a la entrada final de uno de los dos pipes creados, de ahora en adelante, todas las lecturas vendrán del pipe. Para conectar con el otro final del pipe del proceso hijo, copia el fichero stream *stdout* con `os.dup2(childStdout,stdoutFd)`, el texto escrito por el hijo a *stdout* termina siendo enrutado a través del pipe stream *stdin* del padre. En este ejemplo anterior es de más bajo nivel y menos portátil.

- *anon_pipe_bidirectional_test_child.py*

```
import os, time, sys

mypid = os.getpid()
parentpid = os.getppid()
sys.stderr.write('Child %d of %d got arg: "%s"\n' %
(mypid, parentpid, sys.argv[1]))

for i in range(2):
    time.sleep(3)   # make parent process wait by sleeping here
    recv = input()   # stdin tied to pipe: comes from parent's stdout
    time.sleep(3)
    send = 'Child %d got: [%s]' % (mypid, recv)
    print(send)   # stdout tied to pipe: goes to parent's stdin
    sys.stdout.flush()   # make sure it's sent now or else process blocks
```

Output stream: deadlocks y flushes

Estado deadlock ocurre cuando un *input* se queda esperando eventos que no ocurren desde un *output*, como recibir el mensaje del *output*, por lo que es necesario realizar un *flush* o *envío*.

`sys.output` es *line-buffered* (se envía la información cuando encuentra un salto de línea) en una terminal, pero es *fully buffered* cuando se conecta a otros dispositivos como ficheros, sockets, pipes, es debido a esto que el texto se muestra inmediatamente como se producen, pero no hasta que el proceso cierre o su buffer se conecta a algo más.

Esta es una función de la librerías de sistema usado para acceder a los pipes y no de los pipes en sí, los pipes solamente encolan la data de salida, pero nunca la ocultan a los lectores.

Formas de evitar problemas deadlock cuando se está lidiando con dialogos doble vía:

- *Flushes*: realizar `sys.stdout.flush()` manualmente en el stream de salida del pipe es una forma fácil de forzar la limpieza de los buffers.
- *Argumentos*: la opción de Python en lína de comando `-u`, desactiva el buffering del stream `sys.stdout`, mostrando inmediatamente la salida tan rápido como se genere los datos y sin esperar que se llene el buffer (útil en contenedores, orquestadores, programas que se requiera un vista rápida de la salida). Establecer la variable de entorno `PYTHONUNBUFFERED`, `export PYTHONUNBUFFERED=0`.
- *Open modes*: es posible modo unbuffered en pipes. Cualquiera de los módulos de bajo nivel de `os` llaman a leer y escribir descriptores pipes directamente, o pasar un tamaño de buffer de argumento `0` (unbuffered) o `1` (line-buffered) para `os.fdopen` para deshabilitar el buffering en el objeto fichero usado para envolver el descriptor. Se puede usar argumentos open de la misma forma para controlar el buffering de la salida de ficheros *fifos*. Python 3.X soporta completamente el modo unbuffered solamente en ficheros de modo binario, no textos.
- *Command pipes*: se puede especificar el modo de buffering usando argumentos para pipes de línea de comandos cuando es creado por `os.popen` y `subprocess.Popen`, pero este pertenece al caller final del pipe, no al programa generado. Por lo tanto, no puede evitar las salidas retrasadas de este último, pero puede usarse para enviar texto al canal de entrada de otro programa.
- *Sockets*: `socket.makefile` acepta un modo argumento de buffering similar para sockets, pero Python 3.X requiere llamar en modo-texto y parece no soportar modo line-buffering.
- *Tools*: para tareas más complejas, también podemos utilizar herramientas de nivel superior que esencialmente engañan a un programa haciéndole creer que está conectado a una terminal. Estos programas no están escritos en Python, por lo que `-u` no es una opción.

`Thread` se puede usar, pero realmente se está delegando el problema, los hilos generados todavía estarán estancados. De las soluciones anteriores, las primeras 2 son una solución simple. Al comentar las líneas `sys.stdout.flush()` de un programa y usar `python -u file.py` para desactivar el buffering, funcionará correctamente de forma igual.

Named pipes

Son ficheros externos, los procesos de comunicación pueden no estar relacionados, pueden ser iniciados por programas independientemente.

El sistema operativo sincroniza el acceso a fifos, haciendo especialmente útil para IPC, los pipes fifos son mejor ajustado a mecanismos generales IPC para clientes independientes y programas

servidores, por ejemplo, ejecutar un servidor permanentemente para escuchar peticiones en un fifo para accedido por clientes arbitrarios sin bifurcar el servidor.

En algunas plataformas, es posible crear pipes de larga vida que existan como un fichero nombrado real en el sistema de ficheros, como los ficheros son llamados *named pipes* (*fifos*) porque ellos se comportan como pipes anónimos.

Sin embargo, debido a que los fifos están asociados con un archivo real en su computadora, son externos a cualquier programa en particular: no dependen de la memoria compartida entre tareas y, por lo tanto, pueden usarse como un mecanismo IPC para subprocesos, procesos y ejecutarse de forma independiente. programas.

Cuando son creados, los clientes lo abren por su nombre, leen y escriben data usando operaciones normales de fichero. **Fifos son streams unidimensionales.** En operaciones típicas, un servidor lee la data desde el fifo, y una o más clientes escriben la data en este, adicionalmente, un grupo de dos fifos pueden ser implementados en comunicaciones bidireccionales así como pipes anónimos.

Con fifos, los pipes son accedidos en lugar por un nombre de ficheros visible para todos los programas que se están ejecutando en el computador sin importar la relación entre parent/child.

Fifos, en cierta forma, son una alternativa a puerto de interface socket. A diferencia de *sockets*, fifos no soportan directamente conexiones remotas, no están disponibles actualmente en el estándar Windows Python y se accede a ellos mediante la interfaz de archivos estándar en lugar de los números de puerto de socket más exclusivos.

- *named_pipe.py*

```
import os, time, sys

fifoname = '/tmp/PYpipefifo'   # must open same name

def child():
    # open fifo pipe file as file descriptor
    pipeout = os.open(fifoname, os.O_WRONLY)
    zzz = 0
    while True:
        time.sleep(zzz)
        msg = ('%s -- Spam %03d\n' % zzz).encode()   # binary text
        os.write(pipeout, msg)
        zzz = (zzz + 1) % 5


def parent():
    pipein = open(fifoname, 'r')   # open fifo as text file obj
    while True:
        line = pipein.readline()[:-1]   # blocks until data sent
        print('Parent %d got "%s" at %s' % (os.getpid(), line, time.time()))


if __name__ == '__main__':
    if not os.path.exists(fifoname):
        os.mkfifo(fifoname)   # create a named pipe file

    if len(sys.argv) == 1:
        print('PARENT')
```

```
        parent()   # run parent()
    else:
        print('CHILD')
        child()   # run child
```

- `os.open(path, flags, mode=0o777, *, dir_fd=None)` : abre el fichero, retorna file descriptor. `os.O_RDONLY`, `os.O_WRONLY` permiten leer y escribir al fifo.
- `os.mkfifo(path, mode=0o666, *, dir_fd=None)` : crea un FIFO usando la ruta dada, con permisos por defecto 666.

Casos de uso de fifos

- Para mapear puntos de comunicación a un fichero de sistema enteramente accesible para todos los programas que se ejecutan en la máquina, fifos puede abordar una amplia gama de objetivos de IPC en las plataformas donde son compatibles.
- Para programas sin relación, ficheros *fifos* son más ampliamente aplicables a modelos cliente/servidor. Por ejemplo, fifos pueden crear una integración entre un GUI y un depurador de línea de comandos, se pueden conectar stream GUI con stream no GUI.

Los sockets brindan una funcionalidad similar pero también nos brindan un conocimiento de red inherente y una portabilidad más amplia para Windows, como se explica en la siguiente sección.

Sockets

Sockets documentación

Sockets permite transferir data entre programas en ejecución en el mismo computador (IPC), el programa se conecta a sockets a un número de puerto global y transfiere la data. O en la red es bidireccional, el programa provee un nombre de la máquina y el número de puerto para transferir y recibir data del programa remoto.

> Por cada nueva conexión se crea una en el lado del servidor usando el método `.accept()` (retorna `conn, address = sock.accept()`) se crea un nuevo objeto socket este es exclusivo de dicha conexión. En este nuevo objeto socket se puede enviar y recibir datos.

- Como FIFOs (*first in, first out*), sockets son globales a través de la máquina, no requieren de memoria compartida entre hilos o procesos, aplicables a programas independientes.
- Se distinguen de los FIFOs (*first in, first out*[1]), sockets son identificados por un número de puerto, no una ruta en el sistema de fichero. Sockets implementan API no fichero, en la que se pueden envolver en un objeto tipo fichero, son más portables, funcionan en cualquier plataforma Python.

Módulo `socket` incluye una variedad de herramientas avanzadas:

- Convertir bytes a un orden de red estándar (`.ntohl(x)`, `.htonl(x)`).
- Consultar nombre y dirección de máquina (`.gethostname()`, `.gethostbyname(hostname)`).
- Envuelve un objeto socket en un fichero objeto interface (`sockobj.makefile()`).
- Crea un socket no bloqueante (`sockobj.setblocking(flag)`).
- Establece timeout socket (`sockobj.settimeout(value)`)

Python provee librería `Secure Socket Layer (SSL)`, se puede envolver SSL en un socket usando `ssl.wrap_socket`. Demás es implementada en `http.client` y `urllib.request` para soportar HTTPS, transferencia segura de emails (`poplib` y `smtplib`), y más.

Código cliente y servidor

La comunicación usando socket es bidireccional.

```
from socket import socket, AF_INET, SOCK_STREAM   # portable socket API

from threading import Thread

port = 50008   # port number
host = 'localhost'   # server and client machine direction

def server():
    sock = socket(AF_INET, SOCK_STREAM)   # ip addresses tcp connection
    sock.bind(('', port))   # bind to port on this machine
    sock.listen(5)   # allow up to 5 pending clients
    while True:
        conn, addr = sock.accept()   # wait for client to connect
```

```
        data = conn.recv(1024)   # read bytes from this client
        reply = 'Server got: [%s]' % data   # conn is new connected socket
        conn.send(reply.encode())   # send bytes reply back to client


def client(name):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)   # reuse port
    sock.connect((host, port))   # connect to socket port
    sock.send(name.encode())   # send bytes to listener
    reply = sock.recv(1024)   # recieve data from listener to 1024 bytes
    sock.close()   # close socket
    print('client got: [%s]' % reply)


if __name__ == '__main__':
    sthread = Thread(target=server)
    sthread.daemon = True   # don't wait for server thread
    sthread.start()   # do wait for children to exit
    for i in range(5):
        Thread(target=client, args=('client%s' %i, )).start()
```

En el ejemplo anterior, con este tipo de socket el servidor acepta una conexión de cliente, que de forma predeterminada se bloquea hasta que un cliente solicita el servicio y devuelve un nuevo socket conectado al cliente. Una vez conectados, el cliente y el servidor transfieren cadenas de bytes mediante llamadas de envío y recepción en lugar de escrituras y lecturas, aunque, como veremos más adelante en el libro, los sockets se pueden encapsular en objetos de archivo de la misma manera que lo hicimos anteriormente para los *pipes descriptors*. Al igual que los *pipe descriptors*, los sockets no encapsulados tratan con cadenas de `binary` bytes, no con texto `str`; es por eso que los resultados del formato de cadena se codifican manualmente.

En el servidor (`server()`), el puerto se mantiene un abierto luego del último ciclo de conexión de un cliente, al intentar ejecutar nuevamente el servidor dará error `ConnectionRefusedError` debido a que se está usando ese puerto. `TIME_WAIT` es determinado por el fichero en Linux `/proc/sys/net/ipv4/tcp_fin_timeout` es de `60` segundos.

Agregar la línea, `sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)` permite reutilizar el puerto usado sin esperar el tiempo `TIME_WAIT`, pero **precaución, ya que puede haber problemas con las conexiones pendientes o en curso en el mismo puerto cuando intentas volver a abrir el socket.**. El `1` habilita la opción y `0` la deshabilita.

> `tcp_fin_timeout` documentación `tcp_fin_timeout` (integer; default: 60; since Linux 2.2) This specifies how many seconds to wait for a final FIN packet before the socket is forcibly closed. This is strictly a violation of the TCP specification, but required to prevent denial-of-service attacks. In Linux 2.2, the default value was 180.

`socket.SOL_SOCKET` y `socket.SO_REUSEADDR` son constantes del módulo `socket` que se utilizan para establecer configuraciones específicas de socket. Métodos `getsockopt` permite obtener los atributos del socket y `setsockopt` permite establecer los atributos del socket.

Sockets y programas independientes

Al usar sockets y threads, el modelo de memoria compartida de threads permite emplear un dispositivo de comunicación simple como nombres compartidos , objetos y queues.

Sockets tienen más relevancia cuando se están usando con programas independientes, en comparación con las anteriores opciones de conexión.

Al ejecutar un programa servidor, se ejecuta en bucle esperando hasta que llegue una petición de un cliente, una vez que se responde al cliente la conexión se termina, pero el servidor sigue hasta que se cierre el servidor.

Casos de uso de sockets

- Objetos Python como listas y diccionarios (o copias de estos) pueden ser transferidos sobre sockets, usando una **serialización bytes** de los strings que facilita el módulo `pickle` o `json` o convertir los datos manualmente a bytes.
- La salida de un script puede ser redireccionada a una ventana GUI, realizado mediante la conexión del stream de salida de un socket a un socket de escucha en modo no bloqueante de la aplicación GUI.
- Los programas que obtienen texto arbitrario de la Web pueden leerlo como cadenas de bytes a través de sockets, pero decodificarlo manualmente usando nombres de codificación incrustados en encabezados de tipo de contenido o etiquetas en los propios datos.
- Internet puede ser visto como caso de uso de socket; email, FTP, páginas web son formateados en bytes y transportados sobre sockets. Socket son usados para transferir ficheros y escribir socket de servidor más robustos que generan hilos o procesos para conversar con clientes evitando la negación de servicio.

Sockets y clientes múltiples

> Forking server

Intentar manejar clientes múltiples a un servidor, y este le toma mucho tiempo para responder, en un momento este fallará, además el costo de manejar una petición evita que el servidor responda clientes nuevos no podrán ser atendidos denegándoles la conexión.

En programas cliente/servidor, un servidor debe evitar bloqueos para nuevas peticiones mientras está manejando otros clientes en paralelo, ya sea, en una copia del proceso, nuevo hilo, o cambiado manualmente (multiplexión) entre clientes en un loop evento.

Técnicamente existen tres formas de realizar esto:

1. Bifurcando o copiando el servidor (`fork()`) para cada nueva petición solo en sistemas Unix-like, pero no es portable no se puede utilizar en Windows a menos que se use `Cygwin`, esto tiene limitaciones, debido a que se emula elementos de un sistema Unix-like, consume muchos recursos.
2. Hilos (`thread`) por cada petición nueva, más portable, permite ahorro de recursos debido a que comparten descriptores y memoria.
3. Multiplexar el servidor.

Si la máquina tiene múltiples CPUs, lo hilos y procesos se pueden ejecutar en paralelo asignando a cada una un CPU hasta que se completen. De lo contrario, el sistema para ejecutar

estas tareas debe dividir el poder de procesamiento computacional entre todas las tareas activas, ejecutando parcialmente todas estas tareas hasta completarlas, dando una impresión de ejecución en paralelo. El sistema operativo va alternando entre las tareas tan rápido que el humano no se da cuenta, este proceso es llamado **time-slicing** y generalmente conocido como **multiplexación**.

select - multiplexación

Python puede realizar esta dividir las tareas en múltiples pasos, ejecutar una tarea de un paso y otra de otro paso, hasta que todas se completen, el script necesita saber como dividir la atención entre múltiples tareas activas para poder realizar la multiplexión.

Los servidores pueden aplicar está técnica para manejar múltiples clientes, sin requerir de hilos o bifurcaciones. Multiplexando las conexiones clientes y el despachador principal usando el módulo **select**, el cual es un *event loop* que procesa clientes múltiples y acepta nuevos en paralelo, el evento decide que cliente toma la atención. Estos servidores son llamados **asincrónicos** porque atienden a los clientes a intervalos, a medida que cada uno está listo para comunicarse. Las solicitudes de los clientes y el bucle principal del despachador reciben cada uno una pequeña porción de la atención del servidor si están listos para conversar.

`select` permite usar socket listos para comunicarse, evitando bloqueos entre llamados. Cuando se pasan `sockets`, estamos seguros que las llamadas de `accept`, `recv` y `send` no se bloquearán, debido a que es un servidor event-loop único.

Debido a que no inicia hilos o procesos, puede ser más eficiente cuando las transacciones con clientes son relativamente cortas. Requiere de transacciones rápidas, de lo contrario se corre el riesgo de estancamiento a menos que en estos casos se usen hilos `thread` o bifurquen `fork()`.

servidor echo select

El servidor agrega los sockets de cada cliente conectado a la lista `readables` pasado a `select`, y espera por el socket se muestre en la lista `select`. En este caso, el servidor escucha en más de 1 puerto por cliente nuevos, el socket principal es interrgado por `select`, las conexiones requests en ambos puertos pueden ser escuchados sin bloquearse entre ellos. `select` retorna el resultado de los sockets en `readables` que estén listos para procesar, tanto socket puerto principal y socket clientes serán procesados.

```
import sys, time
from select import select
import socket


def now():
    """
    Tiempo actual del servidor.
    """
    return time.ctime(time.time())


myHost = ''
myPort = 50008
```

```
if len(sys.argv) == 3:
    myHost, myPort = sys.argv[1:]


# número de puertos para clientes
numPortSocks = 2


# crea sockets principales para aceptar clientes nuevos
mainsocks, readsocks, writesocks = [], [], []

# bucle para cear sockets
for i in range(numPortSocks):
    portsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    portsock.bind((myHost, myPort))
    portsock.listen(5)

    mainsocks.append(portsock)
    readsocks.append(portsock)

    myPort += 1


# event Loop - multiplexa hasta matar al server
print('###  select-server loop - Ready.  ###')

while True:
    readables, writeables, exceptions = select(readsocks, writesocks, [])
    for sockObj in readables:
        if sockObj in mainsocks:
            print('> New client')
            newsock, address = sockObj.accept()
            print('>> Connect: ', address, id(newsock))
            readsocks.append(newsock)
        else:
            print('> Serving client.')
            data = sockObj.recv(1024)
            print('>> Got: ', data, ' on ', id(sockObj))
            if not data:
                print('>>> Client finished.')
                sockObj.close()
                readsocks.remove(sockObj)
            else:
                print('>>> Send reply to Client.')
                # escribir datos puede ser bloqueante.
                # se puede usar `select` para enviar el mensaje.
                reply = 'Echo => %s at %s' % (data, now())
                sockObj.send(reply.encode())
```

select - Notas

- **Llamado** (*call*, `select()`) : es usado generalmente con 3 listas de objetos seleccionables (fuentes input, output y exceptions), adicionalmente una de timeout que es un `int` o `float`.
- **Portabilidad** : como `thread`, a diferencia de `fork()`, servidores `select` funcionan en Windows Python estándard, técnicamente en en windows `select` funcionan solamente con `sockets`. En sistemas Unix y Macintosh `select` permiten usar `sockets` (principalmente), ficheros, pipes.
- **Socekts No bloquantes** : `select` nos garantiza que no se bloquearán los sockets en los llamados `accept` y `recv`. Se puede establecer socket bloqueantes o no usando `socket.setblocking([0|1])`.
- **asyncio módulo** : `asyncio` permite escribir código concurrente utilizando la sintaxis `async/await`. `asyncio` es utilizado como base en múltiples frameworks asíncronos de Python y provee un alto rendimiento en redes y servidores web, bibliotecas de conexión de base de datos, colas de tareas distribuidas, etc.
- **Twisted** (webpage) : framework asíncrono de red que soporta TCP, UDP, multicast, SSL/TLS, comunicación serial, y más. Incluye implementación cliente y servidor para los servicios más comunes como servidor web, IRC, mail, interface de database relacional, broker object (intermediario o entidad que facilita la comunicación o intercambio entre diferentes partes en un sistema distribuido). Usa hilos y soporta procesos para acciones de larga duración, además de ser asíncrono, modelo de cliente basado en eventos. Abstrae el event loop, que multiplexa las conexiones sockets abiertas, automatizando detalles del servidor asíncrono, además de proveer framework basado en eventos para realizar tareas mediante scripts. Internamente es similar al servidor `select` anterior y al módulo `asynio`.

Signals

> Signal - documentación

Señales se refieren a mecanismos que permiten a un proceso recibir notificaciones asincrónicas sobre eventos o cambios en su entorno. Son una forma de obstaculizar un proceso, los programas generan señales para desencadenar un manejador para esa señal en otros procesos.

El sistema operativo también puede obstaculizar un proceso, algunas señales son inusualmente generadas desde eventos de sistema y pueden matar al programa si no se manejan. Algo así como `raise Exception` en Python.

Señales son eventos generados por software y análogo entre procesos de las excepciones, pero a diferencias de las excepciones, las señales son identificadas por números y no son apiladas, son un mecanismo de eventos asíncronos fuera del alcance de interpretador Python controlado por el sistema operativo.

Módulo `signal` permite a programas Python registrar funciones Python como manejadores de evento de señales. Disponible en plataformas Unix-like y Windows.

```
import sys, signal, time


def now():
    return time.ctime(time.time())   # current time string
```

```
def onSignal(signum, stackframe):   # python signal handler
    print(
        'Got signal ', signum, ' at ', now()
    )   # most handlers stay in effect



signum = int(sys.argv[1])
signal.signal(signum, onSignal)   # install signal handler

while True:
    signal.pause()   # wait for signals or pass
```

Señal	Descripción
SIGINT (`Ctrl + C`)	señal de interrupción de un programa en ejecución.
SIGTERM	señal de terminación, usado para enviar una solicitud de termino de proceso.
SIGKILL	señal para forzar terminación inmediata de un proceso.
SIGHUP	usado para indicar recarga de configuración.
SIGUSR1, SIGUSR2	señales de usuario usados para propósitos específicos del usuario.

- `signal.signal` Toma una señal numérica y un objeto función que instala esa función para manejar ese número de señal cuando es elevada. Python automáticamente restaura muchas manejadores de señales cuando una señal ocurre, por lo que no es necesario llamar a la función cuando ya se ha registrado.

  `signal.signal(signalnum, handler)`, handler es un manejador como una función que reciba dos parámetros (`signum, stackframe`).

  Excepto para *SIGCHLD*, un manejador de señales se mantiene instalado hasta que explícitamente se reinicia (ejemplo, *SIG_DFL* para restaurar comportamiento por defecto o *SIG_IGN* para ignorar la señal). *SIGCHLD* se comporta específicamente para cada plataforma.

- `signal.pause` Crea un proceso de *sleep* hasta la siguiente señal que capture. No usar: *time.sleep()* causa fallas y no funciona en Linux, *while True: pass* usa muchos recursos.

- `signal.alarm` Función para programar una señal *SIGALRM* que ocurrirá en algunos segundos en el futuro. Para desencadenar y captar timeouts, se debe establecer alarmas e instalar un manejador *SIGALRM*.

```
import sys, signal, time

def now():
    return time.asctime()


def onSignal(signum, stackframe):  # python signal handler
```

```
    print(
            'Got alarm ', signum, ' at ', now()
        )       # most handlers stay in effect


while True:
    print('Setting at ', now())
    signal.signal(signal.SIGALRM, onSignal)):  # install signal handler
    signal.alarm(5)):  # do signal in 5 seconds
    signal.pause()):  # wait for signals
```

Obtener la señal

Usar `kill`, el número de la señal usada y el número del proceso permite obtener el mensaje.

```
$ kill -2 11307
```

Eliminar la señal

Para identificar el proceso en background, se puede usar `ps` o `pgrep` en Linux, identificar su PID y eliminarlo con `kill -9`.

```
$ sudo pgrep -af signals.py
11307 python signals.py 2

#

$ ps aux | grep -i signals.py
user      11307  0.0  0.1 21118  9461 pts/1    S    14:04   0:00 python signals.py 2
```

- Algunos sistemas no reaccionan bien siendo interrumpidos por señales, solo el hilo principal se le puede instalar un manejador de señales y responde para señales en programas multi-hilo.
- Bien usados, señales proveen un mecanismo de comunicación basados en eventos, son menos poderosos que los stream de data como *pipes*, pero son suficientes en situaciones que se debe indicar a un programa algo importante que ocurre y no pasar mayores detalles del evento en si.
- Se pueden combinar con otras herramientas IPC, por ejemplo, informar a un programa que un cliente desea comunicarse sobre un `named pipes` (*fifos*), el equivalente a tocar el hombro de alguien para llamar su atención antes de hablar.
- Muchas plataformas reservan una o más señales *SIGUSR* para eventos definidos por usuarios de eventos. Esta estructura de integración es a veces una alternativa a ejecutar una llamada de entrada de bloqueo en un hilo generado.
- `os.kill(pid, sig)` envía señales a procesos conocidos desde un script Python en plataformas Unix-like, como comando `kill`, requiere del pid obtenido desde `os.fork()` de un proceso hijo (disponible en `Cygwin` - Python)

Forking Servers

Bifurcar (copiar) el servidor para cada cliente y retornar la respuesta, es una técnica no portable, es decir, solamente funciona en sistemas Unix-like y no en Windows, usando `os.fork()`. No se resuelve usando módulo `multiprocessing`, porque este bifurca un proceso del sistema, incluso si el procesos child usa la misma conexión socket que el padre, en clientes Windows en una instalación Python estándar, y la razón es que `multiprocessing` no `pickled` correctamente cuando se pasa un argumento a un nuevo proceso. `multiprocessing` tiene otras herramientas IPC como sus propios pipes y queues que pueden ser usados en lugar de socket y los clientes deben usarlos, pero se pierde el propósito general de Internet. Usando Cygwin el servidor se ejecuta normal y clientes.

El problema principal acá es la portabilidad.

```
import os, time, sys

import socket


myHost = ''
myPort = 50008


# make TCP object
sockObj = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# reuse port
sockObj.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
# bind server to port number
sockObj.bind((myHost, myPort))
# allow 5 pending clients
sockObj.listen(5)


def now():
    """
    Tiempo actual del servidor
    """
    return time.ctime(time.time())


activeChildred = []
def killChildren():
    """
    Colecta y elimina proceso Child muertos, si el proceso está vivo
    llena la tabla de sistema.
    No toma en cuenta los procesos Child exit.
    Lista vacía es False, sino es True
    """
    while activeChildred:
        pid, stat = os.waitpid(0, os.WNOHANG)
        if not pid:
            break
        activeChildred.remove(pid)
```

```
def handleClient(connection):
    """
    Proceso hijo, responde y sale.
    Bloqueo arbitrario de 5 segundos, simulando
    petición costosa o demorosa.
    Lee y escribe a un cliente.
    """
    time.sleep(5)
    while True:
        data = connection.recv(1024)
        if not data:
            break
        reply = 'Echo => %s at %s' % (data, now())
        connection.send(reply.encode())
    sys.exit(0)


def dispatcher():
    """
    Escucha hasta que el proceso muera, espera la siguente conexión.
    Limpia los procesos children zombies.
    Copia el proceso padre, si el proceso hijo es nuevo lo maneja, si
    el proceso ya existe, lo agrega a la lista `activeChildred`.
    """
    while True:
        connection, address = sockObj.accept()
        print('Server connected by ', address, end=' ')
        print('at', now())
        killChildren()
        childPid = os.fork()
        if childPid == 0:
            handleClient(connection)
        else:
            activeChildred.append(childPid)


dispatcher()
```

waitpid() - prevenir hijos zombie

`os.waitpid()` `os.WNOHANG`

Además, existe la posibilidad que los procesos hijos no terminen y mueran, quedando en modo zombie y consumiendo recursos.

En sistemas Linux, no Cygwin, los padres se deben esperar un system `wait` para eliminar las entradas de procesos hijos muertos desde la tabla de proceso del sistema.

El uso `sys.exit` no garantiza que los procesos muertos hijos se terminen y liberen los recursos.

El uso de `os.waitpid(0, os.WNOHANG)`, espera por la salida de los procesos hijos y retorna su processID y estado de salida, `0` indica que espera por cualquier proceso hijo y con `WNOHANG` no hace nada si el proceso hijo ha salido, no pausa o bloquea el caller. En resumen, solamente

pregunta al sistema operativo por el PID del proceso del hijo que haya salido y si existen lo retorna.

Aún es poco portable, porque sigue usando `os.fork()`.

signals - prevenir hijos zombies

En plataformas que lo soporten, usando `signal` es posible limpiar procesos child zombies usando un manejador de señal *SIGCHLD* que entregue al proceso padre para el sistema operativo que notifique cuando un proceso child se detenga o salga. Acción *SIG_IGN* (ignore) como manejador de señal *SIGCHLD*, los zombies serán eliminados automáticamente por el sistema operativo como exit, el padre no necesita esperar para limpiarlos.

- Es mucho más simple y no necesita de seguimiento manual para eliminar procesos hijos.
- Es más acertado, no deja peticiones clientes zombies temporales.

Aún es poco portable, porque sigue usando `os.fork()`.

- *signal.py*

```
import sys, signal, time


def now():
    # current time string
    return time.ctime(time.time())


def onSignal(signum, stackframe):
    # python signal handler
    # most handlers stay in effect
    print('Got signal ', signum, ' at ', now())
    # signal.signal(signal.SIGCHLD, onSignal)


signum = int(sys.argv[1])

# install signal handler
signal.signal(signum, onSignal)

# wait for signals or pass
while True:
    signal.pause()
```

- *socket_signal.py*

```
import os, time, sys
import socket
import signal
```

```
myHost = ''
myPort = 50008


# make TCP object
sockObj = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# reuse port
sockObj.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
# bind server to port number
sockObj.bind((myHost, myPort))
# allow 5 pending clients
sockObj.listen(5)

# avoid child zombies
signal.signal(signal.SIGCHLD, signal.SIG_IGN)


def now():
    """
    Tiempo actual del servidor
    """
    return time.ctime(time.time())


def handleClient(connection):
    """
    Proceso hijo, responde y sale.
    Bloqueo arbitrario de 5 segundos, simulando
    petición costosa o demorosa.
    Lee y escribe a un cliente.
    """
    time.sleep(5)
    while True:
        data = connection.recv(1024)
        if not data:
            break
        reply = 'Echo => %s at %s' % (data, now())
        connection.send(reply.encode())
    sys.exit(0)


def dispatcher():
    """
    Escucha hasta que el proceso muera, espera la siguente conexión.
    Limpia los procesos children zombies.
    Copia el proceso padre, si el proceso hijo es nuevo lo maneja, si
    el proceso ya existe, lo agrega a la lista `activeChildred`.
    """
    while True:
        connection, address = sockObj.accept()
        print('Server connected by ', address, end=' ')
        print('at', now())
        childPid = os.fork()
```

```
        if childPid == 0:
            handleClient(connection)


dispatcher()
```

multiprocessing no ayuda

```
from multiprocessing import Process
import os, time, sys

import socket


myHost = ''
myPort = 50008


# make TCP object
sockObj = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# reuse port
sockObj.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
# bind server to port number
sockObj.bind((myHost, myPort))
# allow 5 pending clients
sockObj.listen(5)


def now():
    """
    Tiempo actual del servidor
    """
    return time.ctime(time.time())


def handleClient(connection):
    """
    Proceso hijo, responde y sale.
    Bloqueo arbitrario de 5 segundos, simulando
    petición costosa o demorosa.
    Lee y escribe a un cliente.
    """
    print('Child: ', os.getpid())
    time.sleep(5)
    while True:
        data = connection.recv(1024)
        if not data:
            break
        reply = 'Echo => %s at %s' % (data, now())
        connection.send(reply.encode())
    sys.exit(0)
```

```
def dispatcher():
    """
    Escucha hasta que el proceso muera, espera la siguente conexión.
    Limpia los procesos children zombies.
    Copia el proceso padre, si el proceso hijo es nuevo lo maneja, si
    el proceso ya existe, lo agrega a la lista `activeChildred`.
    """
    while True:
        connection, address = sockObj.accept()
        print('Server connected by ', address, end=' ')
        print('at', now())
        Process(target=handleClient, args=(connection, )).start()


dispatcher()
```

Threading Server

Bifurcar (*fork*) el servidor tiene las siguientes limitaciones: * Rendimiento, en algunos casos las operaciones pueden ser costosas para el hardware disponible. * Portabilidad, bifurcar procesos es una técnica Unix. En Windows se debe instalar Cygwin, pero puede ser ineficiente y no es una bifurcación Unix como tal. * Complejidad, bifurcar puede ser complicado, generan procesos zombies que no devuelven los recursos.

Con los hilos se ejecuta el mismo proceso y espacio de memoria, automáticamente comparte socket pasados entre ellos, similar en espíritus por la forma que los procesos hijos heredan descriptores sockets, funcionan en plataformas Unix-like y Windows. A diferencia de los procesos, los hilos son caros de iniciar.

Los hilos child mueren silenciosamente al salir, no dejan procesos zombies.

No necesita usar `sys.exit(0)`.

```
import threading
import os, time, sys

import socket


myHost = ''
myPort = 50008


# make TCP object
sockObj = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# reuse port
sockObj.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
# bind server to port number
sockObj.bind((myHost, myPort))
# allow 5 pending clients
sockObj.listen(5)
```

```
def now():
    """
    Tiempo actual del servidor
    """
    return time.ctime(time.time())


def handleClient(connection):
    """
    Proceso hijo, responde y sale.
    Bloqueo arbitrario de 5 segundos, simulando
    petición costosa o demorosa.
    Lee y escribe a un cliente.
    """
    print('Child: ', os.getpid())
    time.sleep(5)
    while True:
        data = connection.recv(1024)
        if not data:
            break
        reply = 'Echo => %s at %s' % (data, now())
        connection.send(reply.encode())
    connection.close()


def dispatcher():
    """
    Escucha hasta que el proceso muera, espera la siguente conexión.
    Limpia los procesos children zombies.
    Copia el proceso padre, si el proceso hijo es nuevo lo maneja, si
    el proceso ya existe, lo agrega a la lista `activeChildred`.
    """
    while True:
        connection, address = sockObj.accept()
        print('Server connected by ', address, end=' ')
        print('at', now())
        threading.Thread(target=handleClient, args=(connection, )).start()


dispatcher()
```

socketserver

`socketserver` documentación

Simplifica tareas de escritura de servidores de red, `socketserver` define clases síncronas e asíncronas.

Las clases servidor tienen los mismos métodos y atributos externos, sin importar cual protocolo se esté usando.

```
import time
import socketserver
import socket


myHost = ''
myPort = 50008


def now():
    """
    Tiempo actual del servidor.
    """
    return time.ctime(time.time())


class MyClientHandler(socketserver.BaseRequestHandler):
    """
    Hereda de `BaseRequestHandler`, se debe implementar `handler()`.
    `setup()` y otros métodos es opcional.
    """

    def setup(self):
        self.request.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    def handle(self):
        """
        Implementar `handler()`, maneja todas las nuevas peticiones.
        """
        print(self.client_address, now())
        time.sleep(5)
        while True:
            data = self.request.recv(1024)
            if not data:
                break
            reply = 'Echo => %s at %s' % (data, now())
            self.request.send(reply.encode())
        self.request.close()



myaddr = (myHost, myPort)
# implementa `ThreadingTCPServer` para crear hilo por cada nueva conexión.
server = socketserver.ThreadingTCPServer(myaddr, MyClientHandler)

print(server.socket)

# configuración de parámetros de servidor.
server.socket.listen(5)

# maneja peticiones hasta que terminan.
server.serve_forever()
```

Clases síncronas

Las clases que heredan de `BaseRequestHandler` deben implementar el método `handle()` la cual procesará las peticiones entrantes. Las instancias se deben utilizar con sentencia `with`. `handle_request()` o `server_forever()` permiten procesar una o más peticiones. Para cerrar el servidor se debe utilizar `server_close()` excepto si se usa `with`.

Clase	Descripción
`TCPServer(server_address, RequestHandlerClass, bind_and_activate=True)`	utiliza protocolo TCP, provee streams continuos de data cliente/servidor.
`UDPServer(server_address, RequestHandlerClass, bind_and_activate=True)`	usa datagramas que son paquetes discretos de información que llegan a destino sin orden o se pueden perder en tránsito.
`UnixStreamServer(server_address, RequestHandlerClass, bind_and_activate=True)`	similar a clases TCP, utiliza socket Unix, solo Unix-like.
`UnixDatagramServer(server_address, RequestHandlerClass, bind_and_activate=True)`	similar a clases UDP, utiliza socket Unix, solo Unix-like.

Clases asíncronas

Las clases deben implementar de `BaseRequestHandler` e implementar el método `hendle()`. Recordar que `fork()` es solamente en sistemas Unix-like y es más costoso que `threads`.

Clase	Descripción
`ForkingTCPServer`	implementa servidor TCP usando `fork()` para manejar nuevas peticiones clientes. Unix-like.
`ForkingUDPServer`	implementa servidor UDP usando `fork()` para manejar nuevas peticiones clientes. Unix-like.
`ThreadingTCPServer`	implementa servidor TCP que utiliza hilo (`thread`) para manejar nuevas peticiones clientes.
`ThreadingUDPServer`	implementa servidor UDP que utiliza hilo (`thread`) para manejar nuevas peticiones clientes.
`ForkingUnixStreamServer`	implementa servidor de socket Unix para cada nueva conexión. Sistemas Unix-like.
`ForkingUnixDatagramServer`	implementa servidor de datagramas Unix para cada conexión. Sistemas Unix-like.
`ThreadingUnixStreamServer`	implementa servidor de socket Unix que utiliza hilos (`thread`) para cada nuevas peticiones clientes. Sistemas Unix-like.
`ThreadingUnixDatagramServer`	implementa servidor de datagramas Unix que utiliza hilos (`thread`) para cada nuevas peticiones clientes. Sistemas Unix-like.

```python
from socketserver import ThreadingTCPServer, BaseRequestHandler

class MiRequestHandler(BaseRequestHandler):
    def handle(self):
        # manejar la conexión aquí

# crear un servidor TCP que utiliza hilos para manejar conexiones
servidor = ThreadingTCPServer(('localhost', 8080), MiRequestHandler)
servidor.serve_forever()
```

Clases Mixins

Mixins Asincrónicos

Si las peticiones toman mucho tiempo o tienen alto costo computacional, porque pueden denegar el servicio a nuevas peticiones, se deben usar clases asíncronas como `ForkingMixIn` y `ThreadingMixIn`.

Cuando se hereda de `ThreadingMixIn` para un comportamiento de hilos en paralelo, se debe declarar explícitamente cómo se deben comportar los hilos ante un cierre o fallo abrupto.

Clase	Descripción
`ForkingMixIn`	crea servidores para cada nueva petición (`fork()`). Sistemas Unix-like.
`ThreadingMixIn`	crea servidores para cada nueva petición (`thread`). Sistemas Unix-like.

```python
from socketserver import ForkingMixIn, TCPServer, BaseRequestHandler

class MiRequestHandler(BaseRequestHandler):
    def handle(self):
        # manejar la conexión aquí
        data = self.request.recv(1024)
        print(f"Datos recibidos: {data.decode()}")

class ForkingTCPServer(ForkingMixIn, TCPServer):
    pass

if __name__ == "__main__":
    # crear un servidor TCP que utiliza fork para manejar conexiones
    servidor = ForkingTCPServer(('localhost', 8080), MiRequestHandler)
    servidor.serve_forever()
```

Server Object

BaseServer documentación

Clase base de todos los objetos Server, es una interface para todas las clases del módulo.

Método/Atributo	Descripción
`fileno()`	retorna un entero de descriptor de fichero del socket del servidor. Más comúnmente pasado a `selectors` (este módulo proporciona una interfaz que facilita la escritura de código para aplicaciones que requieren manejo de eventos de E/S multiplexados.)
`handle_request()`	procesa una única petición.
`serve_forever(poll_interval=0.5)`	maneja peticiones hasta terminan.
`service_actions()`	llama un loop `serve_forever()`, puede ser sobre escrito por subclases o mixins para operaciones específicas, como acciones de limpieza.
`shutdown()`	detiene el loop `serve_forever()`
`server_close()`	limpia el servidor. Puede ser sobre-escrito.
`finish_request(request, client_address)`	proceso actual para la petición de instancia `RequestHandlerClass` y su llamado de método `handle()`
`get_request()`	acepta una petición del socket, retorna una 2-tuple (nuevo objeto socket para el cliente y dirección del cliente).
`handle_error(request, client_address)`	llama a `handle()` de instancia `RequestHandlerClass` elevando una exepción.
`handle_timeout()`	función es llamada cuando atributo `timeout` se establece a un valor distinto a `None`.
`process_request(request, client_address)`	llama a `finish_request()` para crear una instancia de `RequestHandlerClass`.
`server_activate()`	llama el constructor del servidor para activar el servidor, por defecto solamente invocan `listen()`. Puede ser sobre-escrito.
`server_bind()`	llama el constructor del servidor para enlazar (bind) el socket deseado. Puede ser sobre-escrito.
`verify_request(request, client_address)`	retorna un valor booleano, si es `True` la petición será procesada, si es `False` no será procesada. Puede ser sobre escrita para controlar el acceso al servidor. Default siempre retorna `True`.
`address_family`	protocolo usado, comúnmente son `socket.AF_INET` y `socket.AF_UNIX`.
`RequestHandlerClass`	clase manejadora de peticiones establecida por usuario, una instancia de esta clase es creada para cada petición.
`server_address`	dirección de escucha del servidor.
`socket`	objeto socket en el cual el servidor escucha.
`allow_reuse_address`	si el servidor permitirá la reutilización de una dirección, defalult es `False`.
`request_queue_size`	tamaño del queue de peticiones, default es 5.
`socket_type`	tipo de socket usado por el servidor, los más comunes son `socket.SOCK_STREAM` y `socket.SOCK_DGRAM`.
`timeout`	duración del timeout, default es `None`.

Request Object

> `BaseRequestHandler` documentación

`BaseRequestHandler`, superclase para todos los objetos requests, es una interfaz que define métodos, en concreto se debe definir `handle()` en las subclases.

Método/Atributo	Descripción
`setupt()`	llamado después de `handle()` para realizar operaciones de inicialización. Por defecto, hace nada.
`handle()`	función en donde se realiza todo el trabajo de manejo de las peticiones, debe ser implementada, varias atributos instancias están disponibles, como `request`, `client_address`, `server`.
`finish()`	llamado después de `handle()` para realizar acciones de limpieza.
`request`	nuevo objeto `socket.socket` usado para comunicar al cliente.
`client_address`	dirección cliente retornado por `BaseServer.get_request()`.
`server`	objeto `BaseServer` usado para manejar la petición.

`StreamRequestHandler` y `DatagramRequestHandler`, son subclases que sobre-escriben los métodos `setup()` y `finish()`, proveen de atributos `rfile` y `wfile`.

Método/Atributo	Descripción
`rfile`	objeto fichero que recibe la petición sea leída. Soporta interface readable `io.BufferedIOBase`.
`wfile`	objeto fichero en el cual se escribe la respuesta. Soporta interface writable `io.BufferedIOBase`.

Elegir esquema de servidor

Dependiendo del requerimiento del servidor se tienen 3 esquemas de construcción, pero esto no imposibilita la combinación de estas estrategias.

1. `fork()` Especial para transacciones largas, especial para procesamientos de larga duración y más allá de solamente pasar datos por los sockets. Se crea una copia para cada nuevo cliente, solamente es posible en sistemas Unix-like, en sistemas Windows se debe instalar Cygwin, pero tiene limitaciones. Bifurcar toma muchos recursos al crear copias del servidor por cada cliente nuevo.

2. `thread` Especial para transacciones largas, especial para procesamientos de larga duración y más allá de solamente pasar datos por los sockets. Por cada cliente nuevo se crea un hilo separado, es multiplataforma por defecto. Es eficiente en el uso de recursos porque comparte el mismo proceso y espacio de memoria con los hilos creados para cada cliente.

3. `select` Especialmente para transacciones relativamente cortas que no son ligados a CPU. Son completamente inmune a bloqueos. Es más complejo que las opciones anteriores, debido a que se debe pasar el control manualmente a todas las tareas. Librería `asynio` simplifica algunas tareas de implementaciones basadas en servidor socket event-loop `select`. `Twisted` ofrece soluciones híbridas.

Sockets Ficheros y Streams

El rol principal de sockets es la interconexión y transmición de data entre máquinas sobre la red, pero esto no quiere decir que se pueda procesar ficheros regulares o pasar un socket a una interfaz o programa que espera un fichero.

`socket.makefile()` permite retornar un objeto *file* asociado al socket, el tipo del objeto retornado depende de los argumentos dados al método, tiene la misma operaciones que `open()`, pero solo soporta los modos `'r'`, `'w'` y `'b'`.

```
socket.makefile(
    mode='r',
    buffering=None,
    *,
    encoding=None,
    errors=None,
    newline=None
)
```

Socket envuelve un objeto retornado permitiendo transferir data sobre el socket asignado con llamados `read` y `write`, en lugar de `recv` y `send`. Métodos built-in `input` y `print` forman parte de este grupo, pudiendo interactuar con sockets.

Puede tener modo bloqueante, tener timeout, pero el buffer del objeto *file* interno puede terminar en un estado inconsistente si ocurre el timeout.

Cerrar el file object retornado, no cerrará el socket a menos que todos los otros file object sean cerrados y se haga el llamado `socket.close()`.

> En Windows, estos file objects no pueden ser usados donde un file object con un file descriptor es esperado, por ejemplo, como argumento de `subprocess.Popen()`.

Permite agregar soporte de interfaces de ficheros a softwares, por ejemplo, al usar métodos `load` y `dump` del módulo `pickle` esperan un objeto con una interfaz file-like, pero no requieren de un fichero físico. Pasar un socket TCP/IP envuelto con `makefile()` a `pickle` permite serializar objetos Python sobre Internet sin tener que serializar series de bytes por nosotros mismos y realizar llamados a sockets. Estas técnicas permiten agregar mayor soporte a variedades de mecanismos de transporte a nuestro software.

Cualquier componente que espere un protocolo de método file-like aceptará un socket envuelto con `socket.makefile()`. Estas interfaces también soportan clases built-in `io.StringIO` y cualquier otro objeto que soporte los mismos métodos de objetos *file*. Especificando el protocolo (objetos interfaces) no especificando los tipos de datos.

Ejemplo de uso

```
import sys
import socket
import time


port = 50008
```

```
host = 'localhost'


def initListenerSocket(port=port):
    """
    inicializa socket conectado a servidor modo listener.
    """
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('', port))
    sock.listen(5)
    conn, addr = sock.accept()
    return conn


def redirectOut(port=port, host=host):
    """
    conecta stream stdout del caller a un socket para GUI para
    empezara escuchar, o falla antes de aceptar.
    """
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port))
    file = sock.makefile('w')
    sys.stdout = file
    return sock


def redirectIn(port=port, host=host):
    """
    conecta stdin stream a un socket dado por GUI.
    """
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port))
    file = sock.makefile('r')
    sys.stdin = file
    return sock


def redirectBothAsClient(port=port, host=host):
    """
    conecta stream stdin y stdout del caller al mismo socket en
    ese modo.
    cliente al servior, envía msg y recibe reply.
    """
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port))
    outputFile = sock.makefile('w')
    inFile = sock.makefile('r')
    sys.stdout = outputFile
    sys.stdin = inFile
    return sock
```

```
def redirectBothAsServer(port=port, host=host):
    """
    conecta stream stdin y stdout del caller a un mismo socket en
    ese modo.
    servidor a cliente: recibe msg y envía reply.
    """
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind((host, port))
    sock.listen(5)
    conn, addr = sock.accept()
    outputFile = conn.makefile('w')
    inFile = conn.makefile('r')
    sys.stdout = outputFile
    sys.stdin = inFile
    return conn
```

Probando la implementación, código completo en `test_socket_stream_redirect.py`.

```
def server1():
    mypid = os.getpid()
    conn = initListenerSocket()
    file = conn.makefile('r')
    for i in range(3):
        data = file.readline().rstrip()
        print('server %s got [%s]' % (mypid, data))


def client1():
    mypid = os.getpid()
    redirectOut()
    for i in range(3):
        print('client %s: %s' % (mypid, i))
        sys.stdout.flush()
```

Consideraciones del código anterior

- Traducción de binario a texto Socket RAW prefieren strings binarios se deben convertir manualmente, pero al abrir un fichero envuelto en modo texto, automáticamente se traduce en strings binarios tanto como input y output, `print()` requiere de ficheros modo texto envueltos para escribir textos.
- Buffered streams, program output, y deadlock Streams son normalmente buffered, y para imprimir el texto se debe realizar `sys.stdout.flush()` en el socket conectado a un stream output. De no ser así, esperarían eternamente generando un punto muerto, errores de lectura de socket. `input()` automáticamente realiza `sys.stdout.flush()`, enviando la data a `print()`. Si se quiere leer el **output** de un programa, el programa debe llamar `sys.stdout.flush()` periódicamente o ejecutar el código usando stream unbuffered con `python -u` si es aplicable. Se puede establecer socket en modo fichero con *unbuffered mode* usando `socket.makefile(mode='r', buffering=0)`, pero no permite ejecutarse con `print`. Socket no bloqueantes con el método `setblocking(0)` solamente evitan el estado para las llamadas de transferencia y no direcciona los fallos de la data producida enviada a un buffered output.

Requisitos de streams

El siguiente código muestra como algunas de estas complejidades aplican a las redirecciones estándares de streams, intentando conectarlas a ficheros modo texto y binario usando `open()` para acceder a estos ficheros, con `print()` y `input()` para redireccionar la data.

```
import sys


def reader(F):
    tmp, sys.stdin = sys.stdin, F
    line = input()
    print(line)
    sys.stdin = tmp


# works: `input()` retorna texto
reader( open('test_stream_modes.py'))

# works: `input()` retorna bytes
reader( open('test_stream_modes.py', 'rb'))



def writer(F):
    tmp, sys.stdout = sys.stdout, F
    print(99, 'spam')
    sys.stdout = tmp


# works: `print()` pasa texto a `.write()`
writer( open('temp', 'w'))
print(open('temp').read())


# fallará en `print()`: requiere bytes modo binario
writer( open('temp', 'wb'))

# fallará en `open()`: texto debe ser unbuffered
writer( open('temp', 'w', 0))
```

Cuando falla en `print()` es porque pasa texto string a un fichero modo binario (nunca permitido para ficheros en general)

Cuando falla en `open()` es porque no se puede abrir un fichero modo texto en modo unbuffered en Python 3.X (texto implica Unicode encoding).

Estas mismas reglas son aplicables a objetos files envueltos creados con `socket.makefile()`, ellos deben ser abiertos en modo texto por `print()` y usados por `input()` quien recibe esos strings. Pero no se puede usar modo text en ficheros unbuffered.

```
>>> import socket
>>> s = socket.socket()
>>> s.makefile('w', 0)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python3.11/socket.py", line 330, in makefile
    raise ValueError("unbuffered streams must be binary")
ValueError: unbuffered streams must be binary
```

Line buffering

Socket modo-texto envueltos tambíen aceptan argumento buffering-mode, 1 para especificar *line-buffering*.

```
>>> import socket
>>> s = socket.socket()
>>> f = s.makefile('w', 1)
>>>
```

Aparentemente no existe diferencia con full buffering, todavía requiere de un fichero para ser manualmente *flushed* para transferir las líneas.

El servidor al ser *fully buffered* espera hasta 15 segundos para mostrar los mensajes y lo hace porque espera que el fichero envuelto salga, al realizar `flushes` manuales en el cliente, en el lado del servidor los mensajes se mostrarán inmediatamente en seguida.

Cuando *line buffered* es necesario, escritura en ficheros socket envuelto y `print()` asociados son *buffered* hasta que el programa termine, *flushes* manuales son requeridos o los *buffer* se mostrarán cuando estén llenos.

Solución delayed outputs y deadlocks

Script que envíen la data para esperar para que el programa imprima en el socket envuelto, pudiendo usar `print()` o `sys.stdout.write()`, en general. Stream buffered y deadlocks son problemas generales que van más allá de ficheros socket envueltos (socket wrapper files).

- Llamar `sys.stdout.flush()` periódicamente para refrescar las salidas impresas tan pronto sean producidas.
- Ejecutar scripts usando `python -u`, si es posible, para forzar la salida stream sea un-buffered, funciona para programas sin modificar generados por herramientas *pipes* como `os.popen`. Y no ayudan en el caso de reiniciar manualmente el fichero stream a un buffered texto de socket envuelto después que inicie el proceso, esto se evita usando `sys.stdout.flush()`.
- Usar `threads` para leer desde sockets para evitar bloqueos especialmente si el programa es un GUI que no depende de un *flush* del cliente. No arregla realmente el problema, pero el GUI se mantiene funcionando (recordar que generar hilos para lectura también se pueden bloquear o llegar a un punto muerto).
- Implementar socket wrapper object personalizados que intercepten el texto en llamados `write`, lo codifiquen en binario, y lo enruten a un socket con llamado `send`. `socket.makefile()` es la herramienta para ello.
- Saltar `print` y comunicarse directamente con interfaces nativas de dispositivos IPC, como métodos `send` y `recv` de objetos sockets RAW que transfieran la data inmediatamente y esperen que se llene el buffer de data como hacen los métodos de *file*. Transmitir strings bytes simples usando módulo `pickle` los métodos `dump()` y `loads()` para convertir objetos Python a string de bytes y transferirlos al socket.

`python -u` funciona incluso cuando existe probemas de buffering para programas Python 3.X, no se requiere reiniciar el stream en otros objetos en programas generados como redirección socket, en este caso, flushes manuales o reemplazo de socket wrapper son requerido.

Buffered y pipes

Usar pipes no almacena completamente su salida cuando está conectado a una terminal (la salida solo es *line-buffered* cuando se ejecuta desde un símbolo del sistema de shell), pero lo hace si está conectado a otra cosa (incluido un socket o tubería).

Ficheros modo texto son requerido en Python 3.X, `python -u` suprime completamente output buffering stream.

- *pipe_unbuffered_writer.py*

```
import time
import sys

for i in range(5):
    print(time.asctime())
    sys.stdout.write('spam\n')
    time.sleep(2)
```

- *pipe_unbuffered_reader.py*

```
import os
import sys



def buffered_output():
    """
    la salida tarda 10 segundos en mostrarse en la terminal.
    """
    for line in os.popen('python pipe_unbuffered_writer.py'):
        print(line, end='')


def unbuffered_output():
    """
    la salida se muestra cada 2 segundos en la terminal
    (por el time.sleep de writer)
    se muestra a tan pronto se envía la data.
    """
    for line in os.popen('python -u pipe_unbuffered_writer.py'):
        print(line, end='')



if __name__ == '__main__':
    opt = int(sys.argv[1])
    if opt == 1:
```

```
        buffered_output()
    elif opt == 2:
        unbuffered_output()
```

Al ejecutar *pipe_unbuffered_reader.py 2* en la terminal se demora 10 segundos en mostrar los mensajes debido a que está llenando el buffer.

Al ejecutar *pipe_unbuffered_reader.py 2* se muestra cada dos segundos el mensaje desde **pipe_unbuffered_writer.py* por el *time.sleep(2)*, al no usar un timeout se muestra inmediatamente luego de enviarlo.

Para redirecciones de socket, `sys.stdout.flush()` y reemplazo de socket wrappers son requeridos.

Sockets versus command pipes

- **Sockets** Cuando se quiere tener independencia y usar comunicación sobre la red. Se puede iniciar un cliente y servidor independientemente, el servidor se puede ejecutar contínuamente para servir a clientes múltiples. Conexión a máquinas remotas.
- **Pipes** No están pensados para ejecutar modelo “cliente/servidor, no se ejecutan perpetuamente, la desventaja radica en generar las relaciones directas, pipes no soportan largas vidas o accesos remotos. Named pipes (fifos) accedido con open soportan independencia de clientes y servidor, pero a diferencia de *sockets*, están limitados a la máquina local y no son soportadas por todas las plataformas.

`print()` y `input()` están pueden ser enrutados sobre socket de red o máquina local y con cambios mínimos se pueden usar sin socket.