

## PROGRAMACIÓN Y REDES

Diferencia entre dirección IP física y lógica, depende de dónde la mires, si estás mirando dirección IP + puertos (físicos), estamos en la capa de aplicación

Dirección IP lógica, la que eliges tú,

Ventajas e inconvenientes entre los mensajes de texto y mensajes binarios


Mensaje texto: inconveniente: tienes que llegar a un acuerdo si el texto es válido o inválido, inconveniente, como sabes si el texto enviado es correcto (defines una sintaxis (estructura), con una gramática, mediante parsers de texto)

Ventajas,

Mensajes binarios: inconvenientes,

Ventajas, nos protege si el canal no es perfecto, da igual escribir más

Como mínimo para poder intercambiar datos



### Formatos de representación

- » Para la transmisión de formatos binarios tanto emisor y receptor deben coincidir en la **interpretación** de los bits transmitidos
- » Problemática: hay 3 cuestiones básicas que deben acordarse, y son:
  1. **Tamaño** de los datos numéricos
    - > 16 bits vs 32 bits
  2. **Ordenación** de bytes (*endianness*)
    - > *little-endian* vs *big-endian*
  3. **Formatos** de texto
    - > ASCII vs Unicode

25-ene.-23Programación en Red / Entornos Distribuidospágina 12

12

Formatos de texto en utf-8

Endianness. Si vas a intercambiar datos por TCP/IP le interesa saber tu dirección tu dirección IP está formada por 4 octetos de bytes, tienes que definir la endianness, normalmente, se escoge el big endian, ya que en los 80 los procesadores tenían big endian (menos Intel), tenemos una red definida como big endian formada por hosts que utilizan Little endian,

Si vas a transmitir caracteres te tienes que poner de acuerdo con el formato (unicode, utf-8..) y si vas a transmitir en binario tienes que ponerte de acuerdo en la endianess



## Relaciones entre capas

» Cada capa usa a la que tiene directamente debajo

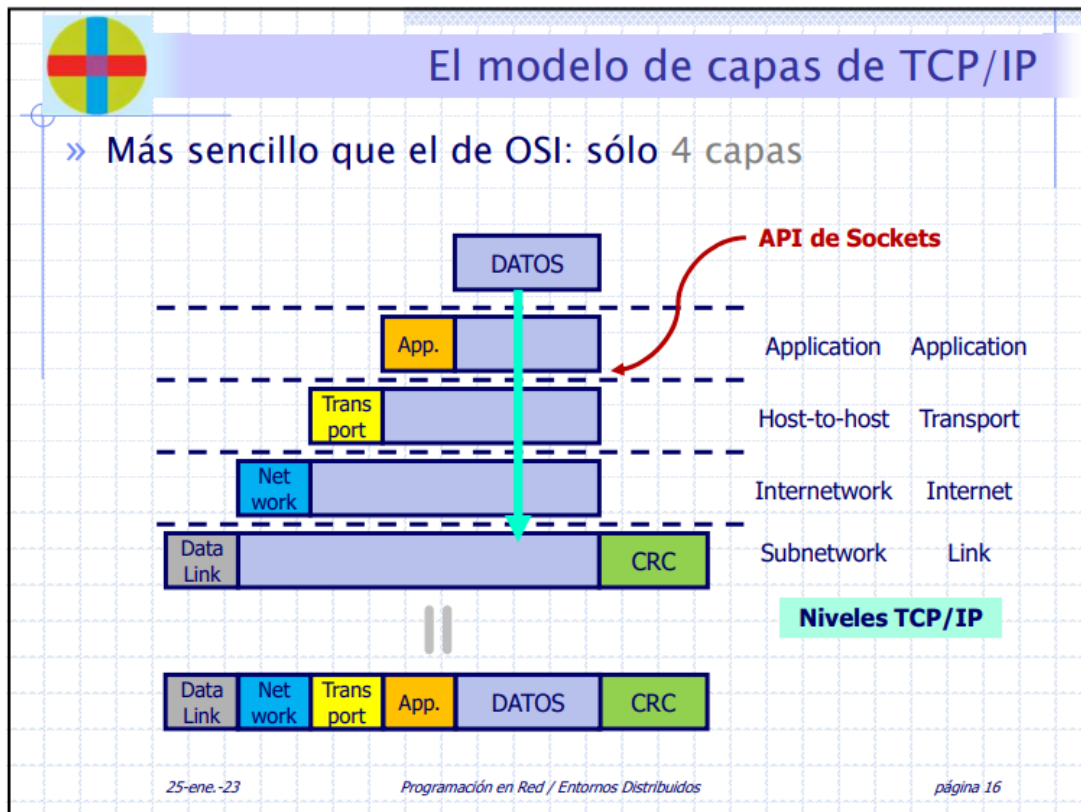
- La capa inferior añade cabeceras a los datos que recibe de la capa superior
- Parte de los datos de la capa superior pueden ser cabeceras de capas aún más altas



25-ene.-23 Programación en Red / Entornos Distribuidos página 15

Vas bajando y se añade cabecera en función de la info que quiera proporcionar cada nivel tcp/ip

Transporte: la cabecera que se añade tiene dentro info acerca del puerto,



Subnetwork: la trama,

Las rayas son interfaces, la primera es la interfaz de usuario,

Puerto: te permite distinguir proceso de una maquina, una máquina tiene normalmente 220 procesos o así,

Para subir a la capa de aerba necesitas undriver para enviar y recibir, para integrarlo dentro de tcp/ip,

La interfaz entre interntenetwork y subnetwork,Lo que hace es conectar con varias tecnologías

IP, encaminamiento,

Garantia de ip, besteffort, no da ninguna, perdida de fallos, duplicación, que lleguen en desorden

Udp, pone y quitas los puertos, es un mux de paquetes ip, funciona bien cuando las cosas a transmitir son sencillas

Tcp, conmutación de circuitos, conexión virtual entre los extremos, (ventana deslizante), reordena paquetes,

Red ip privada, se utilizan para cuando no pongas la ip publica, ya que si la pones significa que tu casa es internet, es decir, que si quieres acceder al mismo recurso de internet que has establecido como tu dirección ip no podrás


## TUBERÍAS Y FIFOS

No existe ningún unix que no tengan tuberías, salvo versiones anteriores a 1973,

Sistema de memoria virtual, traduce las direcciones de memoria, en unix la memoria es privada, no se puede utilizar para transmitir datos, para ello se crearon las tuberías, es una ola de tipo de fifo, además permite la transmisión fiable (garantía de que llegan los datos, si metes 5 números transmite 5 números), chorro de bytes (que es continuado, no tienes forma de saber todo lo que hay, si tienes una tubería, si metes 3 cosas, el otro extremo solo tiene que leer una vez, no 3,)

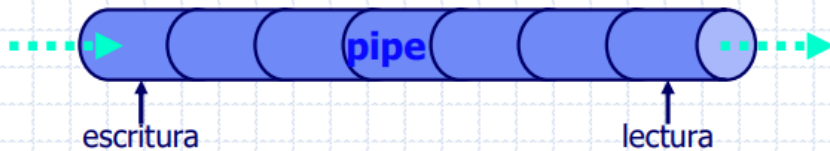
Halfduplex (solo puedes leer o escribir, no puedes hacer las 2 cosas a la vez)

Presentan un tamaño máximo fijo para el buffer (zona de almacenamiento temporal, que luego se borra todo el tiempo), coges un cacho de memoria, y esta memoria es de una tubería, cuando quieras enviar algo lo que haces es acceder al kernel, es un cacho de memoria que tiene el kernel, escribir dentro del kernel con llamadas al sistema, y luego lo lees con otra llamada al sistema,



### Gestión de las tuberías

- » La **gestión de las tuberías** está integrada en el sistema de ficheros.
  - Implementadas clásicamente en el sistema de ficheros.
  - Posteriormente como un caso particular, mediante *sockets* (4.3 BSD) o *STREAMS* (SVR4).
- » Constituyen un **canal de comunicación**:
  - Los datos escritos en un extremo del canal se leen en el otro extremo.
  - La tubería usa un buffer que define su tamaño:



25-ene.-23      Programación en Red / Entornos Distribuidos      página 3


Si quieres enviar un mensaje a alguien en una tubería de 7 bytes, es una implementación de un buffer memoria circular que se escribe en un lado y se lee en otro, en una tubería se escribe y se lee igual que en fichero en unix (con llamadas al sistema, open (si el open funciona, el proceso tiene un file descriptor), read write (se escribe bytes), close)

Unix. Abstracciones (una para guardar info, el fichero (hay 3, ficheros, directorios (guardan nombres de otros ficheros y sirven para llegar a otros ficheros) y especiales (representa cosas de E/S (un usb, el ratón, el teclado..)) dispositivos modo bloque y modo carácter, diferencia (bloque, debajo tiene la block buffer cache))) y otra para hacer cosas (el proceso)

Cache. Esta cacheando algunos sectores de los discos (estos se guardan dentro de la zona de memoria del kernel), los más utilizados, y los guarda en la RAM, cuando tenía que leer muchos los guardaba,

» La lectura/escritura se realiza mediante un chorro de bytes sin ninguna estructura. · La lectura de los datos es independiente de la escritura. · Permite leer de una vez datos escritos en varias ocasiones., lo mismo que un fichero, pero ahora con tuberías,

Cuando haces una llamada al sistema se lo pide al kernel, porque tú no tienes acceso a esas cosas, y el kernel le dice que esperas, en un estado, en concreto, dormido (hasta que termine de escribir), y el kernel luego lo despierta hace lo que le ha mandado, y pasa a la cola de los activos, el read funciona igual, y con una tubería, lo mismo, si ocupa el tamaño de la tubería, utilizas otra llamada al sistema para poder seguir escribiendo en dicha tubería aunque no tenga almacenamiento, le dices que se duerma, la lectura es independiente de la escritura, se lee y escribe todo lo que se puede,



## Creación de una tubería

» La llamada al sistema `pipe()` sirve para crear una tubería:

```
import os  
r, w = os.pipe()
```

- `w` es un descriptor de fichero de escritura
- `r` es un descriptor de fichero de lectura
- Lo que se escribe en `w` se lee en `r`

- Lanza una excepción si hay error
- En caso de error se fija *errno* con el valor adecuado:
  - *EMFILE* – el proceso tiene demasiados descriptors abiertos
  - *ENFILE* – el sistema tiene demasiados descriptors abiertos

25-ene-23 Programación en Red / Entornos Distribuidos página 5

Es una llamada al kernel, pipe es una llamada al sistema, la escritura en unix con Python se hace importando os, la sintaxis es universal, es decir, es como se accede al kernel.,

`r, w = os.pipe()`, con esto creas una tubería, si hay un error, al `errno`, esto depende del so,

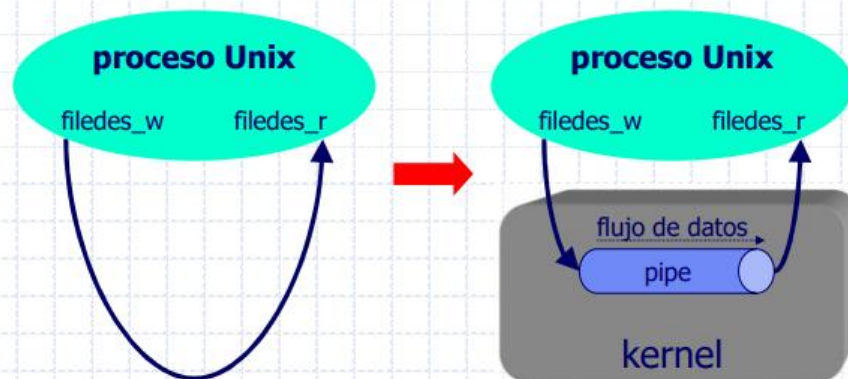




## Tuberías (pipes)

» La llamada `pipe()` crea *dos* descriptors de fichero. ¿Por qué?

» Respuesta:



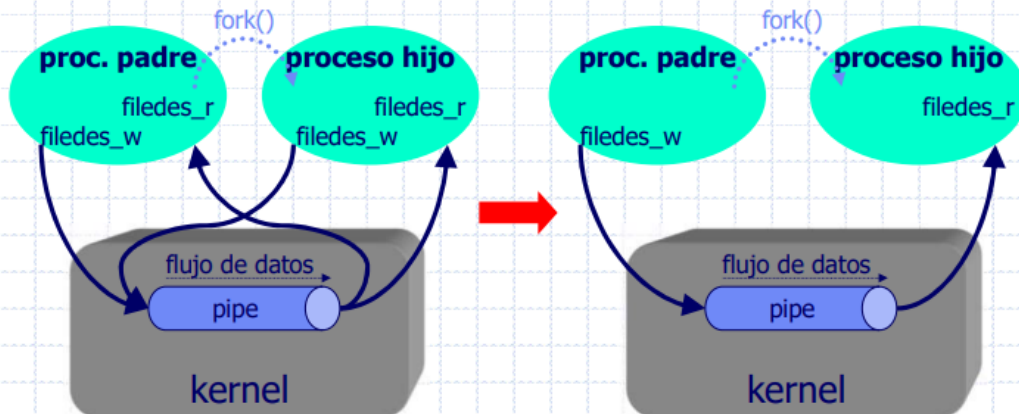
En el kernel se crea una zona de memoria, un extremo del pipe se conecta al descriptor de escritura y otro al de lectura, conecta el mismo proceso Unix el cual tiene dos descriptors de fichero,

Si solo tienes un descriptor e intentas meter cosas a una tubería,

## Procesos y tuberías

» Normalmente, un proceso crea una tubería y luego hace un `fork()`

- Si se quiere comunicación de **padre a hijo**, se cierra el descriptor de lectura del padre y el de escritura del hijo.
- Si se quiere comunicación de **hijo a padre**, se cierra el descriptor de escritura del padre y el de lectura del hijo.



Las tuberías solo funcionan entre procesos distintos si ambos tienen un padre común, en la diapo, hace un close de los de la izq, para comunicar la escritura del padre con la lectura del hijo,



## Entrada/salida en las tuberías

### » La tubería usa un **buffer**:

- Cuando el buffer está **lleno**, `write(fd_w...)` se bloquea.
- Cuando el buffer está **vacío**, `read(fd_r...)` se bloquea.
- Si se intenta escribir cuando el extremo **lector** ha **cerrado** se genera `SIGPIPE`.
- Cuando se **cierra** el extremo **escritor**, se recibe un `EOF` (*end-of-file*) tras la recepción de los últimos datos.

### » Para comunicación *full-duplex* tendríamos que usar dos tuberías.

### » Otra posibilidad útil:

- El hijo hace un `exec()` para ejecutar un programa.
- Pero antes el hijo conecta `fd_r` a `stdin`, con lo que el padre puede enviar datos a la entrada standard del programa.
- Ejemplo...

Condición final de lectura, cuando te devuelve si ya no queda nada por leer 0, como ha 2 procesos, cuando el escritor hace un close, el read ya sabe que ya ha terminado porque el extremo ha devuelto un 0, si ocurre al revés, es decir, que el extremo del lector ha cerrado antes, se genera un SIGPIPE, cuando lo recibe dicho proceso se muere,





## Conceptos básicos de las tuberías

- » El **tamaño del buffer** de una tubería es **finito**, es decir, sólo pueden escribirse una cierta cantidad de bytes en la tubería (hasta que no se lean).
  - El tamaño máximo fijo para el buffer es típicamente de **512 bytes**, que es el tamaño mínimo definido por POSIX.
  - Una ventaja de esto es que los datos raramente llegan a escribirse en el disco, sino que quedan en memoria (en la *block buffer cache*).
- » Las tuberías sólo pueden ser usadas entre procesos que tienen un **padre común**.
- » Un proceso creado con `fork()` hereda todas las tuberías abiertas que tenga su padre.

7

El tamaño de la stuberias, 512 bytes(es el sector de un disco),



## Entrada/salida en las tuberías

- » La tubería usa un **buffer**:
  - Cuando el buffer está **lleno**, `write(fd_w...)` se bloquea.
  - Cuando el buffer está **vacío**, `read(fd_r...)` se bloquea.
  - Si se intenta escribir cuando el extremo **lector** ha **cerrado** se genera `SIGPIPE`.
  - Cuando se **cierra** el extremo **escritor**, se recibe un `EOF` (*end-of-file*) tras la recepción de los últimos datos.
- » Para comunicación *full-duplex* tendríamos que usar dos tuberías.
- » Otra posibilidad útil:
  - El hijo hace un `exec()` para ejecutar un programa.
  - Pero antes el hijo conecta `fd_r` a `stdin`, con lo que el padre puede enviar datos a la entrada standard del programa.
  - Ejemplo...


9

A que velocidad va, vas a la velocidad mas lenta, las tuberías son autosincronizantes, es mejor que el buffer sea pequeño, para adaptarse a la velocidad de los extremos,

Who(busca procesos) | sort(los ordena) | lpr(imprime), es decir, imprime la lista ordenada de los procesos del sistema,

Coencta la salida de who con la entrada de sort,

Riesgo de seguridad, las tuberías solo los puedes utilizar tu y tus hijos, entonces son inviolables,



## Ejemplo de código: tuberías

### *código Python 3*

```
import os, sys, time

rd, wd = os.pipe()          # son file descriptors, no file objects
r, w = os.fdopen(rd, 'rb', 0), os.fdopen(wd, 'wb', 0) # file objects
pid = os.fork()
if pid:                     # padre
    w.close()
    while True:
        data = r.readline()
        if not data:
            break
        print("el padre lee: " + data.decode('utf8').strip())
else:                       # hijo
    r.close()
    for i in range(10):
        mensaje = "línea %s\n" % i
        w.write(mensaje.encode('utf8'))
        w.flush()
        time.sleep(1)
```

25-ene.-23Programación en Red / Entornos Distribuidospágina 11

11

Una tubería se maneja como un fichero, hay un proceso hijo que escribe 10 líneas cada segundo y el padre las lee,

Limitaciones, solo puedes esto entre parientes(es la más restrictiva), es unidireccional, el tamaño del buffer es fijo, (cuanto más grande sea, es peor, ya que se puede dar el caso de que el otro proceso no lo lea y estes perdiendo tiempo)

Vamos a hacer que las tuberías parezcan ficheros, una vez que el pipeline() FIFO parezca un fichero, lo abres, escribes y lo cierras, lo único que tienen que saber es como se comparte el fichero

Pregunta abierta: Las FIFOs son menos seguras que las tuberías clásicas. ¿Por qué?

Le pones permisos al fichero para que no se meta cualquier proceso, el sistema de permisos de Unix es un sistema cuyo objetivo es la protección, diferencia entre seguridad (limitar los procesos comunes, borrar ficheros) y protección (evitar que nos hagamos daño inadvertidamente), no es recomendable utilizar un usuario root, ya que no le afecta el sistema

de protección, ya que dicho sistema evita que te hagas daño a ti mismo, es mejor usar un usuario normal.

El sistema de protección de unix no es un sistema de seguridad,

Mkfifo(), es mkfifo para crear un fifo, para el pipeline, no tiene tamaño inicialmente,

Si haces cat < f, estas activando el proceso lector, se ha quedado pillado,

Cat > f para escribir ,

Control +d , no matar ni dormir el proceso,

Abres dos terminales,

Si ejecutas tres terminales con el mismo f, va a ver balanceo de carga(no exactamente), es porque es una cola, cuando le envia la escritura a uno, se sale de la cola y va al otro proceso, por eso lo va alternando, depende de como este la planificación de la cola, es insensato hacerlo, si escribes dos ficheros en un pipe, lo va escribir a cachos, esos cachos son el tamaño de la tubería, el tamaño no aumenta a media que vas escribiendo, por la parte del principio de ellos permisos

Ls -lai f, te da el inodo,

Utilizar la llamada mkfifo,

```
import os, fcntl
flags = fcntl.fcntl(fd, fcntl.F_GETFL)
flags |= os.O_NONBLOCK;
try:
    fcntl.fcntl(fd, fcntl.F_SETFL, flags)
except IOError:
    print "error en fcntl"
```

Le añadimos un flag llamada modo no bloqueante(O\_NONBLOCK), si escribes a alguien que no lee te duermes, o sea que pongas el flag y el write te devuelve un error ya que no esta preparado,

La constante PIPE\_BUF define el número máximo de caracteres que se pueden escribir en una FIFO atómicamente, es decir, los cachitos que puedes escribir, si pones esto y un nonblock, no puedes escribir ya que si escribes 2k y solo entran 1k, te da error,



## Ejemplo de aplicación de FIFOs

» Ejemplo de aplicación: Un servidor y varios clientes que se comunican con aquel mediante una **FIFO** de nombre conocido.

- Los clientes escriben sus peticiones en la FIFO.
  - › Las peticiones deben tener una longitud inferior a `PIPE_BUF` (que se encuentra en el módulo `select`) para que las escrituras sean atómicas y no se produzcan solapamientos.
- Pero, ¿cómo responde el servidor a los clientes?

» Una solución es que cada cliente envíe su PID en la petición.

- El servidor crea entonces una FIFO específica por cada cliente, con un nombre basado en su PID.
- El servidor escribe la respuesta en esta FIFO. El lector, que conoce su nombre, lee la respuesta.

Como contestas peticiones de los clientes?,

Hace falta otra fifo una fifo peticiones y otra fifo respuestas, es decir, habría que quedar una fifo de respuestas de cada cliente, y esta fifo la crea el cliente antes de realizar la petición

Cuántas llamadas al sistema hacen falta para montar esto??,

Mkfifo f

2 open, uno para leer y otro para escribir,

Pregunta abierta: ¿Por qué una FIFO se abre en modo de sólo lectura o escritura pero no en modo lectura/escritura?

Lo que estas escribiendo se te va venir encima, porque las fifo son unidireccionales, es decir, conexión bocabuclo, si quieres tener una comunicación bidireccional hacen falta dos fifos