*-Clase del 15-09-2021. “Sockets”*

Hasta acá vimos capa de enlace, todo lo que es el entramado, y en redes que no son PPP, como compartir el medio (MAC).

Arriba de esto, vimos que, si necesitamos interconectar mas de una red, hay que poner una capa adicional de abstracción, la capa de red, donde vimos distintos servicios (con conexión y sin conexión), como ejemplo vimos IP.

Hasta acá, podríamos encaminar un paquete desde una maquina a otra que estuvieran en distintas redes interconectadas.

El tema es que necesito que los procesos se comuniquen, entonces la manera de que los procesos se intercomuniquen es colocar otra abstracción sobre la capa de red, la capa de transporte, que me permite multiplexar distintas direcciones de transporte (puertos) en la misma dirección de red (IP).

Hasta tenemos lo que se refiere conceptualmente a la pila de protocolos.

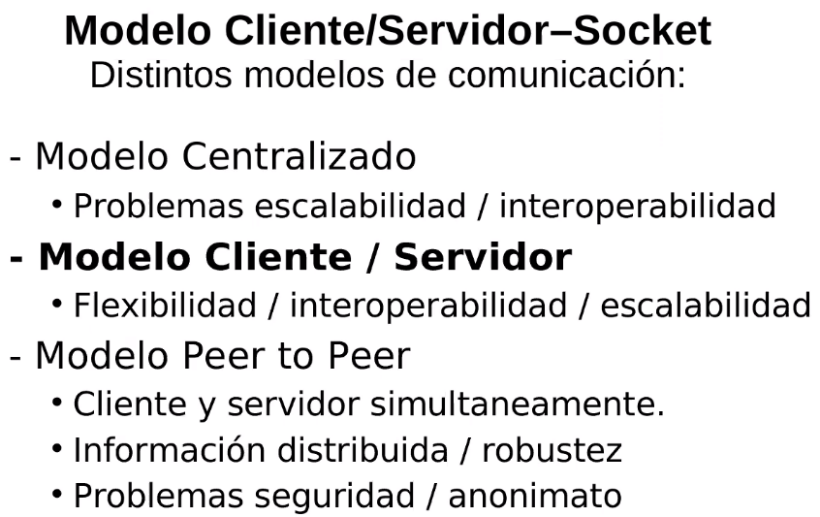
Ahora vemos ¿Cómo desde mi proceso puedo usar esa RED? ¿Cómo hago? ¿Qué hago?

¿Crea el usuario todo? NO, esto lo genera el SO, el proceso tiene que realizar un pedido, una llamada a sistema.

Hay una API, Interfase de Programación de Aplicaciones, que me permite pedirle al SO que se encarga de armar las tramas, paquetes, segmentos y todo el encapsulado. Desde el punto de vista del programador no veo nada de esto.

Esto se llama API de Sockets.

Vemos un par de cosas antes de arrancar con la API de sockets.

**Repasamos modelo cliente servidor:**

Antes de que existieran las redes, normalmente todo era un modelo centralizado donde todo el procesamiento lo realizaba la unidad central, o mainframe, y el problema que tenía, es que era un cuello de botella, porque si esa máquina se rompía no funcionaba mas nada.

En el modelo Cliente servidor se dividen el trabajo entre las distintas maquinas, hay una o varias que pueden tomar el rol de servidores para los clientes.

Peer to peer, es un modelo cliente servidor donde se intercambian los roles en cada momento. Hacemos hincapié en este modelo porque Internet funciona de esa manera.

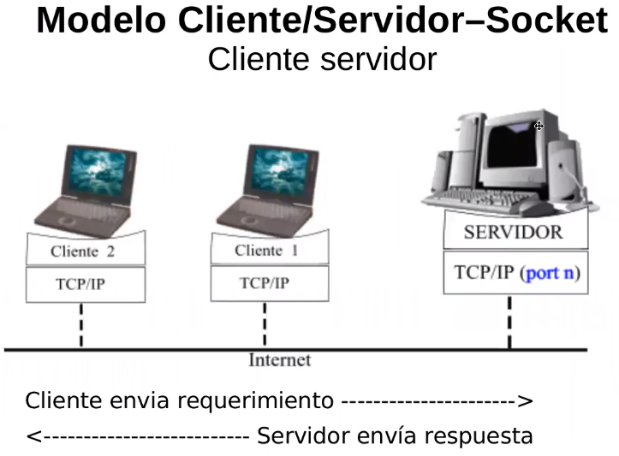
IMPORTANTE: Primero y fundamental, cliente y servidor son APLICACIONES ósea procesos o programas que yo escribo.

EJ: podría ser una maquina de poca potencia, un celular, una rasberry …. No interesa la potencia sino la aplicación que tiene.

Servidor: Aplicación que tiene la información o los datos.

Cliente: Aplicación que solicita la información.

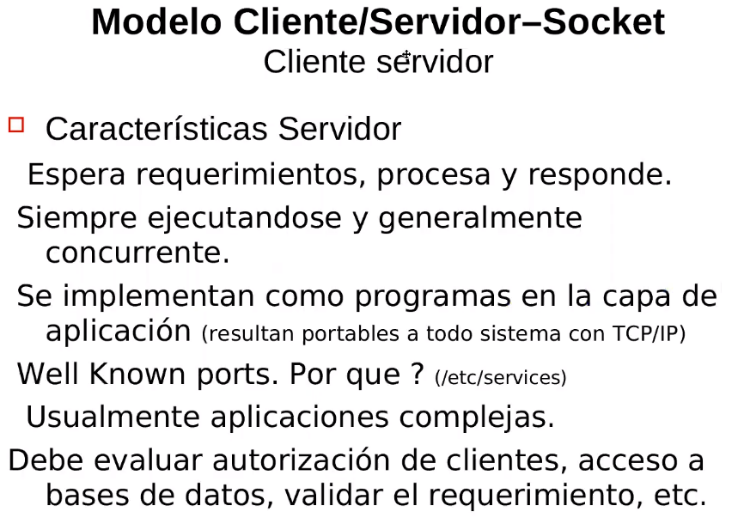
Protocolo: Si el cliente y el servidor hablan distinto protocolos, no hay forma de comunicarse.

¿Cómo funciona esto?

Normalmente el cliente, que es un proceso que corre en la Notebook, realiza una llamada a sistema, por lo que inicia un requerimiento y se lo pide al servidor.

Los datos se encapsulan en todas las capas que hemos visto, se envían, se desencapsulan al llegar al servidor donde hay una aplicación corriendo donde van a parar los datos. Entonces es como si “virtualmente” los datos del proceso cliente hablaran directamente con el proceso servidor.

Cliente inicia la comunicación con una solicitud, y normalmente el servidor responde.



**Características del proceso servidor:**

-Siempre está a la espera, ejecutándose. Porque no sabe en que momento le van a pedir datos.

-Usa Puertos bien conocidos:

-Normalmente es una aplicación compleja, porque debe hacer varias cosas: autenticar clientes, acceder a datos, validar requerimientos, atender a varios clientes a la vez, etc.

**Características del proceso servidor:**

-Se ejecutan nada más cuando se hace el requerimiento. Ej: navegador, o cliente de correo Outlook

-Usan algún puerto aleatorio cualquiera, asignado por el SO.

-Como el cliente inicia la comunicación debe conocer la IP del servidor o su nombre, y también el puerto o el servicio. Ej: servidor de correo, o servidor web

Analogía de porque los servidores, tienen “los puertos bien conocidos”

Supongamos que somos un cliente y queremos consumir un servicio. Tenemos la dirección San Martin 1266 (equivalente a la IP)

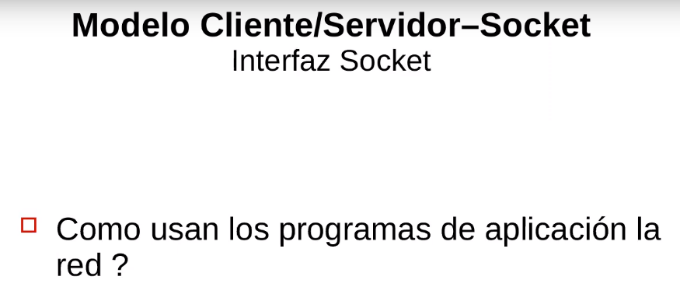
Busco un libro en un local de una galería de 65 mil (2^16 cantidad de puertos) locales. No seria optimo preguntar uno por uno. Pedir información sin hablar con el proceso apropiado.

Podría haber un índice, tipo cartel en la entrada, entonces uno busca en el índice para ver en que local esta. “Port Mapper”. NO se usa demasiado.

¿Hay que explicarlo?

La forma más común que siempre se ha usado es, asignarle a cada “servicio” o “protocolo de capa de aplicación”, un puerto estándar y previamente definido.

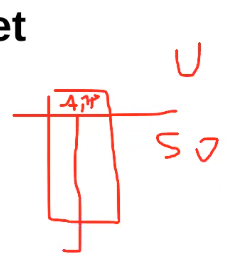
Ej: si yo cliente me quiero conectar a un servidor WEB, en el puerto destino tendría que poner 80 y en el origen poner un puerto de numero alto, mayor a mil, que ninguna aplicación este usando. (Puedo poner también el 80 como origen, pero ningún otro proceso lo podría utilizar, entonces no podría tener un “proceso servidor” corriendo en mi máquina, o el 25 de SMTP para recibir correos).

La forma más fácil es prestablecer, que siempre en las galerías, el local 40 es una librería.

**API Sockets:**

Un programa utiliza interfaz de sockets para poder usar la red.

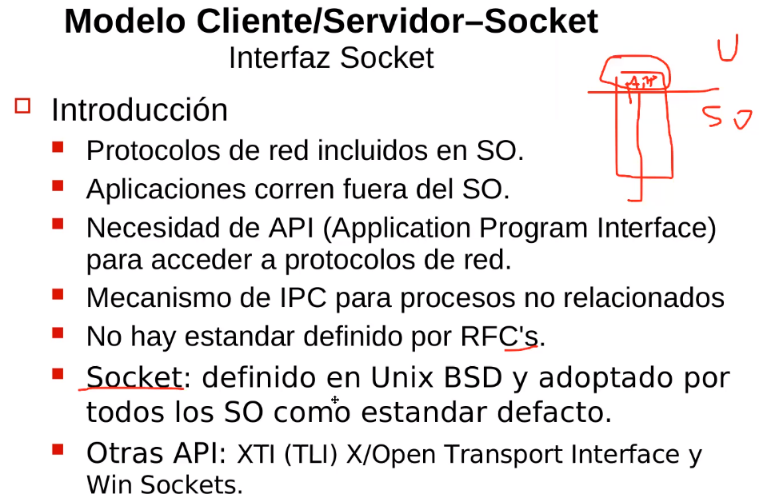
Socket: es como un enchufe, me permite, a la aplicación, enchufarla a la red.



EL SO, implementa toda la parte del Stack de capa de enlace, de red y de transporte. Por lo tanto, queda fuera, la última parte, la capa de aplicación.

* Es un IPC para procesos no relacionados.
* Para utilizar la RED no hay ninguna especificación, ninguna RFC’s, para cada uno de los protocolos sí. Hay un estándar que, si no lo uso, el otro extremo no entiende lo que le mando.

Pero para la forma de interactuar con el SO, no hay un estándar.

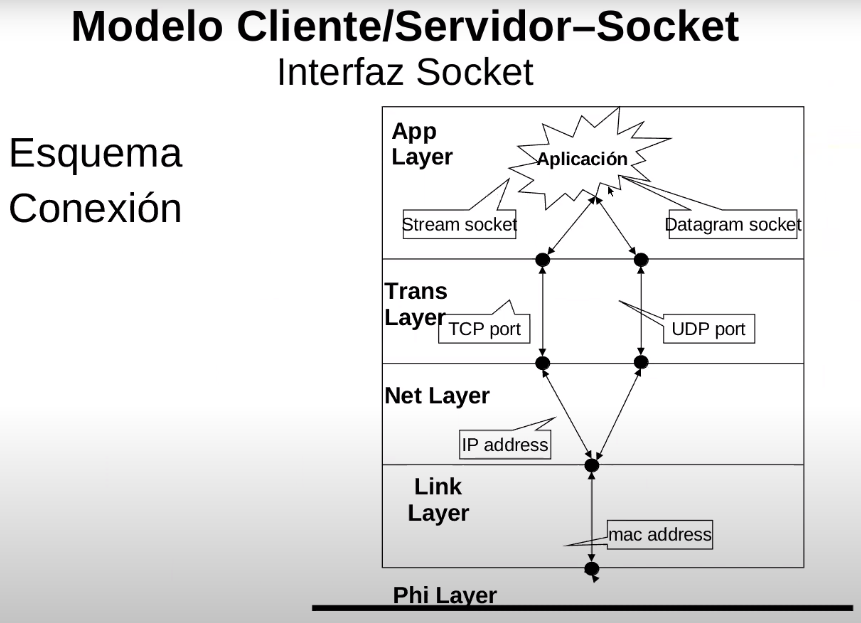


¿Qué es RFCS en redes?

El Request for Comments (RFC) es un documento numérico en el que se describen y definen protocolos, conceptos, métodos y programas de Internet.

-El estándar defacto es Socket, creado por la universidad Berkley a fines de los 70s e implementado en Unix BSD. Después todos los SO lo adaptaron. Hasta Windows

-Hay otros muy poco usados. Casi nadie.

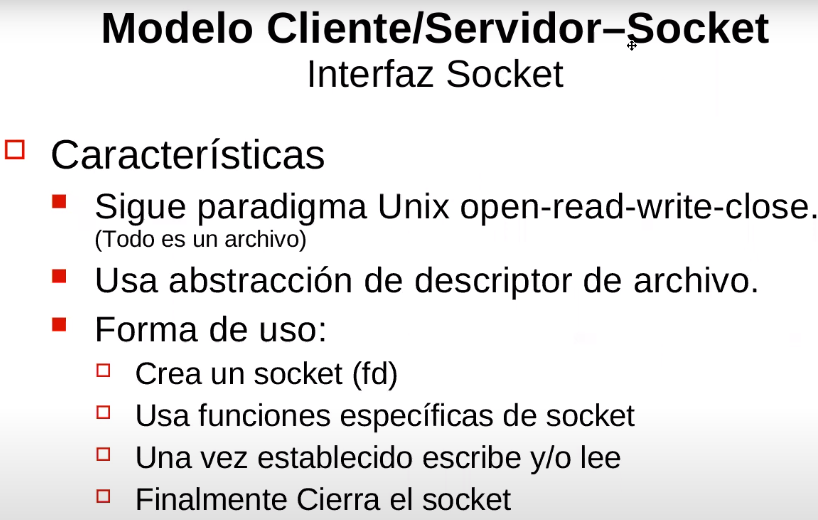
**¿Cómo funciona esto?**

La aplicación podría decidir utilizar UDP o TCP.

Cuando la aplicación utiliza TCP e IP, crea un socket que se llama de “Stream” (flujo de bytes, similar a una tubería, entran en un orden, salen en el mismo orden, no se mesclan, no se pierden).

La otra alternativa es utilizar un socket del tipo “Datagram”, que básicamente va sobre UDP, entonces se pueden perder, llegar en distinto orden, llegar duplicados y ser descartados por el receptor.

Como fue creado en Unix, básicamente usa el mismo paradigma que usa Unix para todo. En Unix todo es un archivo.

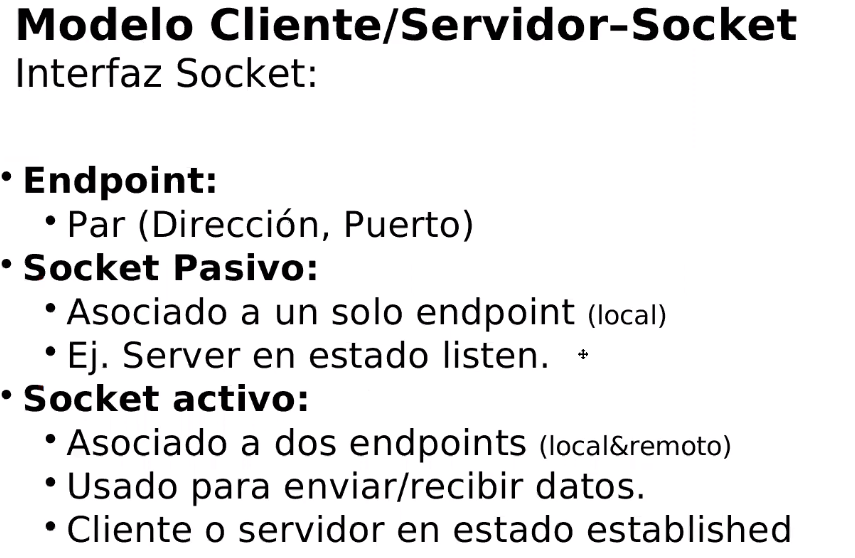
-Abrir: un descriptor de archivo, esta ves uno de socket no uno común. Son archivos especiales.

-Configurarlo: Antes de leer o escribir, hay que configurar algunas funciones específicas para ese socket. Ej: si usa TCP o UDP

-Leer o escribir.

-Lo cierra cuando no lo usa más.

Lo mismo de siempre, pero con un socket.

**Algunas cosas a recalcar:**

Un par de conceptos.

-Endpoit, punto extremo: Se refiere a la dirección y el puerto, si voy a conectar mi cliente con un servidor, al cliente le tengo que dar los datos del servidor. Ósea darle el Endpoint remoto que sería la dirección a la cual me quiero conectar y el puerto.

Ej: no sabe que servidor web, o que servicio quiere consumir.

-Por otro lado, vamos a ver que cuando creamos los sockets, pueden estar de 2 formas. Activo (o conectado) o Pasivo.

La diferencia es, que uno activo tiene los 2 extremos o endpoint. El local y el remoto asignados, con un valor. También se le dice “conectado”, porque cada ves que se envía un paquete tiene IP y puerto, local y remoto, estos 4 valores de un Socket Activo me van a definir o determinar específicamente la IPC entre que procesos de que maquinas van a intercambiar datos. Esto normalmente lo usa un cliente.

En cambio, si fuera un servidor, tengo la IP y puerto local, pero no se quien se va a conectar. Por lo que es pasivo. Si alguien se contacta a esta IP y este puerto, el SO le pasa esos datos a la aplicación para que le pueda responder, recién ahí conoce la IP y puerto remoto del cliente. Ósea recién ahí pueden intercambiar datos.

Crear un servidor lleva mas pasos, porque no tiene de entrada todos los datos.

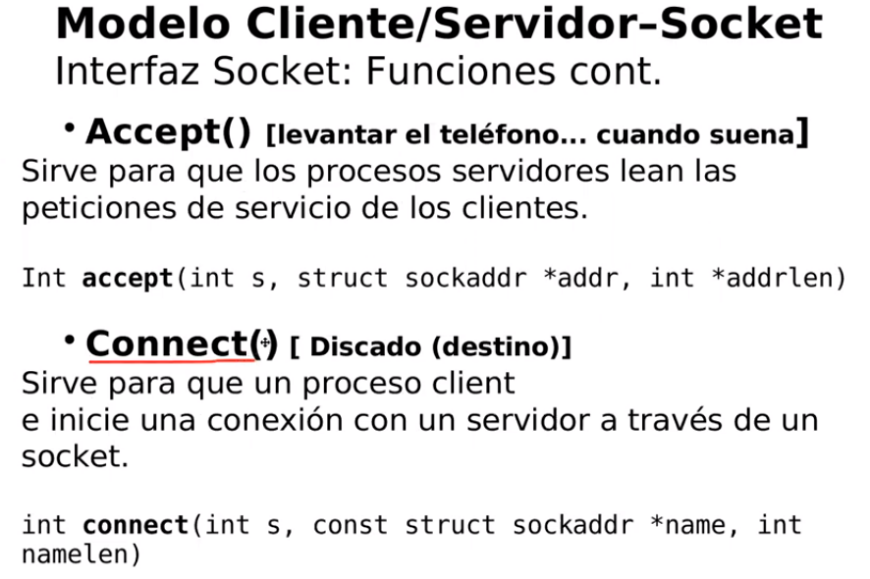
**Llamadas al sistema para utilizar sockets.**



**Socket():** Me devuelve un entero que es un descriptor de

archivo.

Para un servidor, una ves que creo el sockets, debería decirle a que puerto local lo voy a asociar. Bind es ligarlo, atarlo a un puerto. Este Socket es PACIVO.

Después tengo que crear un buffer en el Kernel con **Listen().** Para una lista de “n” nuevas conexiones pendientes. Porque una vez que un cliente se conecta, se crea un nuevo socket, no el que asocie recién y estaba esperando. Esto es totalmente asincrónico. Mi proceso no siempre se ejecuta.

**Accept**() es bloqueante.

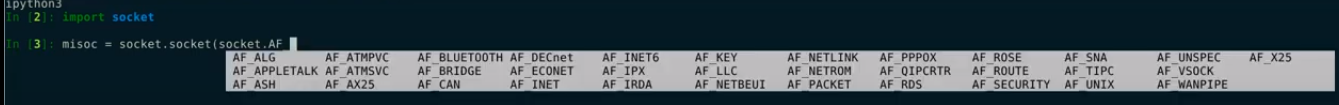
Obviamente a medida que ejecuto accept, voy sacando las conexiones pendientes del buffer y creando un nuevo sockets ACTIVO. Accept () devuelve el descriptor de archivo de este socket

**Connect():** el descriptor del socket, una dirección remota

Si puede conectarse, a partir de ahí, puedo hacer read y writes, en ese descriptor e intercambio datos.

Para los ejemplos el profe usa Pyton, porque le pinto, según el Taffe para que nos de curiosidad … pero en la práctica usan C++.

Pyton, s un lenguaje interpretado, ósea que no se compila. A medida que escribo líneas de código, las va ejecutando. (según el esta bueno para mostrar el tema de redes.) Este ejemplo es de un cliente.

Cuando se crea hay que asignar una familia, hay un montón posibles, la más conocidas son internet AF\_INET e INET6 (addrees family Intenet)… hay tantas porque sockets existe antes que existieran las redes, se usaba para IPC.

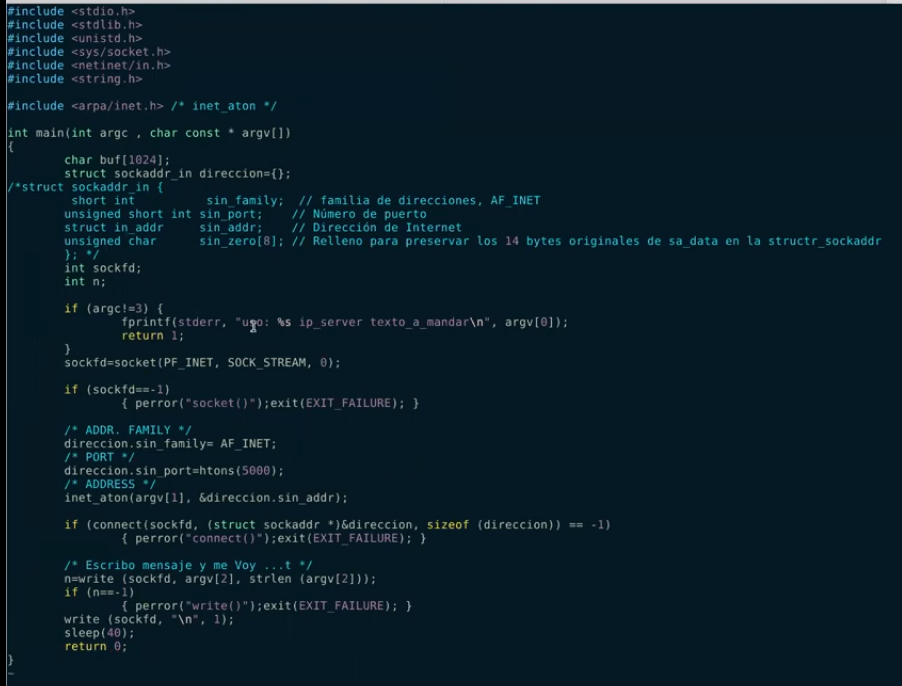
Cuando se empezaron a usar las redes, y tenían que crear una API para usarlas, en ves de crear una API nueva agregaron una familia a una ya existente.

Dentro la familia hay que decirle si es orientado a conexión o no, ósea Stream (Flujo) o Datagrama.

Ejemplo donde lee y escribe con recv y send (esto es pyton)

Entonces con estas 2 llamadas al sistema, con socket puedo crear un descriptor, y con connect (pasándole a donde me quiero conectar), podría crear una aplicación cliente.

Ahora el ejemplo de una ap. cliente, pero en “C++”



El 3er argumento de “socket()” cuando lo creo, es por si tengo más de un protocolo que haga sock Stream, siempre va en cero si usamos TCP/IP

En el connect el argumento de la dirección va como una estructura.

Y le paso el tamaño de la dirección.

Escribe lo que le paso por línea de comando

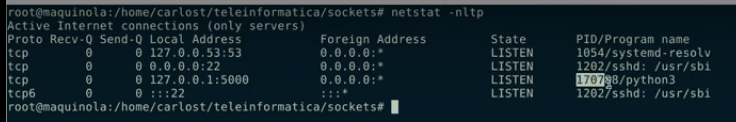
Ejemplo de socket en servidor (TCP):

Notar la diferencia entre pasivo y activo … a Listen le pasa la cantidad de elementos para el buffer

En pyton, accept devuelve el descriptor del socket activo, y la dirrecion IP y el puerto … en C, hay que pasarle un puntero a una estructura para que la llene con la dirección remota y el puerto.

Se queda esperando en el puerto 5000 que alguien se conecte, el proceso servidor está bloqueado por el SO esperando una entrada / salida.

El profe abre otra pestaña, instala la apt install net – tolos y ejecuta netstat que muestra estadísticas de los sockets de red

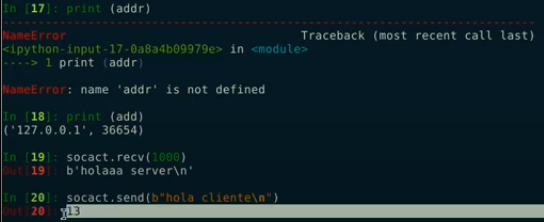




Creo el socket.

Lo primero que hago es asociarlo a una IP y aun puerto, cuando llegue algo ahí, el SO se lo pasa al proceso servidor. (Bind)

En realidad, lo deja en un buffer en el kernel, porque como esto es asincrónico, ósea no se cuando llegan los clientes, si mi proceso justo no está aceptando, se podrían perder las nuevas conexiones si no estuviera el buffer. (Listen crea el buffer)

Entonces la guarda en el buffer, cuando se ejecuta Accept(), lo que hace es buscar en el buffer, saca la mas antigua, y crea un Socket nuevo Activo, donde en el extremo remoto le pone la IP y el puerto del cliente que se conecta.

Obviamente una vez que se conecta, si quiero leer y escribir, con el pasivo no se puede, porque no conozco el otro extremo. Pero si en el activo.

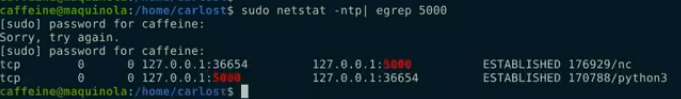
Y al final lo cierra.

Entonces, hasta ahora creo el socket pasivo, lo asocio con Bind a un puerto, y después creo el búfer con Listen.

¿Por qué crear un buffer en el kernel? Porque esto funciona así, una ves que un cliente se conecta, se crea un nuevo socket activo. Para aceptar una nueva conexión tengo que realizar la llamada al sistema ACCEPT(), que toma de todas las conexiones pendiente de aceptar la más antigua, ósea la que llego primero, que esta almacenada en el búfer.

Esto así porque mi proceso servidor es asincrónico, no siempre se esta ejecutando o esta bloqueado en espera de aceptar. Accept es bloqueante hasta que haya una nueva conexión.

Entonces mi proceso no siempre se ejecuta, y si vienen varias nuevas conexiones y si yo no ejecuto accept las pierdo, por eso necesito con Listen especificar un búfer en el Kernel para que guarde “n” posibles nuevas conexiones. Obviamente a medida que se ejecutan los accept las voy sacando del buffer y creando un socket nuevo activo, accept devuelve un nuevo descriptor de archivo que es un socket activo.

Abre otra pestaña

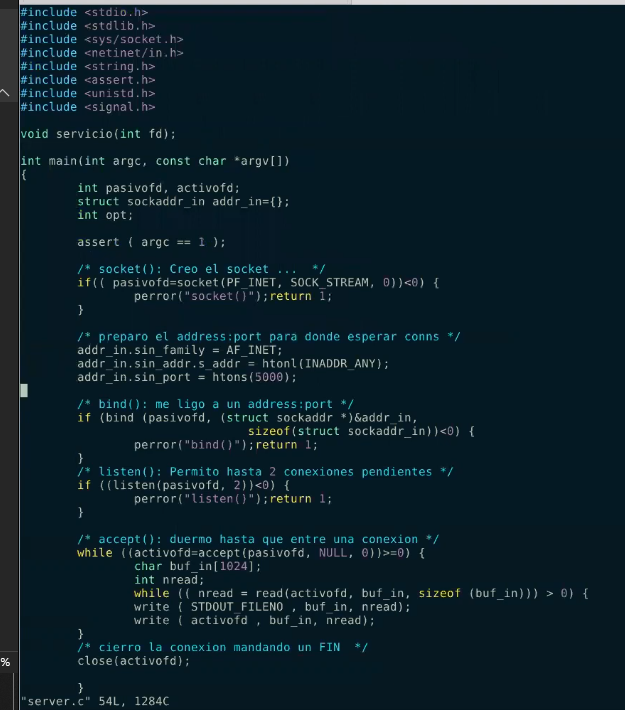
El profe evidencia un problema, si tengo 10 usuarios que se conectan a la vez, tengo que atender al 1ro, si este se toma todo el tiempo del mundo, hasta que no termine el proceso server no puede atender al 2do, y hasta que no termine el 2do usuario no puede atender al 3ro…. Imagínense el 10mo … se cansa …. Y en este caso, que pusimos un Listen = 10, si llegan más les dice conexión rechazada.

Cuando quieren hacer un saludo de 3 vías el SO le contesta con un Reset. Porque no tiene mas lugar en el búfer del Kernel para clientes nuevos.

Una mejora seria multiproceso, o multi Hilo…. Se podría quedar bloqueado en el Accept y cuando se desbloquea hago un Fork(), el proceso padre hace un Close() porque no le interesa tener el socket activo abierto y vuelve a un bucle de Accept(), entonces si hay 10 clientes, rápidamente crea 10 procesos, los acepta a los 10 y los procesa.

El buffer en el Kernel es chiquito, el profe cree que cada conexión ocupa 1KByte … 1000 conexiones 1MB

Ejemplo del proceso Servidor en C++



Esturctura, Tipo y nombre.

Creo el socket y valido si da error, menor que 1.

Arma la estructura interna que va a tener.

INADDR\_ANY: mi IP local

Asocio el socket con la dirección y el puerto, pasándole la estructura, y valida si hay error. Ej: que el puerto ya este ocupado por otro proceso.

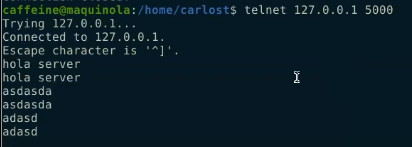
Genera un buffer con Listen() de hasta 2 conexiones pendientes de accept().

Entonces si quiero hacer un servidor que atienda todas las conexiones lo hayan de aca al futuro, lo voy a meter en un while para que este siempre aceptando conexiones.

Acepta y crea un socket activo. En el null normalmente se pone un puntero a una estructura donde nos devuelve la IP y el Puerto del cliente, en este ejemplo particular no lo usa. Solamente el accept devuelve un nuevo descriptor de un socket Activo que tiene los 2 extremos, el remoto y el local.

Al final lee lo que mandan el cliente, y el servidor hace una especie de echo, ósea lo muestra por pantalla y le responde al cliente con lo mismo.

Cliente:

Cuando cierro un socket, no lo corta de manera instantánea, sino luego de un pequeño periodo de tiempo, porque TCP tiene una maquina de estados, que por mas que corte lo deja un tiempo, por si hay algún paquete medio extraviado que demore en llegar.

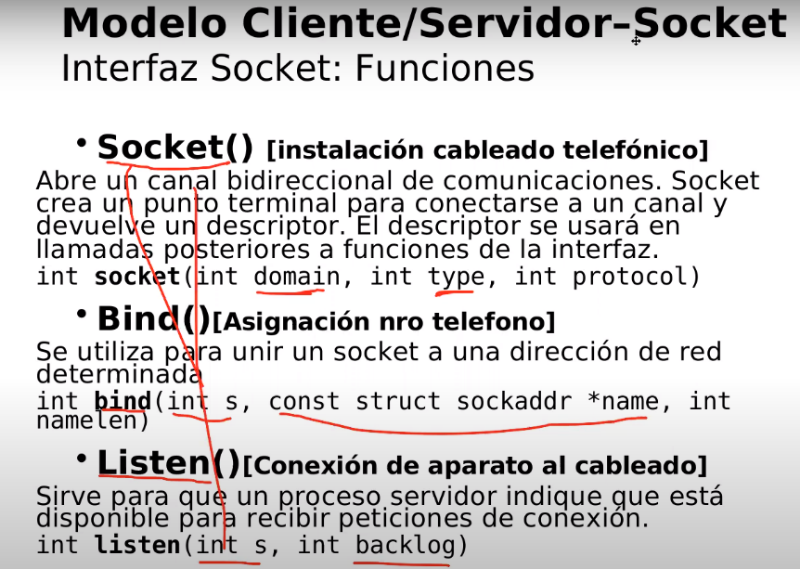
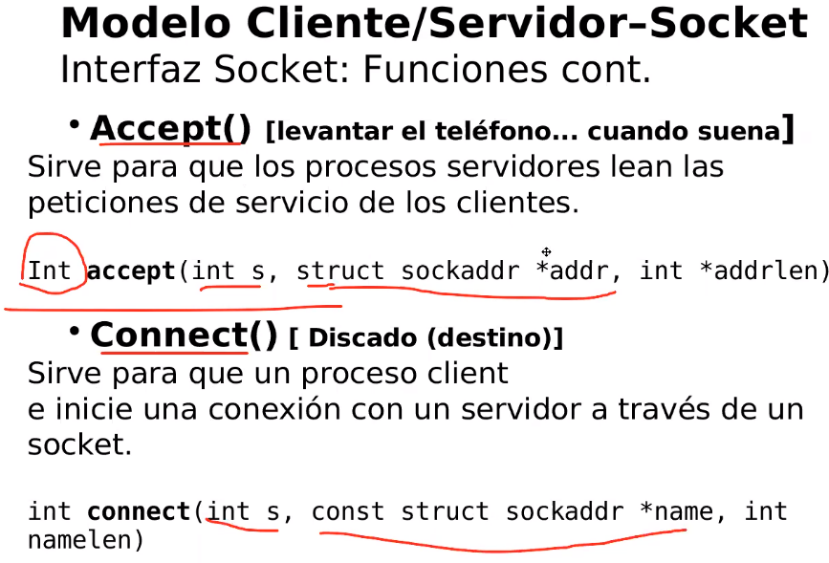
Socket: crea el descritor y hay que colocar el dominio y el tipo, (familia F INET y sockstream o datagram stream. (creo que se llaman distinto)

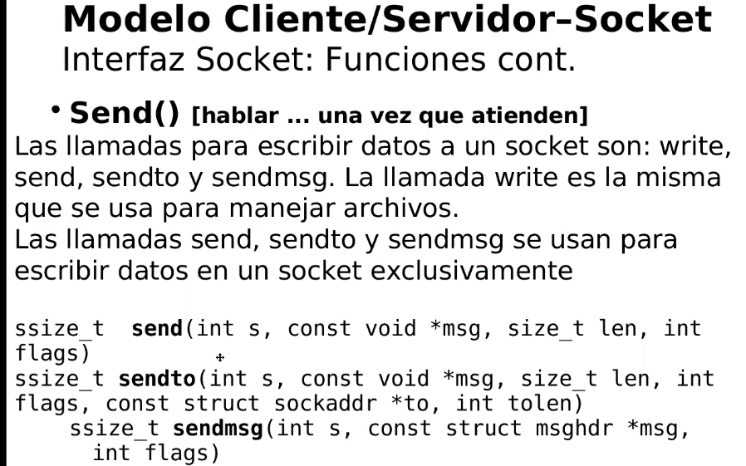
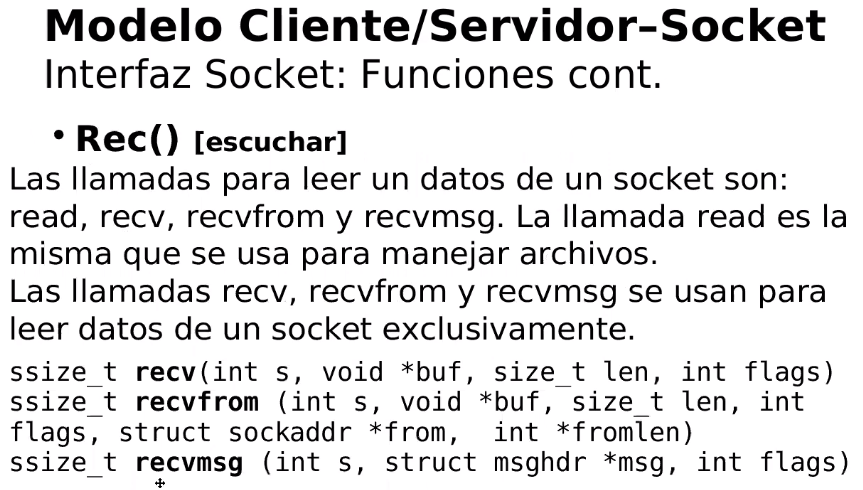
Bind(): es solo para servidores y le pasa el descriptor Pacivo del server, y la estructura donde tenia que tener la IP y Puerto Local “PROPIO” donde quiere esperar que alguien se conecte.

Listen(): es para crear un bufer que se llama backlog, le pasamos el descriptor del socket pasivo y la cantidad de conexiones.

Accept(): Saca del backlog un cliente o conexión pendiente, entonces devuelve un nuevo descriptor de socket Activo que está totalmente conectado. Le pasamos como argumento el socket pasivo, y si quisiera saber la IP y puerto del cliente se coloca un puntero y el SO llena la estructura con los datos del cliente. Todo esto del lado del servidor.

Connect(): Lo realizo del lado del cliente, le paso el descriptor propio y una estructura, donde tengo que poner la IP y el Puerto del proceso SERVIDOR al que me quiero conectar.

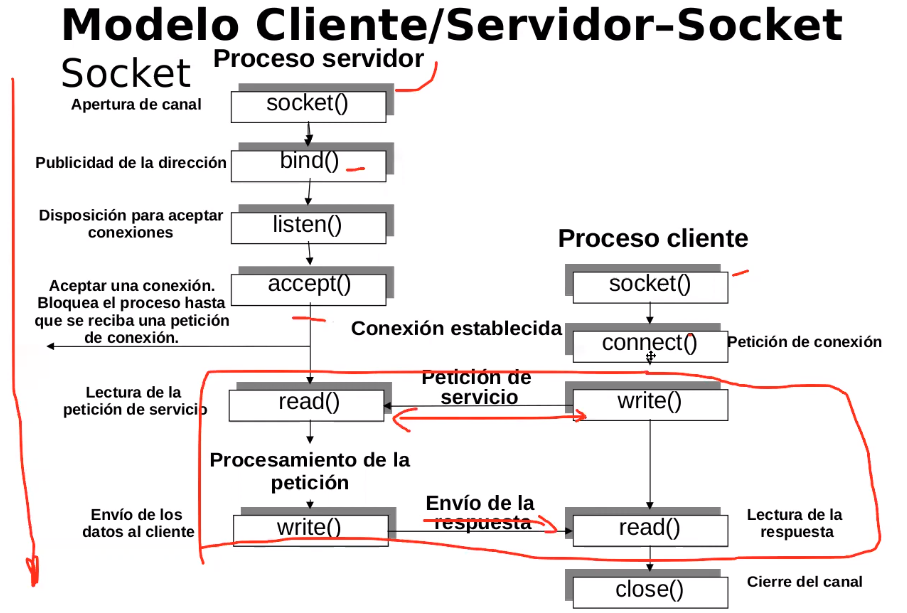




Una vez que estamos conectados podríamos coloca Send() o Write(), Le paso como argumento el descriptor del socket actico, conectado.

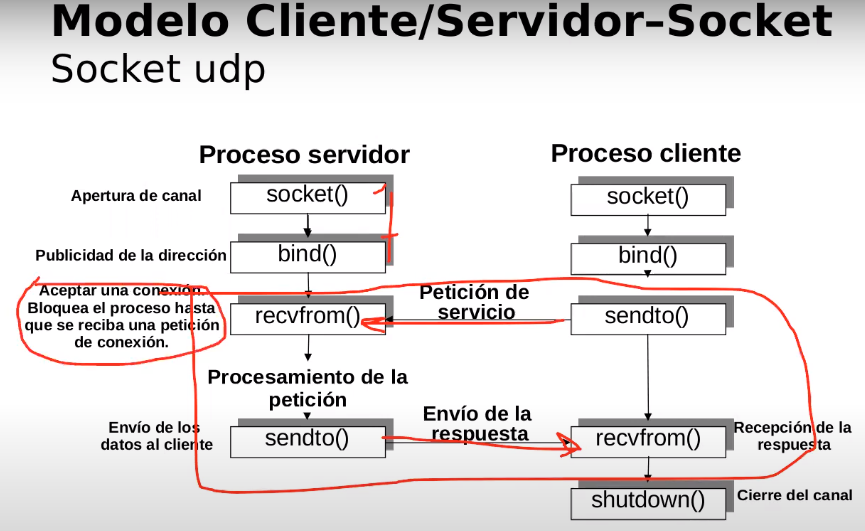
Y Recv() o Read().

Esto sería un Work Flow, flujo de como es la conexión.

Normalmente en el tiempo, del lado del servidor el proceso empieza antes, debería crear el socket, asignarlo, crear el buffer (escuchar) y se bloquea hasta nuevo aviso.

En algún momento un proceso cliente, crea un socket y lo conecta contra la IP y Puerto que especificaba el Bind del servidor, recién ahí se desbloquea el Accept() y el cliente se conecta. A partir de ahí están los 2 conectados.

De acá en mas depende del protocolo de capa de aplicación, quien escribe y quien lee. A partir de que están conectados, depende del protocolo de capa de aplicación que datos mandar y recibir, ósea el proceso cliente y servidor tienen que saber que datos mandar y recibir. Sino los 2 se quedan esperando y nadie hace nada.



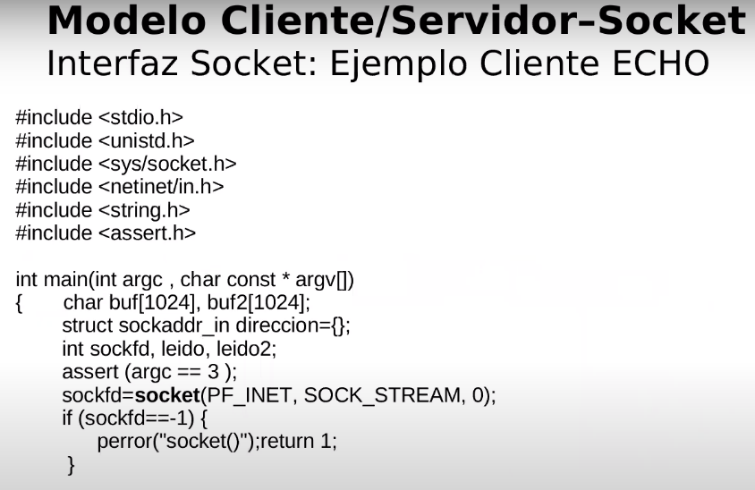
Con respecto de Socket hay una diferencia bastante importante.

1ro no tengo un Listen(): porque es para guardar todas las conexiones nuevas que todavía no atendí, como UDP es sin conexión no hay Listen() y tampoco hay Accept(). Y el cliente no tiene Connect(), no hay saludo de 3 vías, no se conecta.

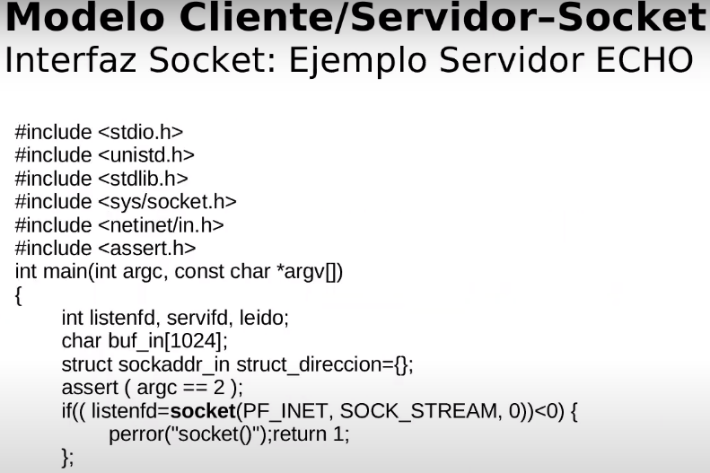
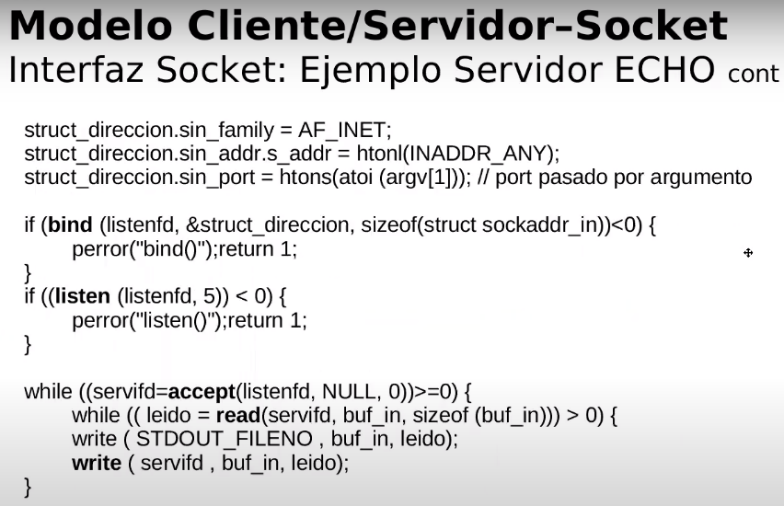
Tanto el servidor como el cliente crean el Socket, lo asocian al proceso, puerto e IP. Y después si alguien escribe y yo no estoy leyendo, se pierde el dato. Lo mismo si escribo y alguien no esta leyendo o escuchando.

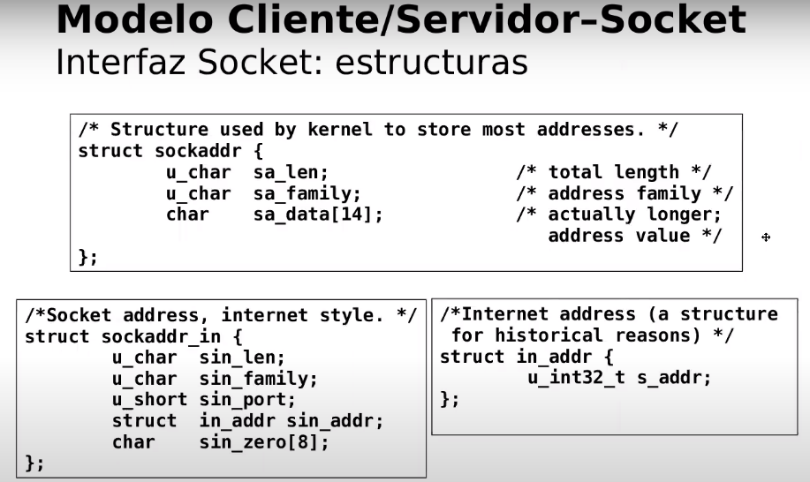
Acá no hay conexión. Lo que esta encerrado esta mal dijo el Taffe. Quedo de la filamina anterior.

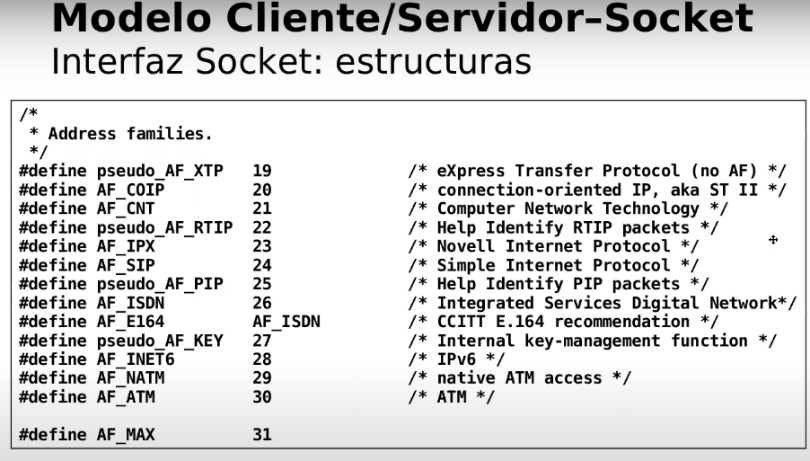
Ejemplos que el taffe pasa de largo.



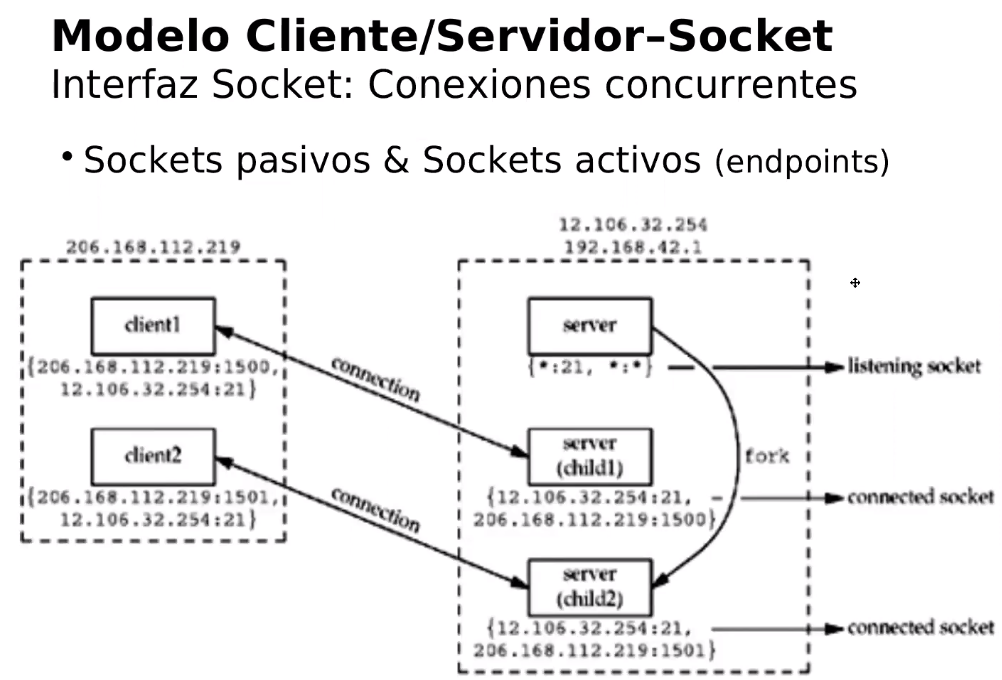








Conexiones concurrentes y tengo un servidor multiproceso, por ejemplo.

¿Cómo es el tema de los Puertos?

Porque se supone que no puedo tener mas de un proceso en el mismo puerto. ¿Entonces los hijos en que puerto están conectados? No es el mismo socket, cambia un dato.

En este ejemplo, el proceso servidor esta esperando en el puerto 21.

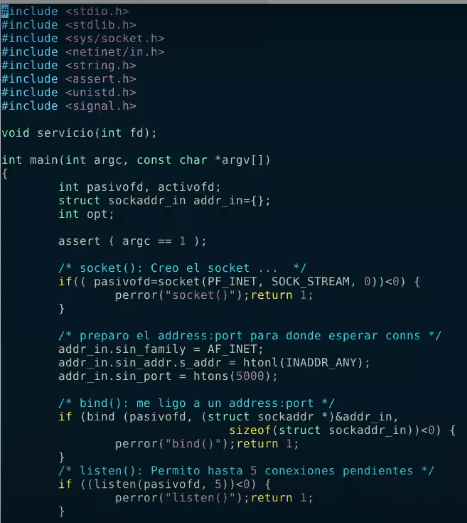
Endpoint local 21, remoto no tiene.

Se observa en la grafica que solo cambian los puertos locales de los clientes que están en la misma máquina. (misma IP)

Por lo que serían IPC distintos. Una para cada uno.

No hay problema que en el servidor haya 3 procesos usando el puerto 21, porque una esta en Listenen(), otra usa el puerto 21 pero lo que nos interesa son los endpoint. NO podría tener 2 procesos que usen los datos de los endpoint exactamente iguales. Mientras allá aunque sea uno de los 4 datos distintos no pasa nada.

Ejemplo de servidor concurrente en lenguaje C++

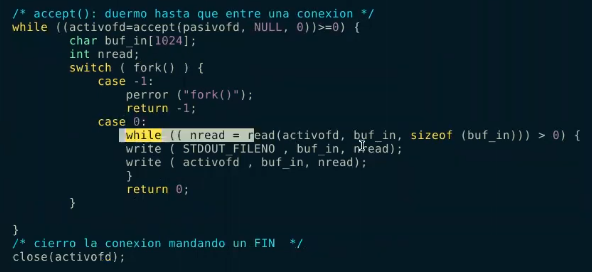


Creo un socket TCP

Armo la estructura

Lo asocio con Bind()

Le digo al SO que me deje un Bufer de 5 conexiones

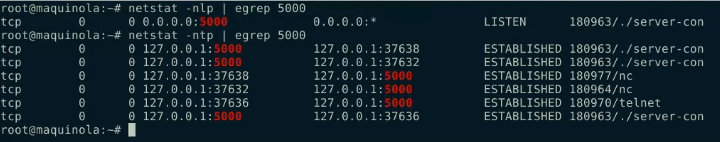
Después me quedo en un while del accept(), pero apenas un cliente se conecta hace un fork()

Evalúa el error

Y en el caso que sea el hijo, se queda en un while todo el tiempo que el cliente escriba el le va a responder, cuando termina el cliente termina el hijo.

El proceso padre continua en el bucle atendiendo mas clientes.

El close no debería estar dentro del bucle while en el padre? Observar que el padre sale del bucle en caso de error del fork, por eso cierra el descriptor que quedo abierto. Si el hijo se crea bien, la variable se sobre escribe en cada ciclo del while, con cada accept y al padre no le interesa.



Se observa que cada cliente tiene un puerto distinto, si no se mesclarían los datos entre unos y otros. Ósea el servidor sabe a quién contestarle en función de esos 4 valores de los endpoint.

Siempre se observa la misma IP porque en el ejemplo estamos en la misma máquina.