Sockets TCP/IP en C++

Di Paola Martín
martinp.dipaola <at> gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

De qué va esto?

Stack Web

Redes TCP/IP (simplificado)

Resolución de nombres

Canal de comunicación TCP

- Establecimiento de un canal
- Envio y recepción de datos
- Finalización de un canal

Stack Web

Stack Web (simplificado)

	...
`firefox`	HTML / CSS / JS
`httpie`	HTTP
	TLS
`netcat/OS`	TCP
`OS`	IP
	...
`HW`	Maxwell's eq

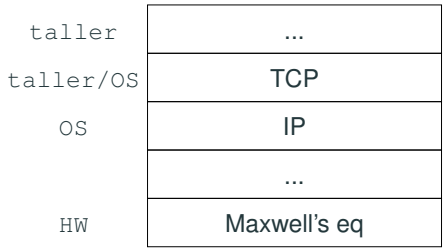
- Nuestras comunicaciones se basana en las leyes de la físicas descriptas por las ecuaciones de Maxwell.
- Trabajar con campos electromagneticos es trabajo del hardware. Sobre él el sistema operativo resuelve el ruteo por la red IP y el transporte de los datos via TCP.
- TCP esta en el límite entre las aplicaciones de user (**`netcat`** y otros) y el OS: mientras que el OS implementa el protocolo TCP, la aplicación de user la usa.
- TLS se usa para auténticas y encriptar las comunicaciones (SSL fue su antecesor). Aunque es opcional, muchas aplicaciones hoy usan TLS y así debería ser.
- **`httpie`** (**`curl`**, **`wget`**, **`aria2`**) son capaces de comunicarse con servidores y hablar HTTP aunque la interpretación del contenido (HTML, CSS) es limitada o nula.
- No tomes este diagrama literal: es una versión simplificada.

Resolucion de nombres (simplificado)

`nslookup`	DNS
`nslookup/OS`	UDP
`OS`	IP
	...
`HW`	Maxwell's eq

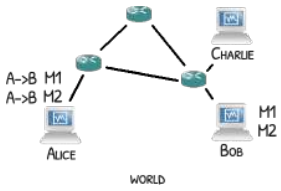
- Resolver un service name a un puerto es fácil. Los servicios estándar son pocos y no cambian, un archvio con el mapping alcanza (**`/etc/services`**)
- Resolver un hostname a una IP es mucho más complicado, un archivo (**`/etc/hosts`**) no es lo suficientemente dinámico.
- El protocolo DNS se encarga de resolver un hostname a una direccion IPv4 o IPv6.
- Al igual que con TCP, UDP está en el límite entre aplicaciones de user (**`nslookup`**) y el OS.

TP de Taller (simplificado)



- A grandes rasgos es sobre TCP/UDP donde nos paramos y desarrollamos el TP en Taller.

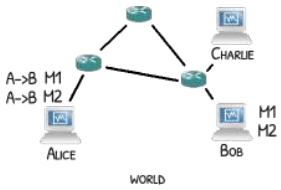
Internet - Protocolo IP (simplificado)



- Los mensajes son *ruteados* a sus destinos (*hosts*)
- Dos esquemas de direcciones: IPv4 (4 bytes) e IPv6 (16 bytes).
- Son redes *best effort*
  - Los paquetes se pueden perder.
  - Los paquetes puede llegar en desorden.
  - Los paquetes puede llegar duplicados.

- Ahora la red esta segmentada: los mensajes son enviados de un segmento a otro a traves de los routers.
- Los routers usan las direcciones IP de destino para saber a donde enviar los mensajes.
- La red esta governada por el protocolo IP. Existen actualmente 2 versiones IPv4 e IPv6.
- El primero usa direcciones de máquina (host) de 4 bytes y el segundo de 16.
- IP no garantiza que lleguen todos los paquetes, ni el orden ni que no haya duplicados.
- Es un protocolo pensado para simplificar el hardware de la red, no para hacerle más fácil la vida a los desarrolladores.

Internet - Protocolo TCP (simplificado)

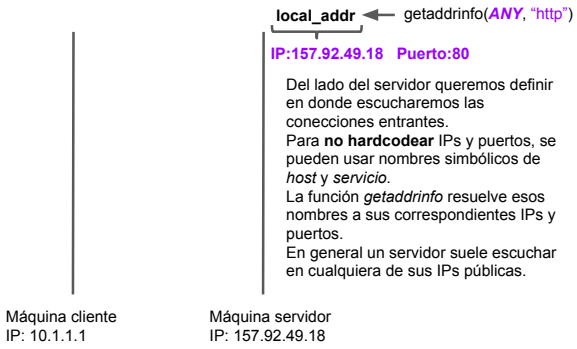


- Corre sobre IP, permite el direccionamiento a nivel de servicio (*port*)
- Orientado a bytes, no a mensajes (*stream*): los bytes no se pierden, desordenan ni duplican pero no garantiza *boundaries*
- Con conexión y full-duplex. Análogo a un archivo binario secuencial.

- IP solo nos habla de los hosts, no de los programas que corren en ellos.
- TCP permite direccionar a cada programa o servicio a traves de un número, el puerto.
- TCP es orientado a la conexión: hay un participante pasivo que espera una comunicación y hay otro que la inicia de forma activa.
- Típicamente el participante pasivo es el servidor y el activo el cliente.
- Una vez establecida la conexión los bytes enviados (full duplex) no se pierden, desordenan ni duplican.
- TCP no garantiza nada sobre los mensajes, solo sabe de bytes, por lo que un mensaje puede llegar incompleto.

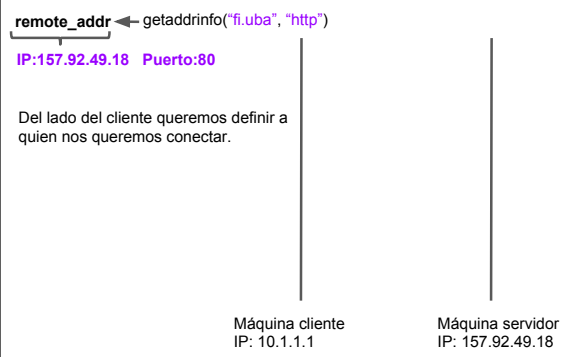
Resolución de nombres

Resolución de nombres: desde donde quiero escuchar



- El servidor tiene que definir desde donde quiere recibir las conexiones.
- Hay más esquemas posibles pero solo nos interesa definir la IP y el puerto del servidor.
- Sin embargo, hardcodear la IP y/o el puerto es una mala práctica. Mejor es usar nombres simbólicos: host name y service name.
- La función `getaddrinfo` se encargara de resolver esos nombres y llevarlos a IPs y puertos.

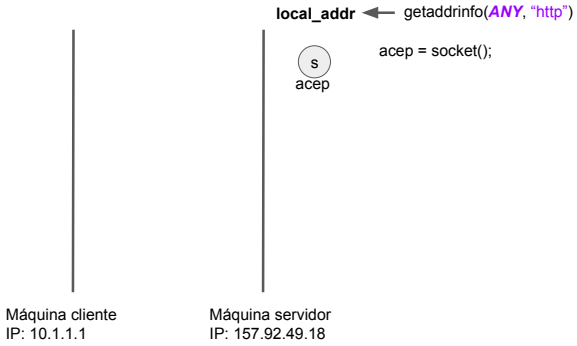
Resolución de nombres: a quien me quiero conectar



Canal de comunicación TCP

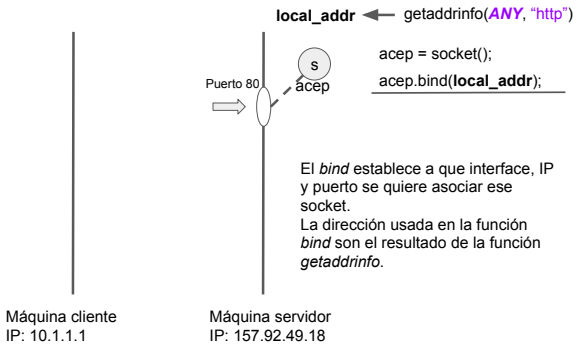
Establecimiento de un canal

Creación de un socket



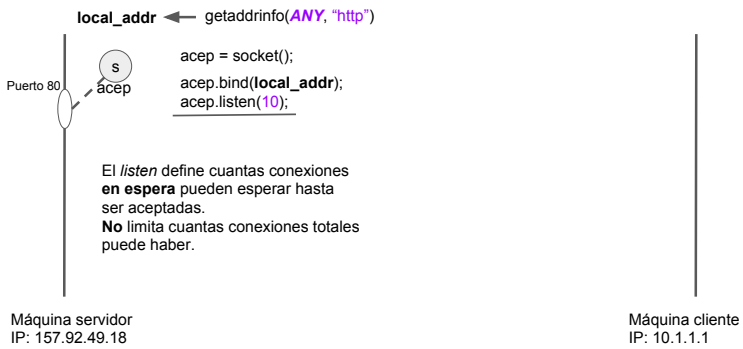
- Crear un socket no es nada mas que crear un file descriptor al igual que cuando abrimos un archivo.

Enlazado de un socket a una dirección



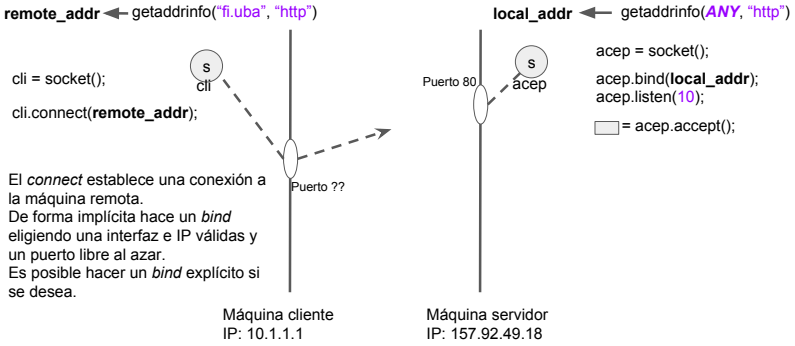
- A los sockets se los puede enlazar o atar a una dirección IP y puerto local para que el sistema operativo sepa desde donde puede enviar y recibir conexiones y mensajes.
- El uso mas típico de **`bind`** se da del lado del servidor cuando este dice "quiero escuchar conexiones desde mi IP pública y en este puerto".
- Sin embargo el cliente también puede hacer **`bind`** por razones un poco mas esotéricas.

Socket aceptador o pasivo



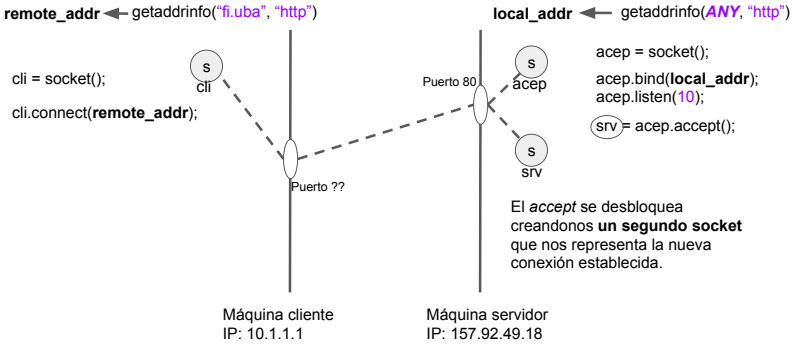
- Una vez enlazado le decimos al sistema operativo que queremos escuchar conexiones en esa IP/puerto.
- La función **`listen`** define hasta cuantas conexiones en "espera de ser aceptadas" el sistema operativo puede guardar.
- La función **`listen`** NO define un límite de las conexiones totales (en espera + las que estan ya aceptadas). No confundir!
- Ahora el servidor puede esperar a que alguien quiera conectarse y aceptar la conexión con la función **`accept`**.
- La función **`accept`** es bloqueante.

Conexión con el servidor: estableciendo conexión



- El cliente usa su socket para conectarse al servidor. La operación `connect` es bloqueante.

Conexión con el servidor: aceptando la conexión

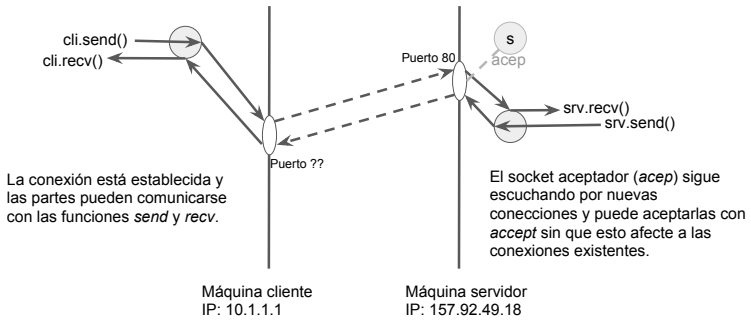


- La conexión es aceptada por el servidor: la función `accept` se desbloquea y retorna un nuevo socket que representa a la nueva conexión.

Canal de comunicación TCP

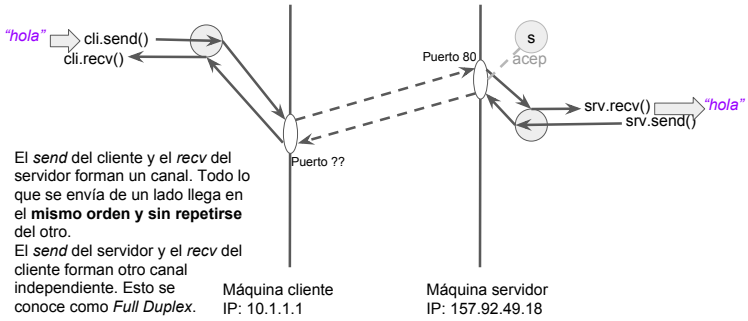
Envio y recepción de datos

Conexión establecida



- El socket **`acep`** sigue estando disponible para que el servidor acepte a otras conexiones en paralelo mientras antiende a sus clientes (es independiente del socket **`srv`**)
- Al mismo tiempo, el socket **`srv`** quedo asociado a esa conexión en particular y le permitirá al servidor enviar y recibir mensajes de su cliente.
- Tanto el cliente como el servidor se pueden enviar y recibir mensajes (**`send`**/**`recv`**) entre ellos.
- Los mensajes/bytes enviados con **`cli.send`** son recibidos por el servidor con **`srv.recv`**.
- De igual modo el cliente recibe con **`cli.recv`** los bytes enviados por el servidor con **`srv.send`**.

Envio y recepción de datos



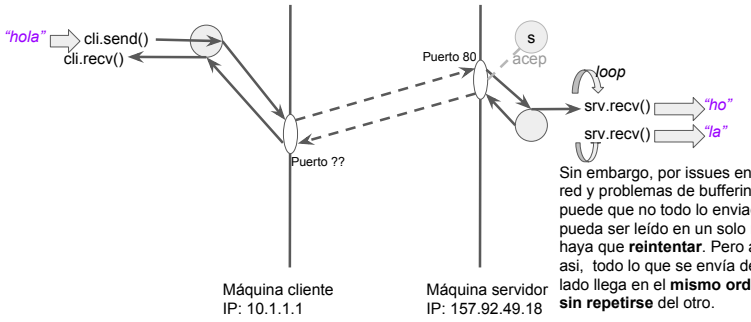
El *send* del cliente y el *recv* del servidor forman un canal. Todo lo que se envía de un lado llega en el **mismo orden y sin repetirse** del otro.
El *send* del servidor y el *recv* del cliente forman otro canal independiente. Esto se conoce como *Full Duplex*.

Máquina cliente
IP: 10.1.1.1

Máquina servidor
IP: 157.92.49.18

- El par **`cli.send`**–**`srv.recv`** forma un canal en una dirección mientras que el par **`srv.send`**–**`cli-recv`** forma otro canal en el sentido opuesto.
- Ambos canales son independientes. Esto se lo conoce como comunicación Full Duplex
- TCP garantiza que los bytes enviados llegaran en el mismo orden, sin repeticiones y sin pérdidas del otro lado.
- Otro protocolos como UDP no son tan robustos...

Envio y recepción de datos en la realidad

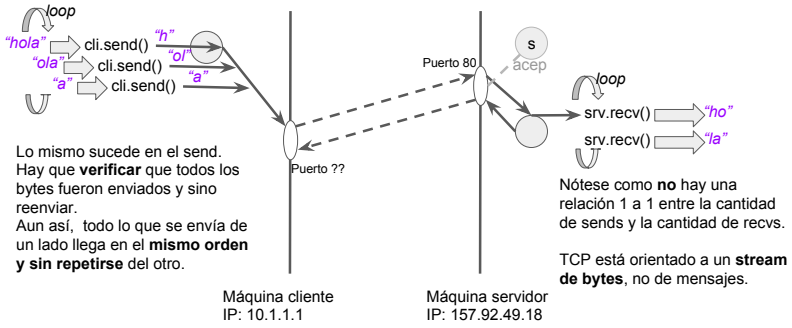


Sin embargo, por issues en la red y problemas de bufferings puede que no todo lo enviado pueda ser leído en un solo recv y haya que **reintentar**. Pero aun asi, todo lo que se envía de un lado llega en el **mismo orden y sin repetirse** del otro.

Máquina cliente
IP: 10.1.1.1

Máquina servidor
IP: 157.92.49.18

Envio y recepción de datos en la realidad



Lo mismo sucede en el send. Hay que **verificar** que todos los bytes fueron enviados y sino reenviar.
Aun así, todo lo que se envía de un lado llega en el **mismo orden y sin repetirse** del otro.

Nótese como **no** hay una relación 1 a 1 entre la cantidad de sends y la cantidad de recvs.

TCP está orientado a un **stream de bytes**, no de mensajes.

Máquina cliente
IP: 10.1.1.1

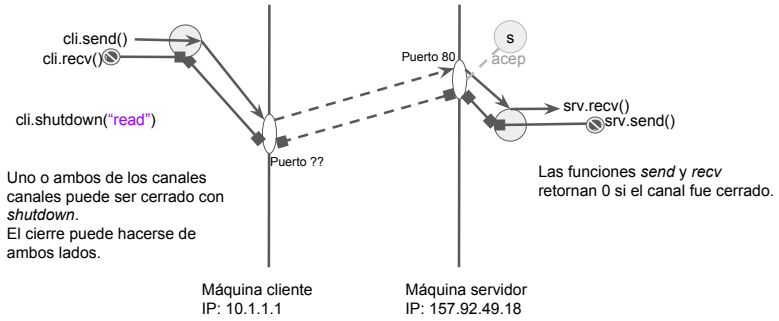
Máquina servidor
IP: 157.92.49.18

- Sin embargo TCP NO garantiza que todos los bytes pasados a **`send`** se puedan enviar en un solo intento: el programador debera hacer múltiples llamadas a **`send`**.
- De igual modo, no todo lo enviado sera recibido en una única llamada a **`recv`**: el programador debera hacer múltiples llamadas a **`recv`**.

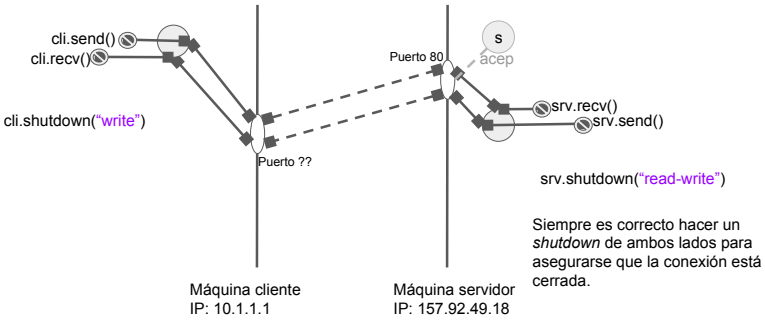
Canal de comunicación TCP

Finalización de un canal

Cierre de conexión parcial

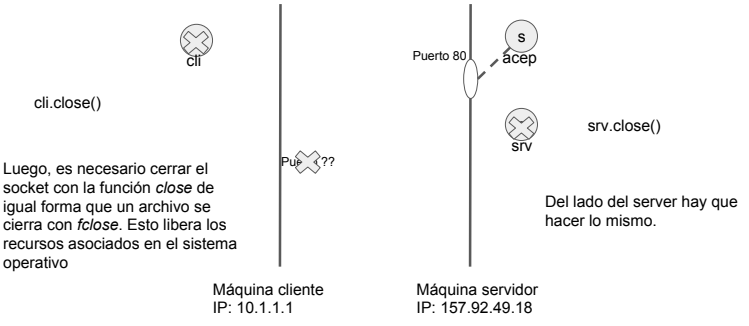


Cierre de conexión total

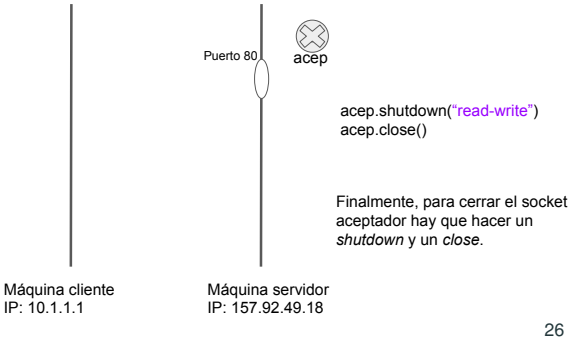


- Parcial en un sentido (envio) **`SHUT_WR`**
- Parcial en el otro sentido (recepción) **`SHUT_RD`**
- Total en ambos sentidos **`SHUT_RDWR`**

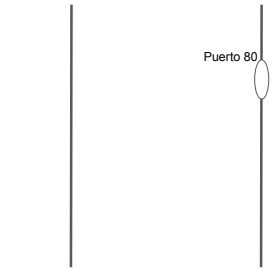
Liberación de los recursos con close



Cierre y liberación del socket aceptador



TIME WAIT



A pesar de haber hecho un *shutdown* y un *close*, el puerto queda en un estado especial llamado TIME_WAIT (incluso si el proceso terminó)

Esto es forzado por el sistema operativo para evitar “ciertos problemas”

Después de unos segundos el puerto queda libre para volver a ser usado.

Máquina cliente
IP: 10.1.1.1

Máquina servidor
IP: 157.92.49.18

Appendix

Referencias

Referencias i

- man getaddrinfo
- man netcat
- man netstat
- RFCs 971, 2460, ...
- RFCs 793, ...
- TCP/IP Illustrated, Richard Stevens
- Data and Computer Comunications, Ed Stallings