Sockets TCP/IP en C

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

Facultad de Ingeniería
Universidad de Buenos Aires

De qué va esto?

Redes TCP/IP (simplificado)

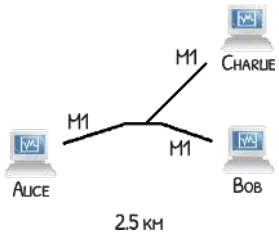
Resolución de nombres

Canal de comunicación TCP

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

Protocolos y formatos

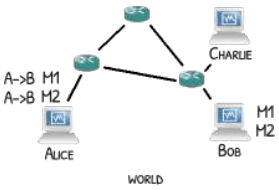
Medios compartidos (simplificado)



- Los mensajes son recibidos por todos (*shared*).
- No se requiere hardware adicional en la red.
- Solo un participante puede hablar a la vez:
  La performance se degrada a mayor cantidad de participantes.

- Dado que el medio es compartido, cuando Alice envia un mensaje este se propaga por toda la red.
- El mensaje es recibido entonces por todos los participantes.
- Y a la vez evita que otros puedan comunicarse por que el medio esta en uso. Esto se conoce como *half duplex*.
- Este tipo de redes requieren un hardware adicional mínimo o nulo lo que las hace particularmente baratas y fáciles de mantener.
- Son para redes locales, como una red Wifi.

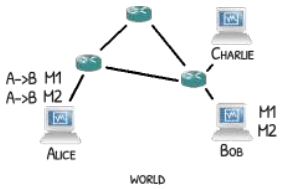
Internet - Protocolo IP (simplificado)



- 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.

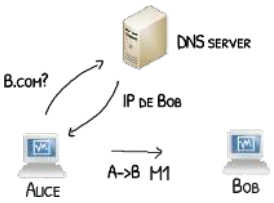
Internet - Protocolo TCP (simplificado)



- 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.

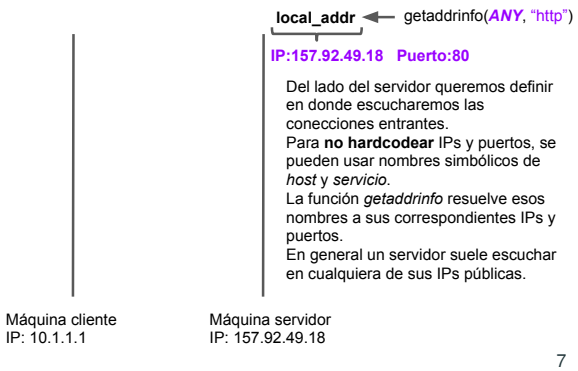
Internet - Protocolo DNS (simplificado)



- No es necesario recordar la dirección IP del destino (4 o 16 bytes) sino su *nombre de dominio*.
- La registración de dominios se hace a nivel gubernamental. Para el caso `.com.ar` lo hace NIC, Cancillería Argentina.
- La resolución de un dominio a una o varias direcciones IP las hace el servidor de DNS.
- Un dominio puede tener múltiples IPs: por redundancia o por ser IPv4 e IPv6.

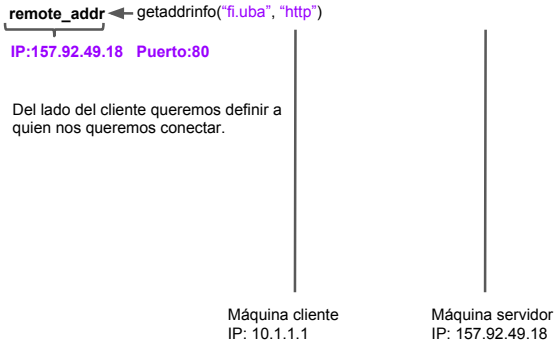
Resolución de nombres

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



Resolución de nombres

Cliente

```
1 memset(&hints, 0, sizeof(struct addrinfo));
2 hints.ai_family   = AF_INET;      /* IPv4  */
3 hints.ai_socktype = SOCK_STREAM;  /* TCP   */
4 hints.ai_flags    = 0;
5
6 status = getaddrinfo("fi.uba.ar", "http", &hints, &results);
```

Servidor

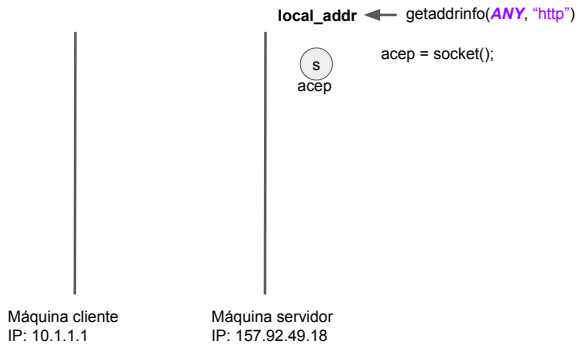
```
1 memset(&hints, 0, sizeof(struct addrinfo));
2 hints.ai_family   = AF_INET;      /* IPv4  */
3 hints.ai_socktype = SOCK_STREAM;  /* TCP   */
4 hints.ai_flags    = AI_PASSIVE;
5
6 status = getaddrinfo(0 /* ANY */, "http", &hints, &results);
```

```
1 freeaddrinfo(results);  // Cliente y Servidor
```


Canal de comunicación TCP

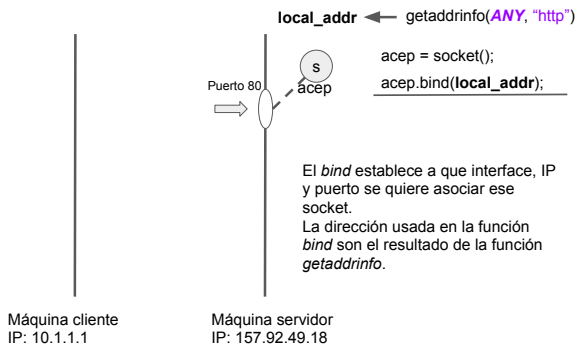
Establecimiento de un canal

Creación de un socket



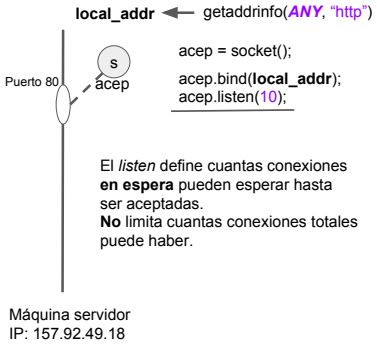
- Crear un socket no es nada mas que crear un file descriptor al igual que cuando abrimos un archivo.

Enlazado de un socket a una dirección



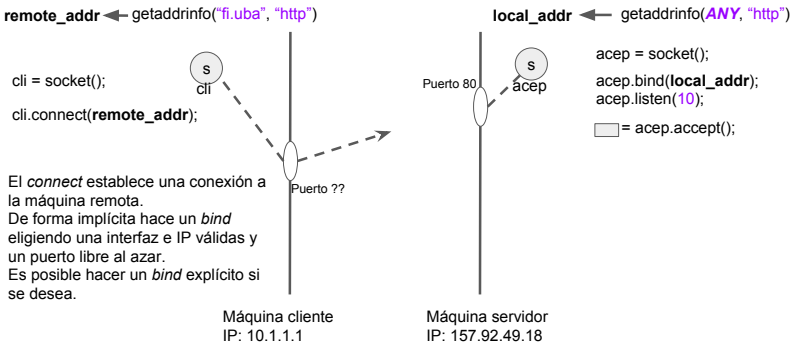
- A los sockets se los puede enlazar o atar a una dirección IP y puerto local para que el sistema operativo sepa desde donde puede enviar y recibir conexiones y mensajes.
- El uso mas típico de **`bind`** se da del lado del servidor cuando este dice "quiero escuchar conexiones desde mi IP pública y en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

Socket aceptador o pasivo



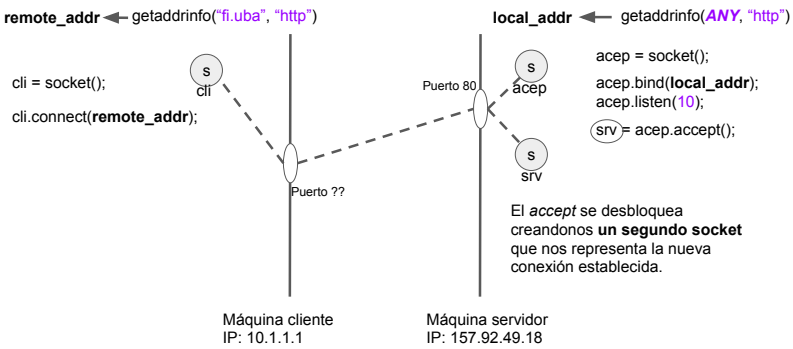
- Una vez enlazado le decimos al sistema operativo que queremos escuchar conexiones en esa IP/puerto.
- La función **`listen`** define hasta cuantas conexiones en "espera de ser aceptadas" el sistema operativo puede guardar.
- La función **`listen`** NO define un límite de las conexiones totales (en 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

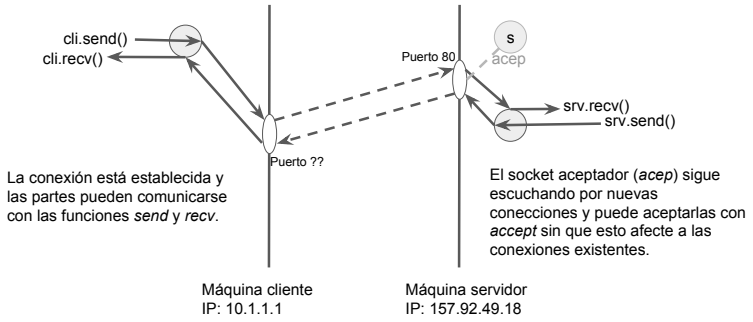


- La conexión es aceptada por el servidor: la función **`accept`** se desbloquea y retorna un nuevo socket que representa a la nueva conexión.

Canal de comunicación TCP

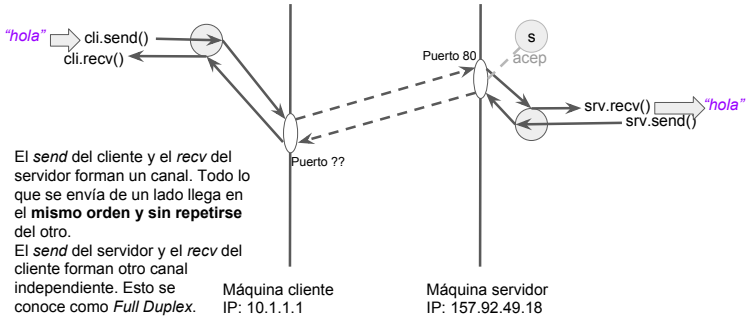
Envio y recepción de datos

Conexión establecida



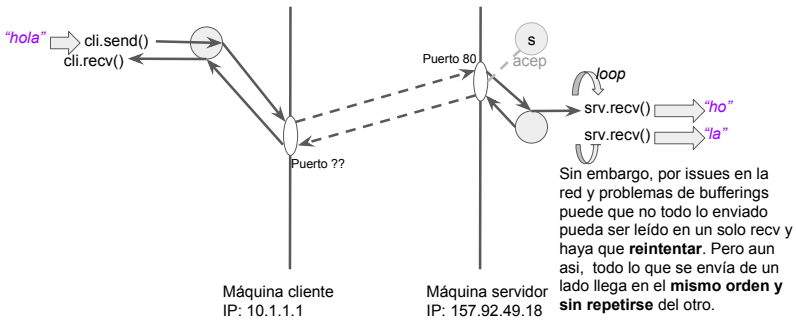
- El socket **`acep`** sigue estando disponible para que el servidor acepte a otras conexiones en paralelo mientras antiende a sus clientes (es independiente del socket **`srv`**)
- Al mismo tiempo, el socket **`srv`** quedo asociado a esa conexión en particular y le permitirá al servidor enviar y recibir 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



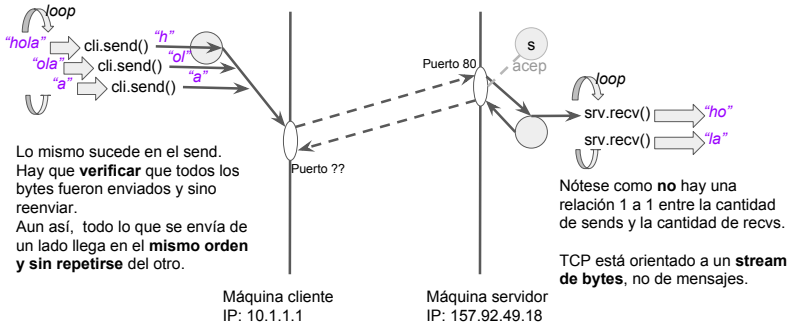
- El par **`cli.send`**–**`srv.recv`** forma un canal en una dirección mientras que el par **`srv.send`**–**`cli-recv`** forma otro canal en el sentido opuesto.
- Ambos canales son independientes. Esto se lo conoce como comunicación Full Duplex
- TCP garantiza que los bytes enviados llegaran en el mismo 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.

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.

- 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`**.

Envio y recepción de datos

```
1  int s = send(skt,
2               buf,
3               bytes_to_sent,
4               flags           // MSG_NOSIGNAL
5             );
6
7  int s = recv(skt,
8               buf,
9               bytes_to_recv,
10              flags           // 0
11            );
12
13   (s == -1)  // Error inesperado, ver errno
14    (s == 0)  // El socket fue cerrado
15     (s > 0)  // Ok: s bytes fueron enviados/recibidos
```


recvall: recepción de N bytes exactos

```
1  char buf[MSG_LEN];  // buffer donde guardar los datos
2  int bytes_recv = 0;
3
4  while (MSG_LEN > bytes_recv && skt_still_open) {
5    s = recv(skt, &buf[bytes_recv], MSG_LEN - bytes_recv - 1,
6                                                        0);
7    if (s == -1) { // Error inesperado, ver errno
8       /* ... */
9    }
10   else if (s == 0) { // Nos cerraron el socket
11      /* ... */
12   }
13   else {
14     bytes_recv += s;
15   }
16 }
```

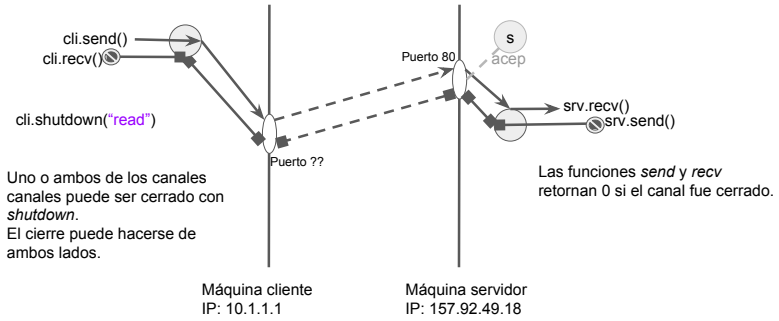

sendall: envío de N bytes exactos

```
1  char buf[MSG_LEN];    // buffer con los datos a enviar
2  int bytes_sent = 0;
3
4  while (MSG_LEN > bytes_sent && skt_still_open) {
5    s = send(skt, &buf[bytes_sent], MSG_LEN - bytes_sent,
6                                                MSG_NOSIGNAL);
7    if (s == -1) { // Error inesperado, ver errno
8       /* ... */
9    }
10   else if (s == 0) { // Nos cerraron el socket
11      /* ... */
12   }
13   else {
14     bytes_sent += s;
15   }
16 }
```

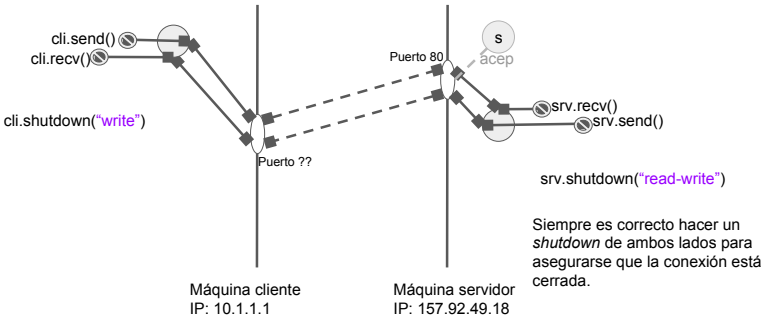
Canal de comunicación TCP

Finalización de un canal

Cierre de conexión parcial

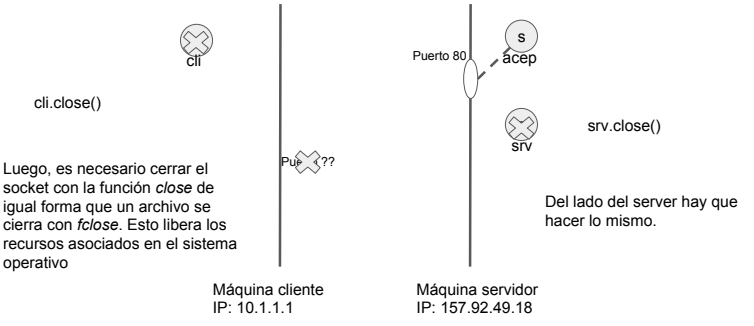


Cierre de conexión total

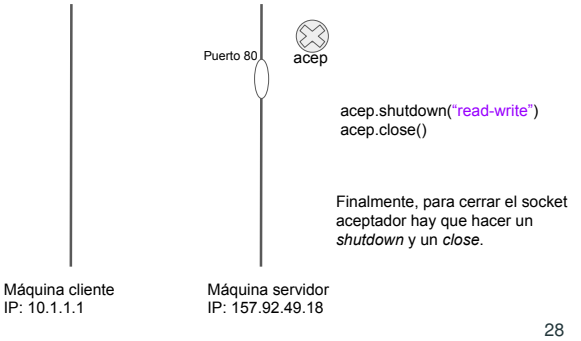


- Parcial en un sentido (envio) **`SHUT_WR`**
- Parcial en el otro sentido (recepción) **`SHUT_RD`**
- Total en ambos sentidos **`SHUT_RDWR`**

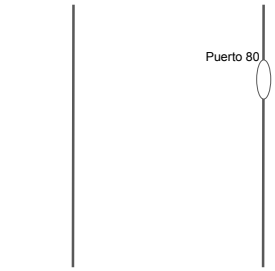
Liberación de los recursos con close



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

TIME WAIT -> Reuse Address

Si el puerto 80 esta en el estado TIME WAIT, esto termina en error (Address Already in Use):

```
1 int acep   = socket(...);
2 int status = bind(acep, ...); //bind al puerto 80
```

La solución es configurar al socket aceptador para que pueda reusar la dirección:

```
1 int acep   = socket(...);
2
3 int val = 1;
4 setsockopt(acep, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
5
6 int status = bind(acep, ...); //bind al puerto 80
```


Protocolos y formatos

Binario o Texto

- Protocolos en Texto: son la contracara de los protocolos binarios, son lentos, ineficientes y más difíciles de parsear pero más fáciles de debuggear. Son independientes del endianess, padding y otros pero dependen del encoding del texto y que caracteres se usan como delimitadores.
- Protocolos en Binario: son simples y eficientes en terminos de memoria y velocidad de procesamiento. Son más difíciles de debuggear. Es necesario tomar en consideración el endianess, el padding, los tamaños y los signos.

HTTP

```
1 GET /index.html HTTP/1.1\r\n
2 Host: www.fi.uba.ar\r\n
3 \r\n
```

- En HTTP el fin del mensaje esta dado por una línea vacia; cada línea esta delimitada por un `\r\n`
- Cuantos bytes reservarían para contener dicho mensaje o alguna línea?
- Que pasa si el delimitador `\r\n` aparece en el medio de una línea, como lo diferenciarían?

- Habitualmente en protocolos en texto se usa uno o una secuencia de caracteres como delimitadores.
- En la cabezera de HTTP se usa `\r\n`
- En C/C++, los fin de strings son marcados con `\0`
- Aunque simple, no es trivial saber cuantos bytes hay hasta el delimitador.
- Tampoco es trivial el caso de que el texto contenga al delimitador meramente por que es parte de su contenido.
- Hay dos opciones, o se opta por otro protocolo o se usa una secuencia de escape para que el delimitador sea considerado un literal y no un delimitador.
- Y si la secuencia de escape es parte del contenido? Hay que escapear la secuencia de escape con otra secuencia de escape.
- Por ejemplo, el compilador de C/C++ ve `"\1"` como un string con el byte 1 (a pesar de haber 2 caracteres). Si se quisiera literalmente poner una barra y un 1 hay que escapear la barra: `"\\1"`

TLV

```
1  struct Msj {
2      unsigned short type;
3      unsigned short length;
4      char* value;
5  };
6
7  read(fd, &msj.type, sizeof(unsigned short) * 2);
8  msj.value = (char*) malloc(msj.length);
9  read(fd, msj.value, msj.length);
```

- Los primeros 4 bytes indican la longitud y tipo del valor; el resto de los bytes son el valor en sí.
- Por qué es importante usar `unsigned short` y no solamente `short`? Qué pasa si `sizeof(unsigned short)` no es 2?
- Que pasa si el endianess no coincide? y si hay padding entre los dos primeros campos?

- Prefijar la longitud del mensaje soluciona varios problemas pero trae otros.
- Si `sizeof(unsigned short)` vale 4 estaríamos enviando 8 bytes con las longitud y tipo pero la máquina que recibe el mensaje puede esperar 4.
- Hay que definir y forzar un endianess y reglas de padding.

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