

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

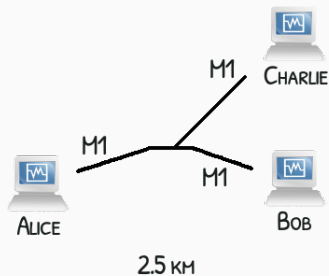
- Establecimiento de un canal

- Envío y recepción de datos

- Finalización de un canal

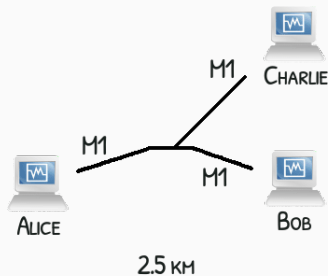
Protocolos y formatos

Medios compartidos (simplificado)



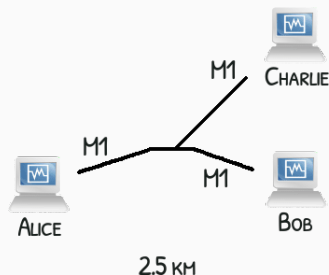
- Los mensajes son recibidos por todos (*shared*).

Medios compartidos (simplificado)



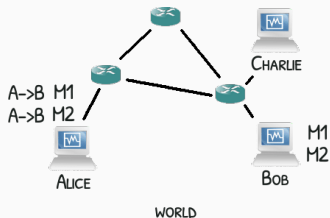
- Los mensajes son recibidos por todos (*shared*).
- No se requiere hardware adicional en la red.

Medios compartidos (simplificado)



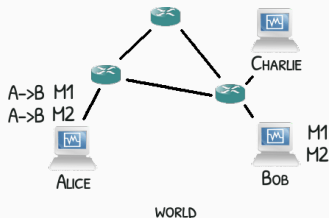
- Los mensajes son recibidos por todos (*shared*).
- No se requiere hardware adicional en la red.
- Solo un participante puede hablar a la vez:
La performance se degrada a mayor cantidad de participantes.

Internet - Protocolo IP (simplificado)



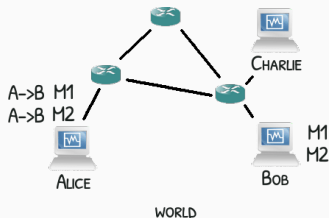
- Los mensajes son *ruteados* a sus destinos (*hosts*)

Internet - Protocolo IP (simplificado)



- Los mensajes son *ruteados* a sus destinos (*hosts*)
- Dos esquemas de direcciones: IPv4 (4 bytes) e IPv6 (16 bytes).

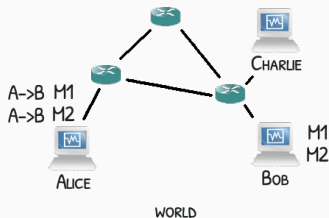
Internet - Protocolo IP (simplificado)



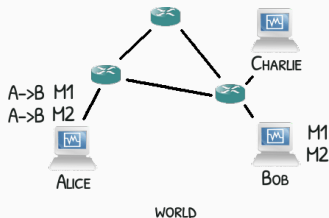
- Los mensajes son *ruteados* a sus destinos (*hosts*)
- Dos esquemas de direcciones: IPv4 (4 bytes) e IPv6 (16 bytes).
- Son redes *best effort*
 - Los paquetes se pueden perder.
 - Los paquetes puede llegar en desorden.
 - Los paquetes puede llegar duplicados.

Internet - Protocolo TCP (simplificado)

- Corre sobre IP, permite el direccionamiento a nivel de servicio (*port*)

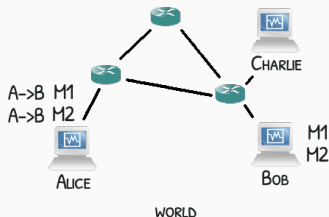


Internet - Protocolo TCP (simplificado)



- 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***

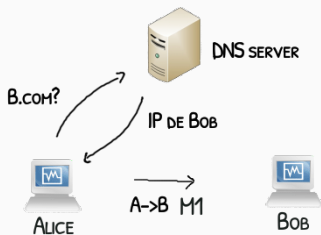
Internet - Protocolo TCP (simplificado)



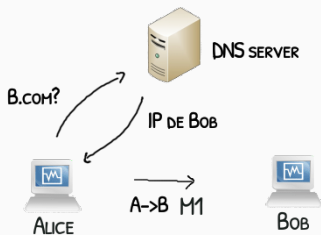
- Corre sobre IP, permite el direccionamiento a nivel de servicio (*port*)
- Orientado a bytes, no a mensajes (*stream*): los bytes no se pierden, desordenan ni duplican **pero no garantiza *boundaries***
- Con conexión y full-duplex.
Análogo a un archivo binario secuencial.

Internet - Protocolo DNS (simplificado)

- No es necesario recordar la dirección IP del destino (4 o 16 bytes) sino su *nombre de dominio*.

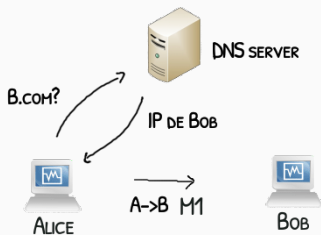


Internet - Protocolo DNS (simplificado)



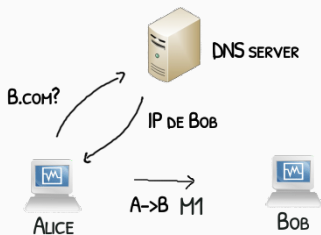
- No es necesario recordar la dirección IP del destino (4 o 16 bytes) sino su *nombre de dominio*.
- La registración de dominios se hace a nivel gubernamental. Para el caso **.com.ar** lo hace NIC, Cancillería Argentina.

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.

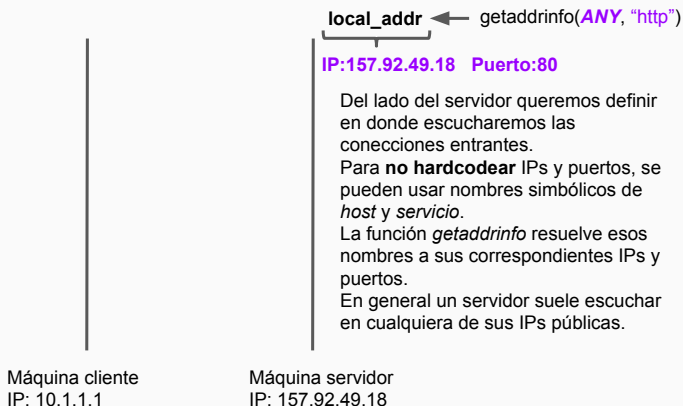
Internet - Protocolo DNS (simplificado)



- 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



Resolución de nombres: a quien me quiero conectar

`remote_addr` ← `getaddrinfo("fi.uba", "http")`

IP:157.92.49.18 Puerto:80

Del lado del cliente queremos definir a
quien nos queremos conectar.

Máquina cliente
IP: 10.1.1.1

Máquina servidor
IP: 157.92.49.18

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);
```

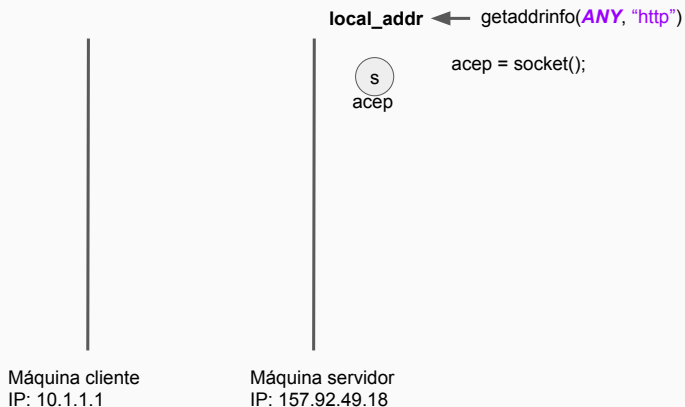
Servidor

```
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
```

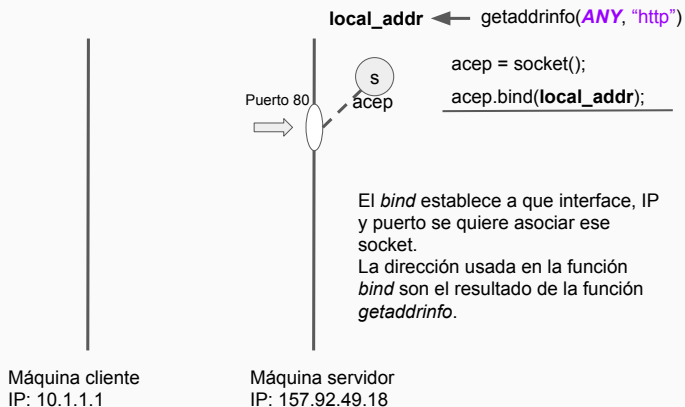
Canal de comunicación TCP

Establecimiento de un canal

Creación de un socket

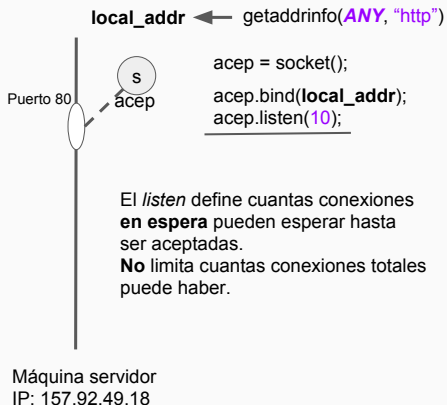


Enlazado de un socket a una dirección



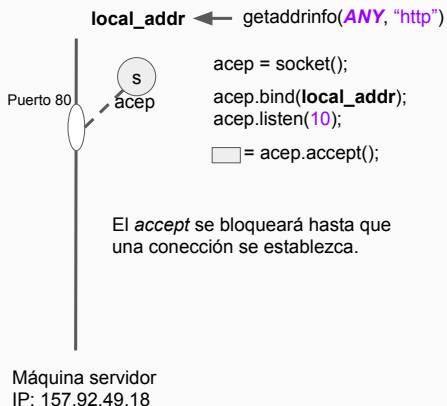
Socket aceptador o pasivo

Máquina cliente
IP: 10.1.1.1



Socket aceptador o pasivo

Máquina cliente
IP: 10.1.1.1



Conexión con el servidor: estableciendo conexión

`remote_addr` ← `getaddrinfo("fi.uba", "http")`

`cli = socket();`

`cli.connect(remote_addr);`



Puerto ??

El *connect* establece una conexión a la máquina remota.

De forma implícita hace un *bind* eligiendo una interfaz e IP válidas y un puerto libre al azar.

Es posible hacer un *bind* explícito si se desea.

Máquina cliente
IP: 10.1.1.1

`local_addr` ← `getaddrinfo(ANY, "http")`

Puerto 80



`acep = socket();`

`acep.bind(local_addr);`

`acep.listen(10);`

 `= acep.accept();`

Máquina servidor
IP: 157.92.49.18

Conexión con el servidor: aceptando la conexión

`remote_addr ← getaddrinfo("fi.uba", "http")`

`cli = socket();`

`cli.connect(remote_addr);`



Puerto ??

Máquina cliente
IP: 10.1.1.1

`local_addr ← getaddrinfo(ANY, "http")`

`accep = socket();`

`accep.bind(local_addr);`

`accep.listen(10);`

`srv = accep.accept();`

Puerto 80



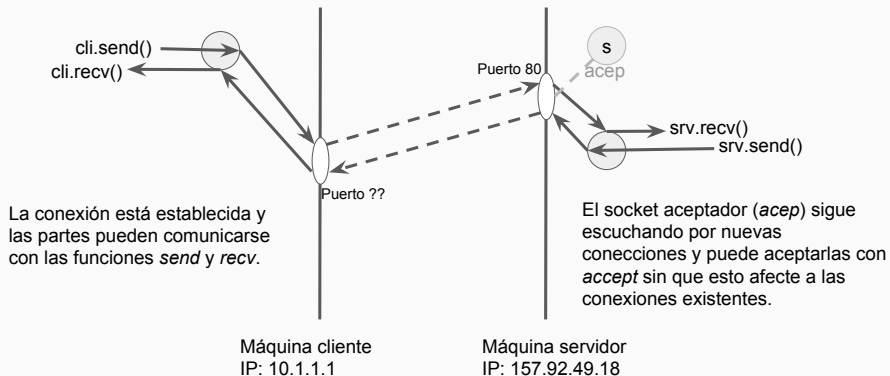
El *accept* se desbloquea
creandonos **un segundo socket**
que nos representa la nueva
conexión establecida.

Máquina servidor
IP: 157.92.49.18

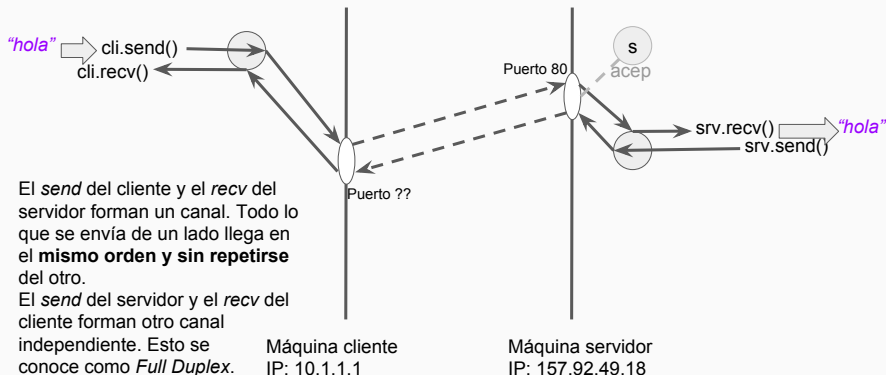
Canal de comunicación TCP

Envío y recepción de datos

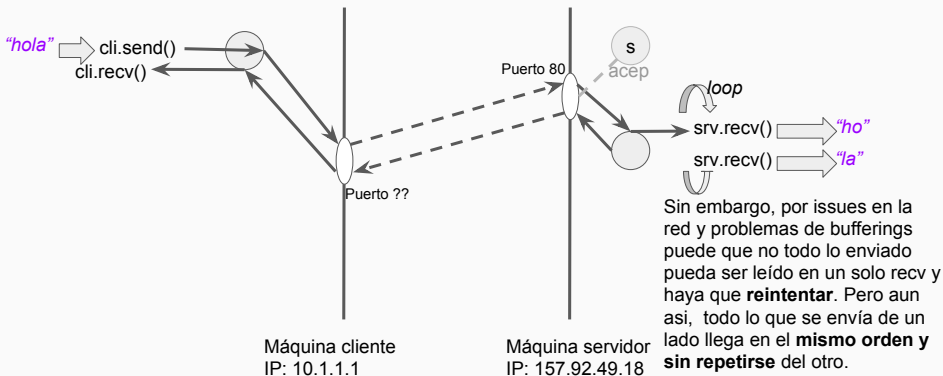
Conexión establecida



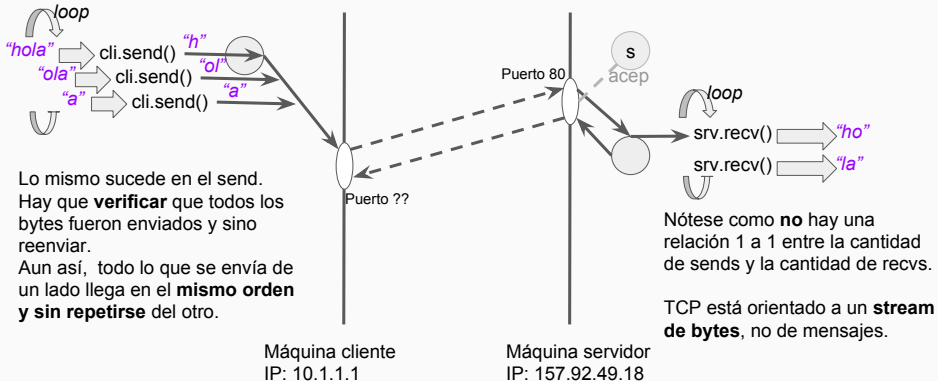
Envío y recepción de datos



Envío y recepción de datos en la realidad



Envío y recepción de datos en la realidad



Envío 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_rcv = 0;
3
4 while (MSG_LEN > bytes_rcv && skt_still_open) {
5     s = recv(skt, &buf[bytes_rcv], MSG_LEN - bytes_rcv - 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_rcv += s;
15    }
16 }
```

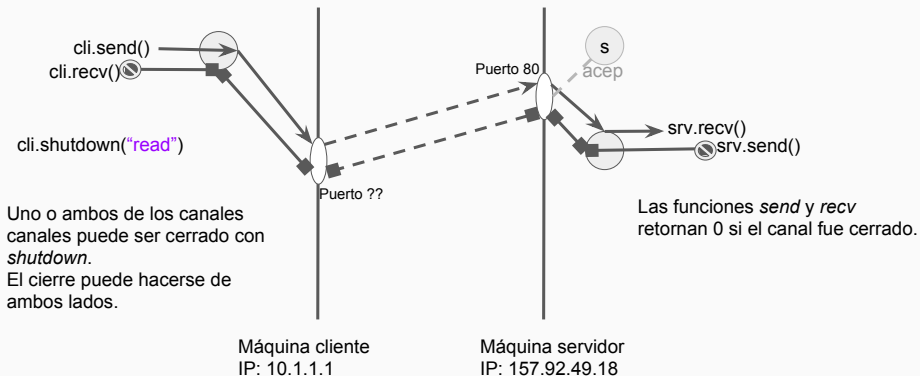
sendall: envío de N bytes exactos

```
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 }
```

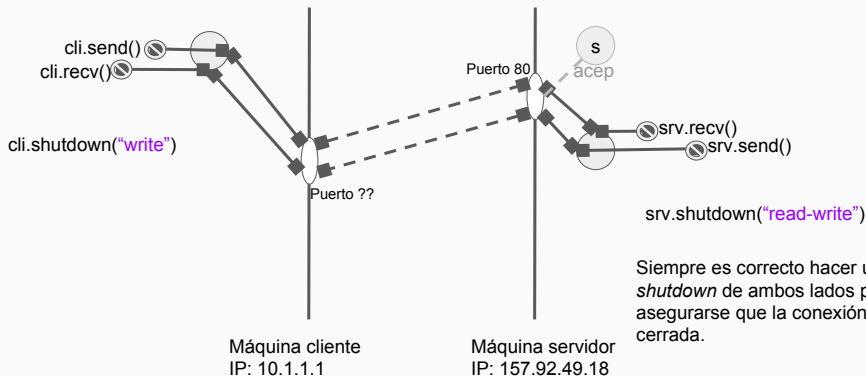
Canal de comunicación TCP

Finalización de un canal

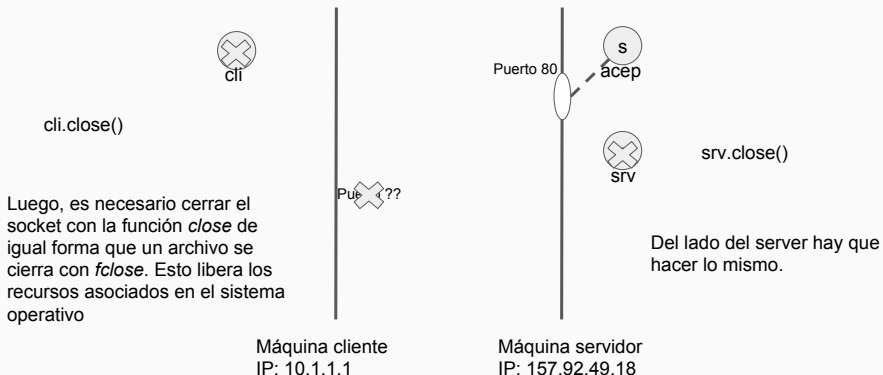
Cierre de conexión parcial



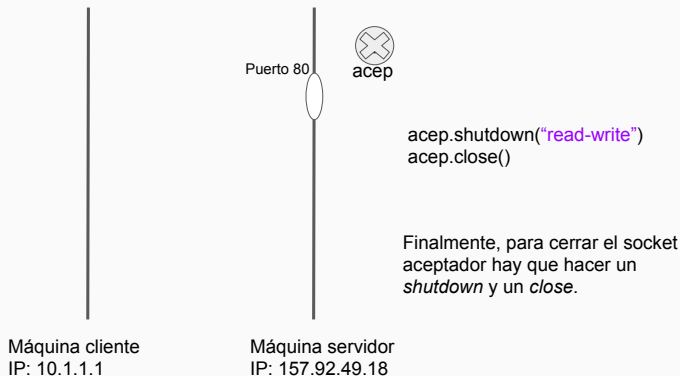
Cierre de conexión total



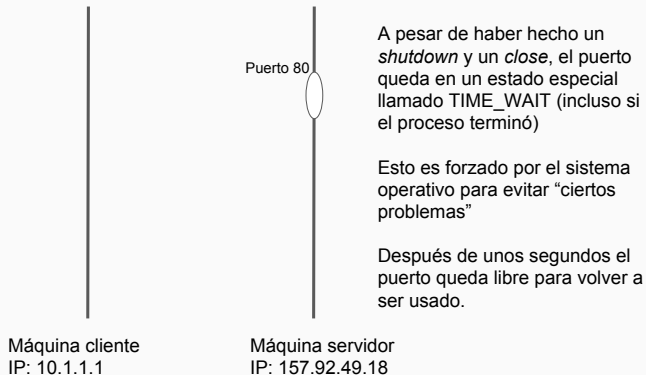
Libерación de los recursos con close



Cierre y liberación del socket aceptador



TIME WAIT



TIME WAIT -> Reuse Address

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

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

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

- 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 endianness, 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 endianness, el padding, los tamaños y los signos.

```
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?

```
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 endianness no coincide? y si hay padding entre los dos primeros campos?

Appendix

Referencias

Referencias I



man getaddrinfo



man netcat



man netstat



RFCs 971, 2460, ...



RFCs 793, ...



TCP/IP Illustrated, Richard Stevens



Data and Computer Communications, Ed Stallings