

Introducción a Sockets TCP en C

Di Paola Martín

`martinp.dipaola <at> gmail.com`

Facultad de Ingeniería
Universidad de Buenos Aires

De qué va esto?

Resolución de nombres

Canal de comunicación TCP

- Establecimiento de un canal

- Envío y recepción de datos

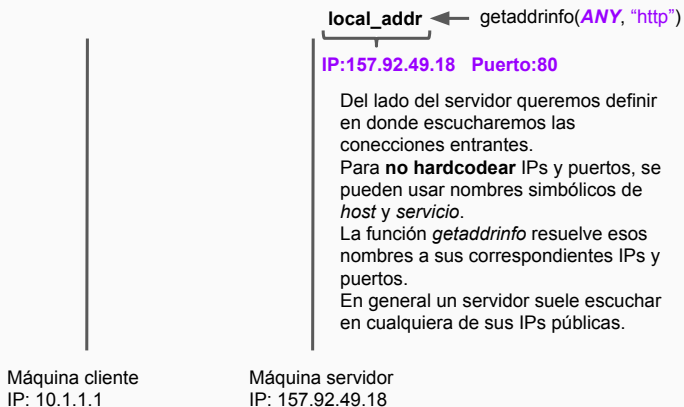
- Finalización de un canal

Protocolos y formatos

Netstat

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

Familias y tipos de sockets

- Familia **AF_UNIX**: para la comunicación entre procesos locales.

Familias y tipos de sockets

- Familia **AF_UNIX**: para la comunicación entre procesos locales.
- Familias **AF_INET** (IPv4) y **AF_INET6** (IPv6): para la comunicación a través de la Internet.

Familias y tipos de sockets

- Familia **AF_UNIX**: para la comunicación entre procesos locales.
- Familias **AF_INET** (IPv4) y **AF_INET6** (IPv6): para la comunicación a través de la Internet.
- Tipo **sock_DGRAM** (UDP): Sin conexión. Orientado a mensajes (datagramas). Los mensajes se pierden, duplican y llegan en desorden.

Familias y tipos de sockets

- Familia **AF_UNIX**: para la comunicación entre procesos locales.
- Familias **AF_INET** (IPv4) y **AF_INET6** (IPv6): para la comunicación a través de la Internet.
- Tipo **SOCK_DGRAM** (UDP): Sin conexión. Orientado a mensajes (datagramas). Los mensajes se pierden, duplican y llegan en desorden.
- Tipo **SOCK_STREAM** (TCP): Con conexión, full-duplex. Orientado al streaming. Los bytes llegan en orden y sin pérdidas. **Análogo a un archivo binario secuencial.**

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);
```

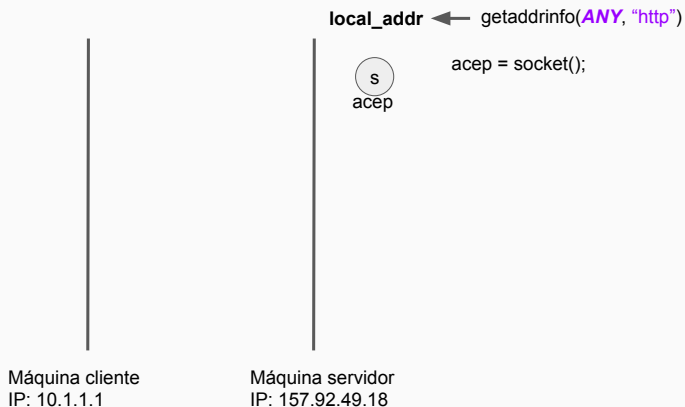
Servidor

```
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);
```

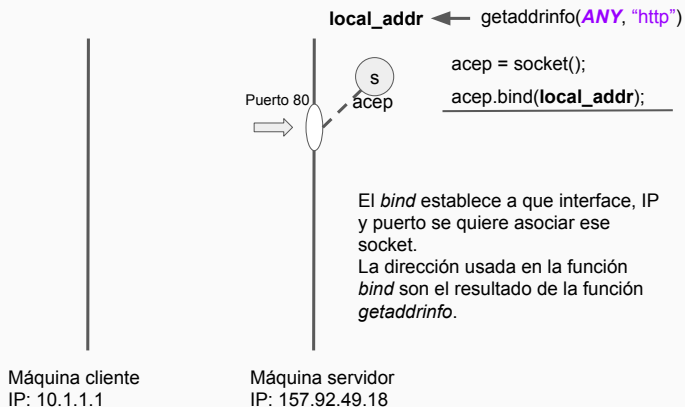
Canal de comunicación TCP

Establecimiento de un canal

Creación de un socket

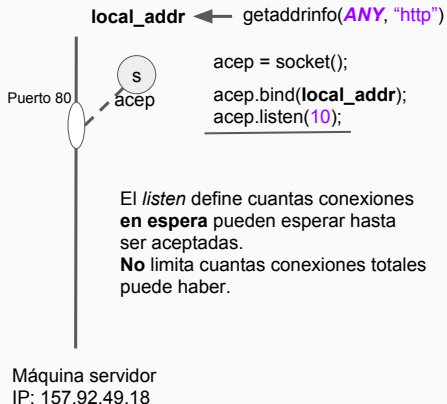


Enlazado de un socket a una dirección



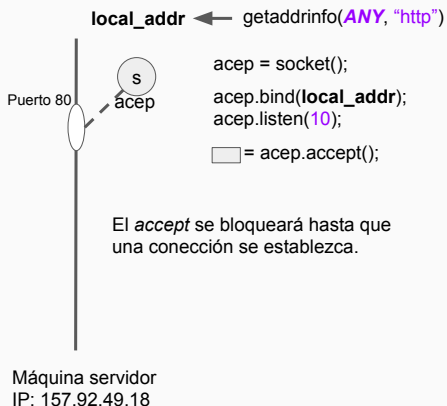
Socket aceptador o pasivo

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")`

`acep = socket();`

`acep.bind(local_addr);`

`acep.listen(10);`

`srv = acep.accept();`

Puerto 80



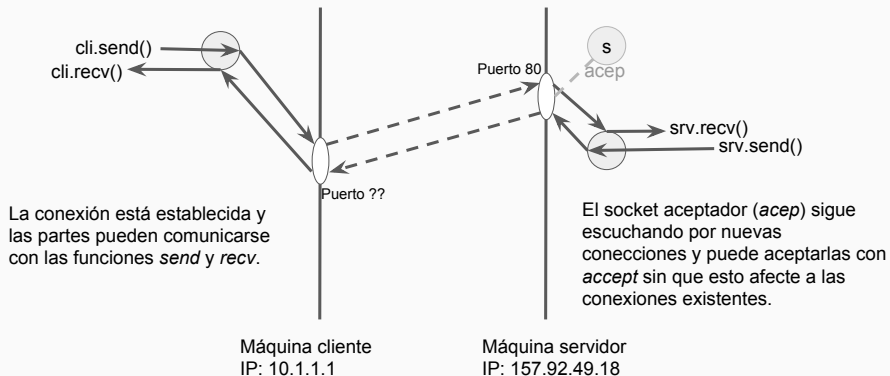
El *accept* se desbloquea
creandonos **un segundo socket**
que nos representa la nueva
conexión establecida.

Máquina servidor
IP: 157.92.49.18

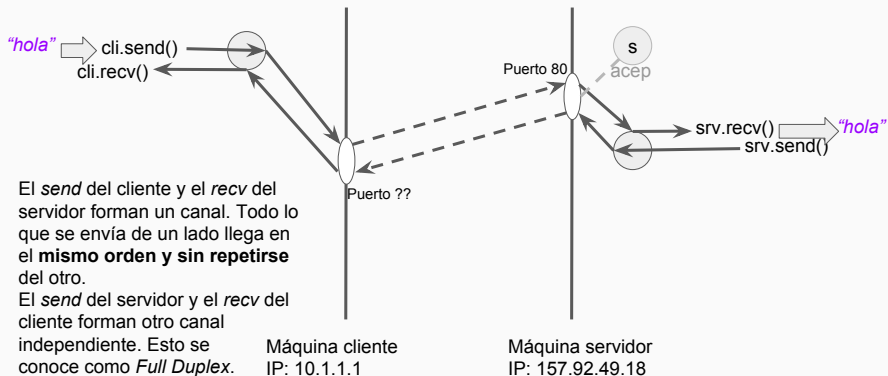
Canal de comunicación TCP

Envío y recepción de datos

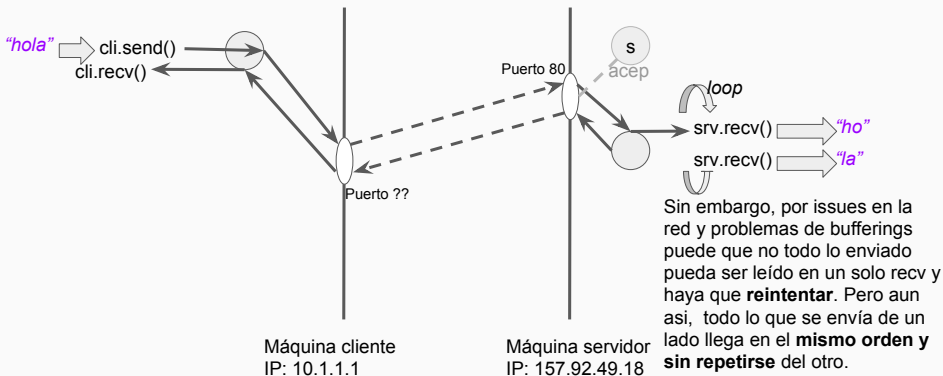
Conexión establecida



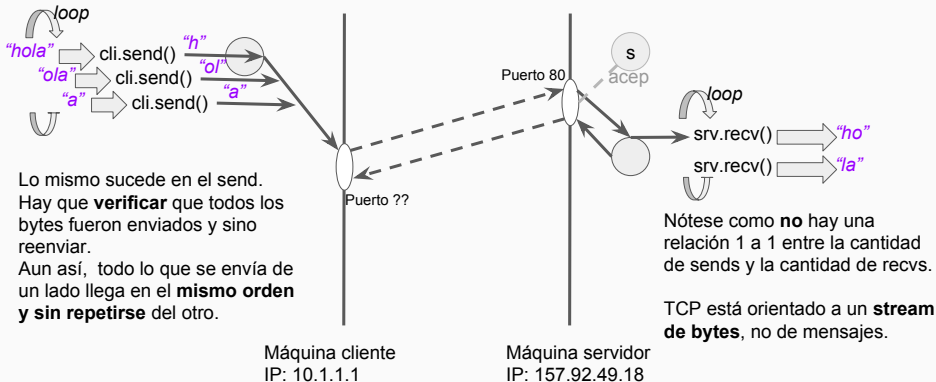
Envío y recepción de datos



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          // MSG_NOSIGNAL
11 );
12
13 (s < 0) // Error inesperado
14 (s == 0) // El socket fue cerrado
15 (s > 0) // Ok: s bytes fueron enviados/recibidos
```

Recepción de datos incremental

```
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 = rcv(skt, &buf[bytes_rcv], MSG_LEN - bytes_rcv - 1,
6                                     MSG_NOSIGNAL);
7     if (s < 0) { // Error inesperado
8         /* ... */
9     }
10    else if (s == 0) { // Nos cerraron el socket
11        /* ... */
12    }
13    else {
14        bytes_rcv += s;
15    }
16 }
```

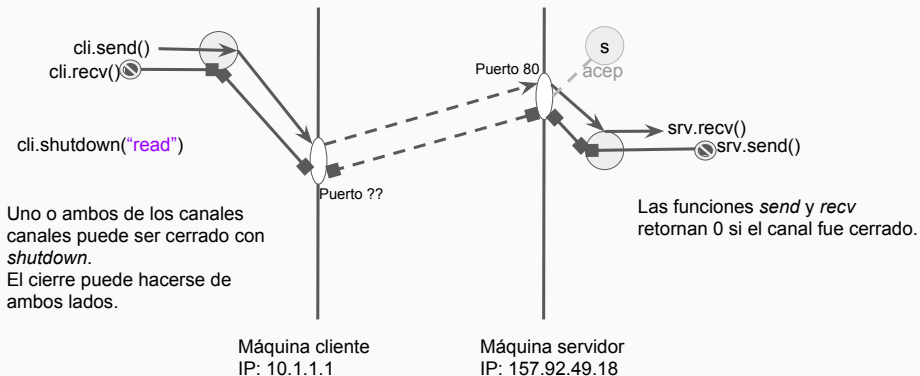

Envío de datos incremental

```
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 < 0) { // Error inesperado
8         /* ... */
9     }
10    else if (s == 0) { // Nos cerraron el socket
11        /* ... */
12    }
13    else {
14        bytes_sent += s;
15    }
16 }
```

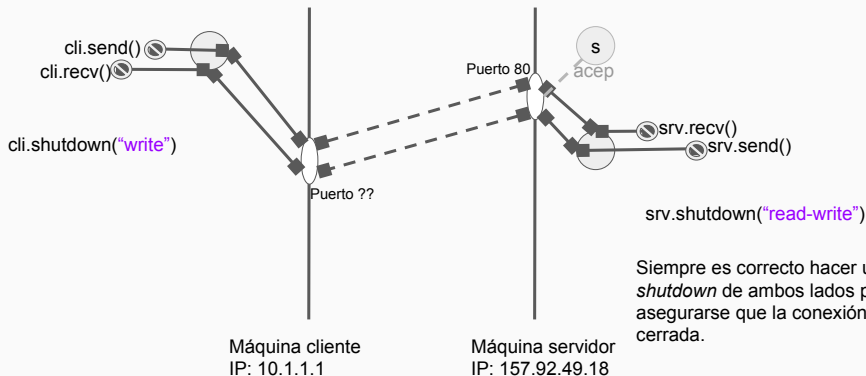
Canal de comunicación TCP

Finalización de un canal

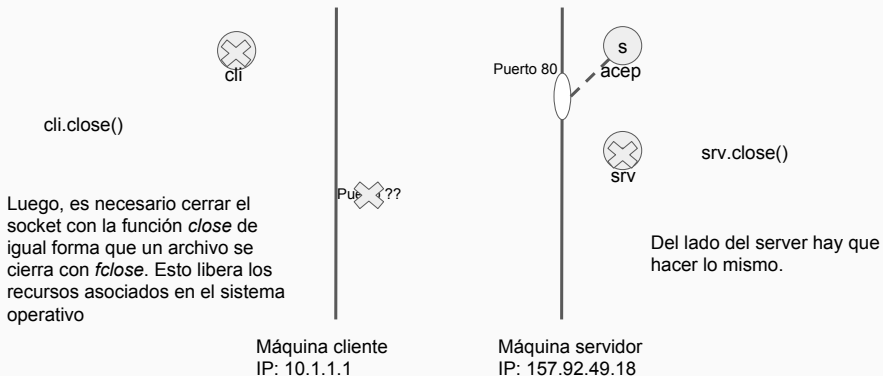
Cierre de conexión parcial



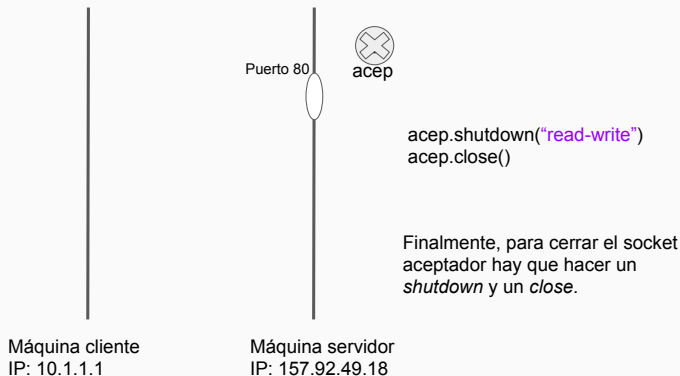
Cierre de conexión total



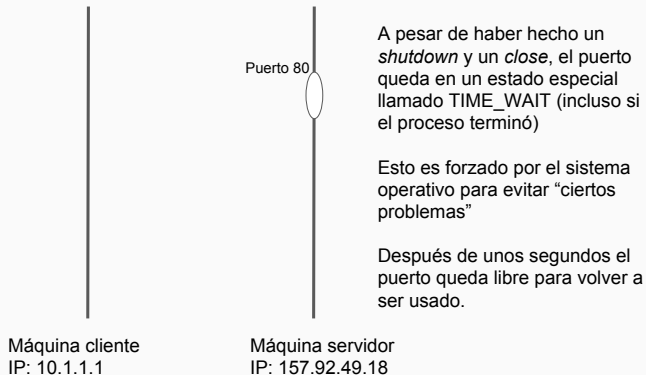
Libерación de los recursos con close



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 acep    = socket(...);  
2 | int status = bind(acep, ...); //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 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.
- 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.

Longitud variable: delimitador

Delimitador: el mensaje no tiene un tamaño fijo y el fin del mensaje esta marcado por un delimitador.

```
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?

Longitud variable: prefijo con la longitud

```
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?

Netstat

Netstat

```
1 machineA$ nc -l 1234 &
2 machineA$ nc -l 8080 &
3 machineA$ nc 127.0.0.1 8080 &
4
5 machineA$ netstat -tauon
6 Active Internet connections (servers and established)
7 Proto Local Address          Foreign Address      State
8 tcp    127.0.0.1:1234      0.0.0.0:*            LISTEN
9 tcp    127.0.0.1:8080      127.0.0.1:33036      ESTABLISHED
10 tcp    127.0.0.1:33036     127.0.0.1:8080      ESTABLISHED
```

```
1 machineA$ sudo killall -9 nc
2
3 machineA$ netstat -tauon
4 Active Internet connections (servers and established)
5 Proto Local Address          Foreign Address      State
6 tcp    127.0.0.1:8080       127.0.0.1:33036      TIME_WAIT
```


Appendix

Referencias

Referencias I



man getaddrinfo



man netcat



man netstat