Introducción a Sockets TCP en C

Di Paola Martín

martinp.dipaola <at> gmail.com

7542 - Taller de Programación I
Facultad de Ingeniería
Universidad de Buenos Aires

De qué va esto?

Resolución de nombres

Canal de comunicación TCP

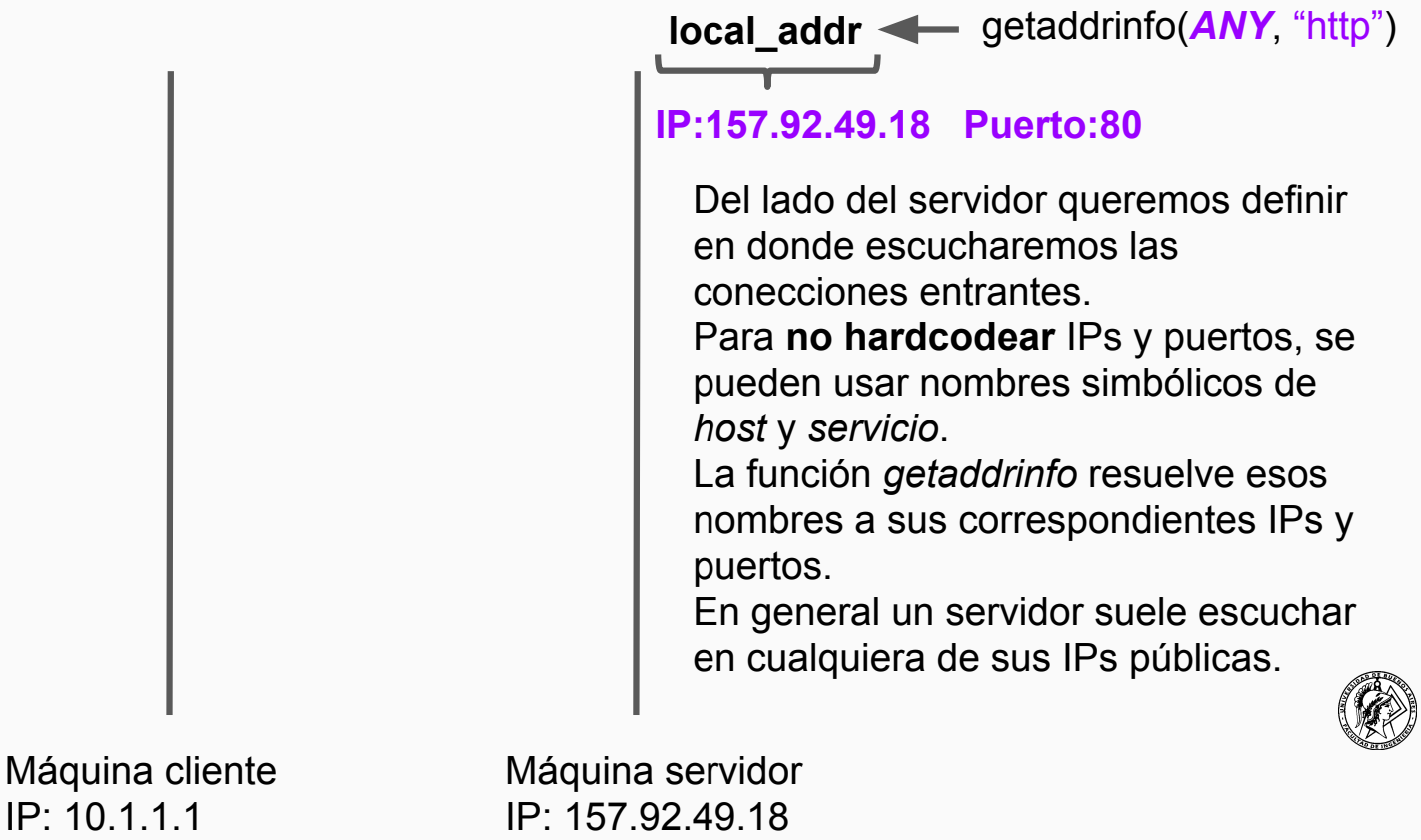
- Establecimiento de un canal
- Envio y recepción de datos
- Finalización de un canal

Protocolos y formatos

Netstat

Resolución de nombres

Resolución de nombres: desde donde quiero escuchar

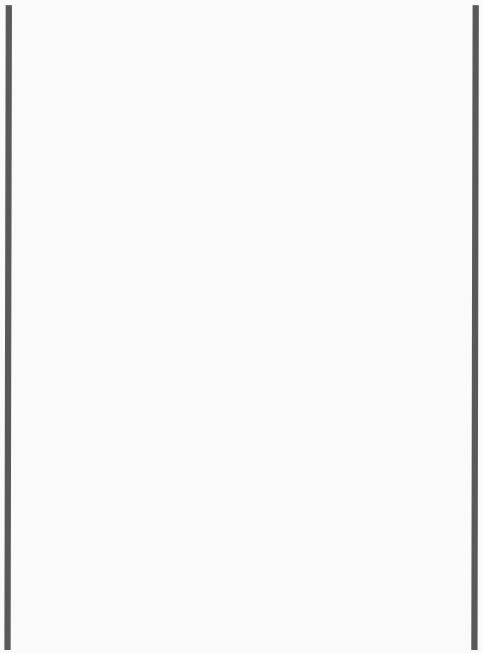


Resolución de nombres: a quien me quiero conectar

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 traves 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 traves de la Internet.

- Tipo **`SOCK_DGRAM`** (UDP): Sin conección. 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 traves de la Internet.

- Tipo **`SOCK_DGRAM`** (UDP): Sin conección. Orientado a mensajes (datagramas). Los mensajes se pierden, duplican y llegan en desorden.
- Tipo **`SOCK_STREAM`** (TCP): Con conección, 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);
```

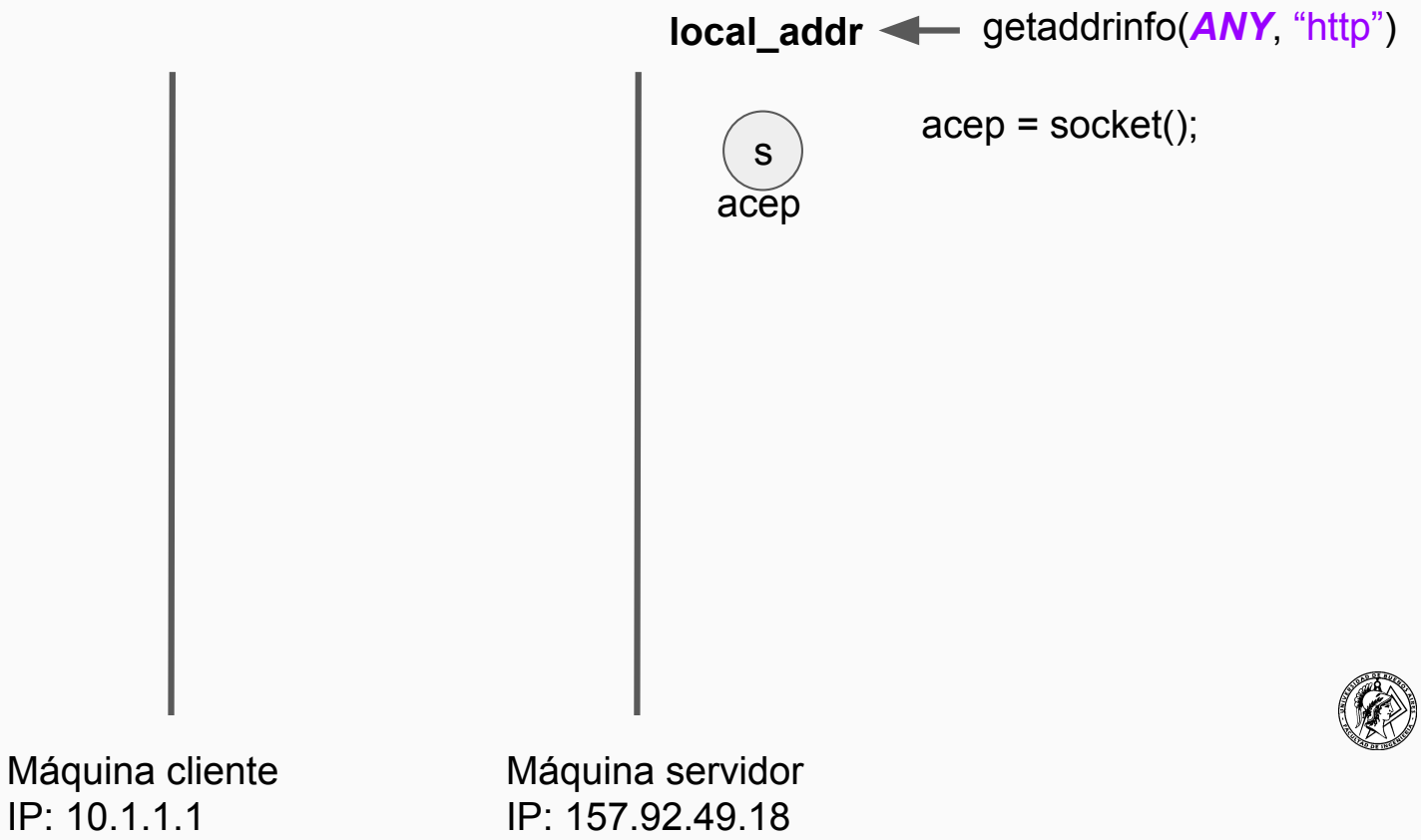
Servidor

```
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);
```

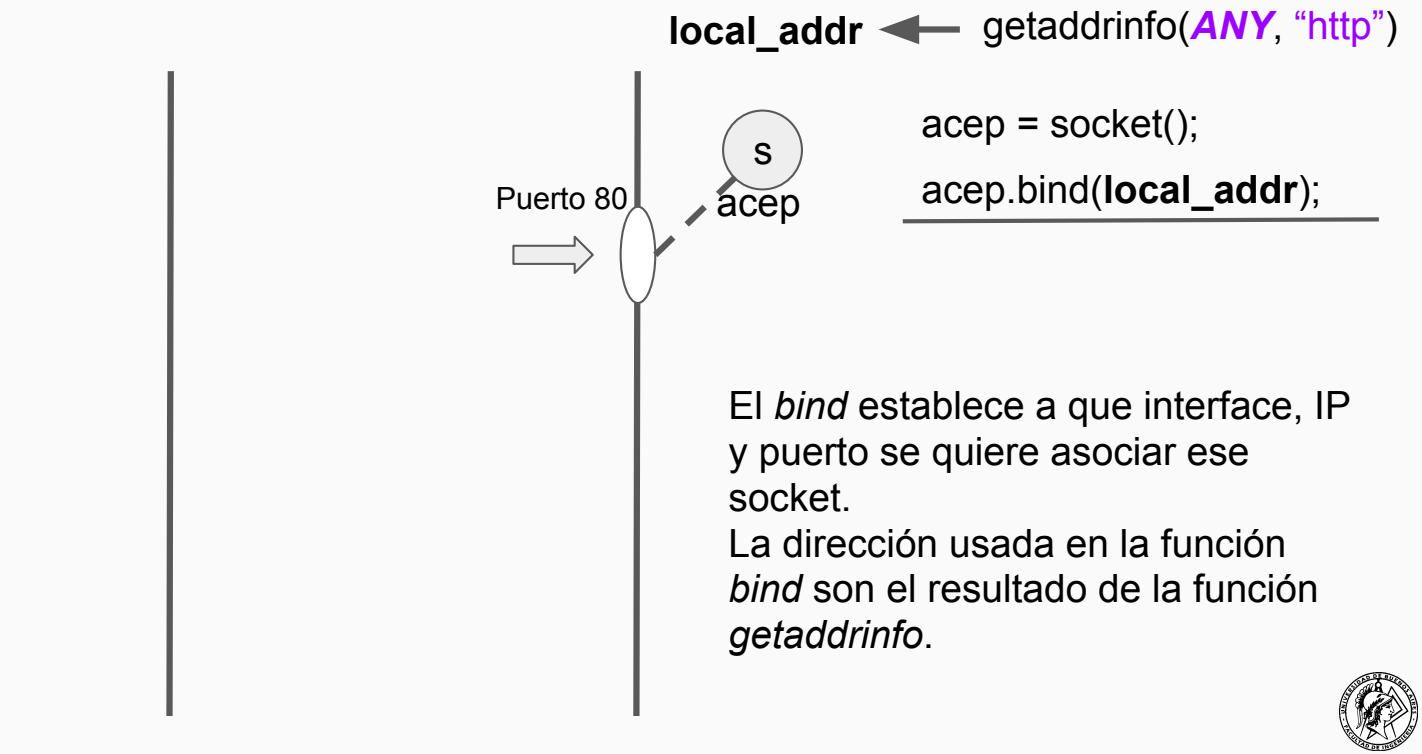
Canal de comunicación TCP

Establecimiento de un canal

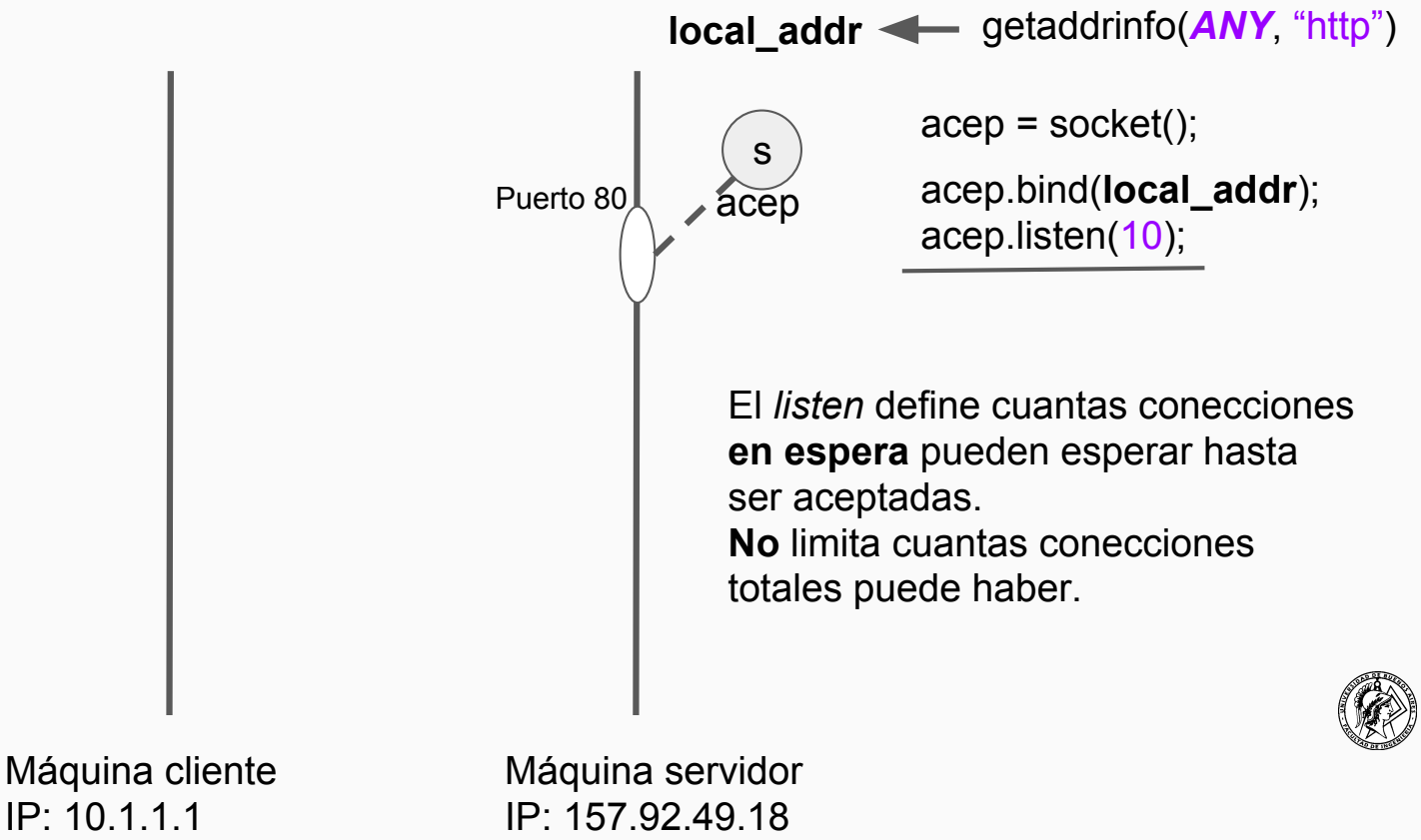
Creación de un socket



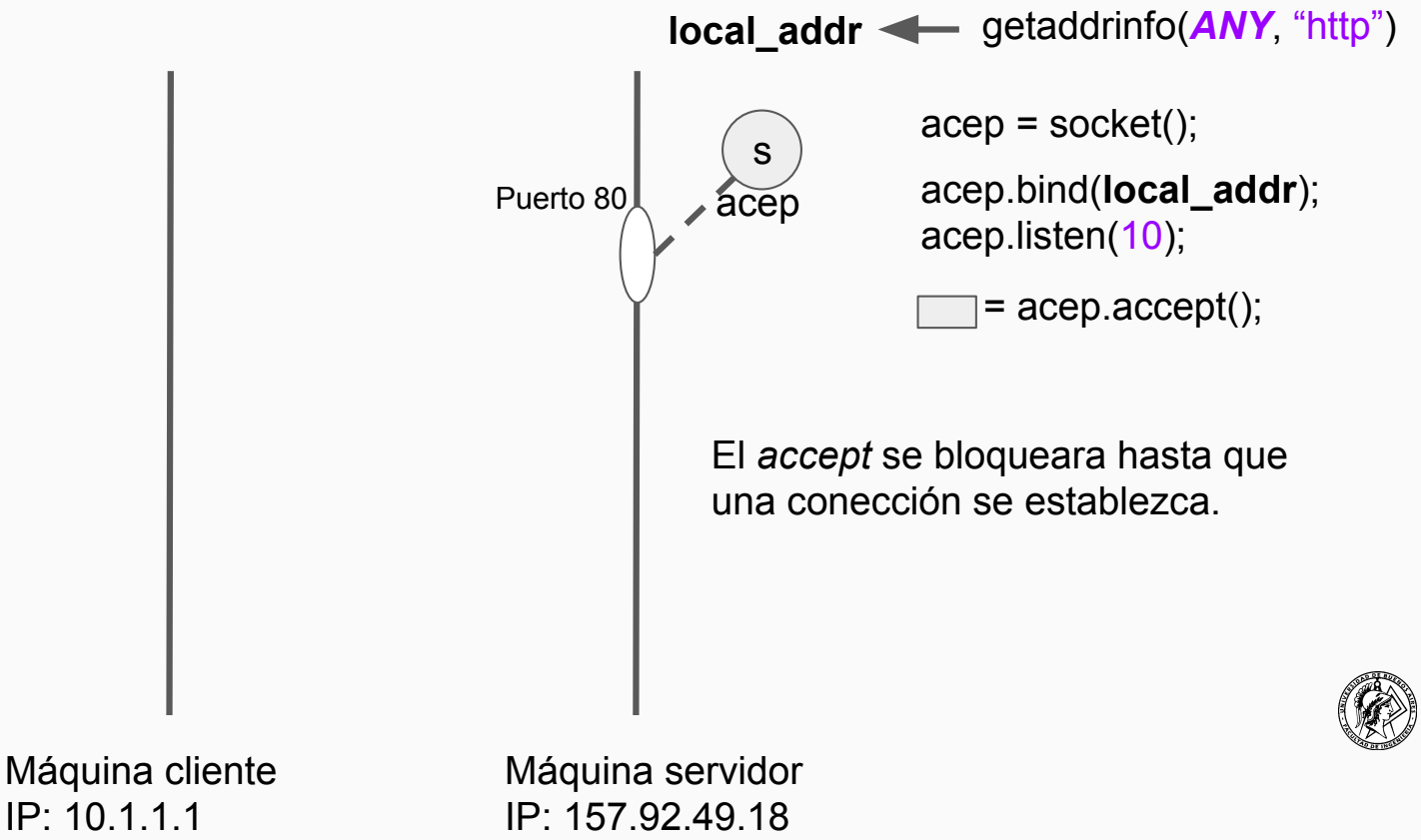
Enlazado de un socket a una dirección



Socket aceptador o pasivo



Socket aceptador o pasivo



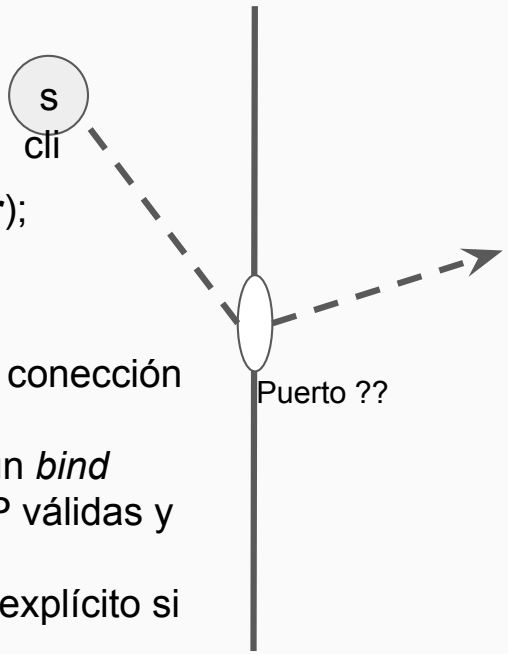
Conección con el servidor

remote_addr ← getaddrinfo(“fi.uba”, “http”)

cli = socket();

cli.connect(**remote_addr**);

El *connect* establece una conección 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.



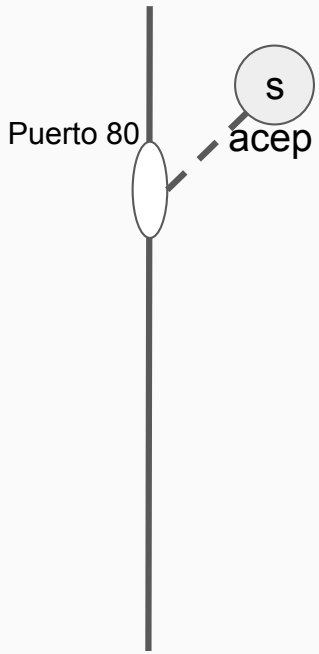
Máquina cliente
IP: 10.1.1.1

local_addr ← getaddrinfo(***ANY***, “http”)

acep = socket();

acep.bind(**local_addr**);
acep.listen(10);

= acep.accept();



Máquina servidor
IP: 157.92.49.18

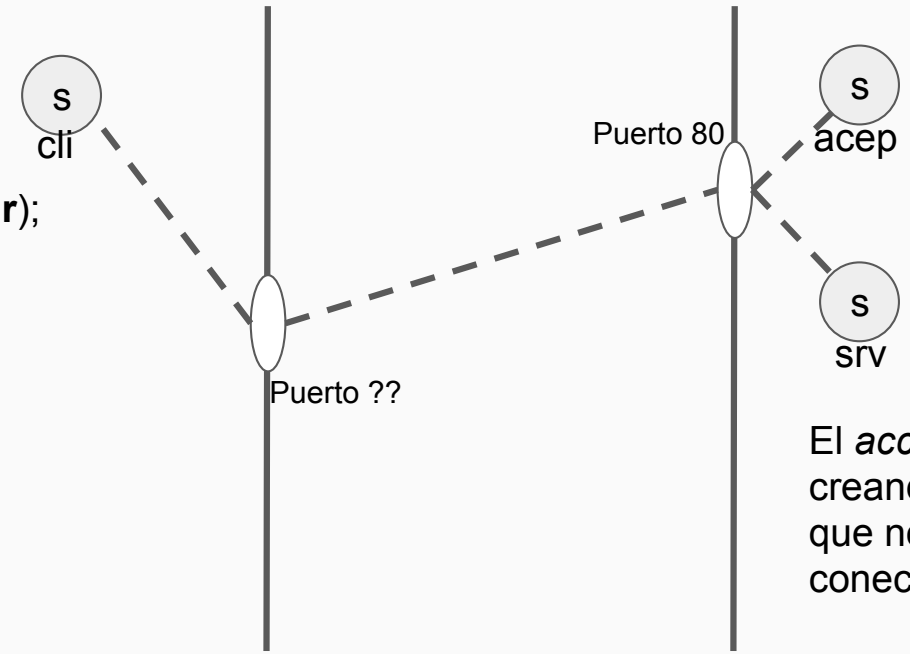
Conección con el servidor

remote_addr ← getaddrinfo(“fi.uba”, “http”)

local_addr ← getaddrinfo(***ANY***, “http”)

cli = socket();

cli.connect(**remote_addr**);



acep = socket();

acep.bind(**local_addr**);
acep.listen(10);

srv = acep.accept();

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

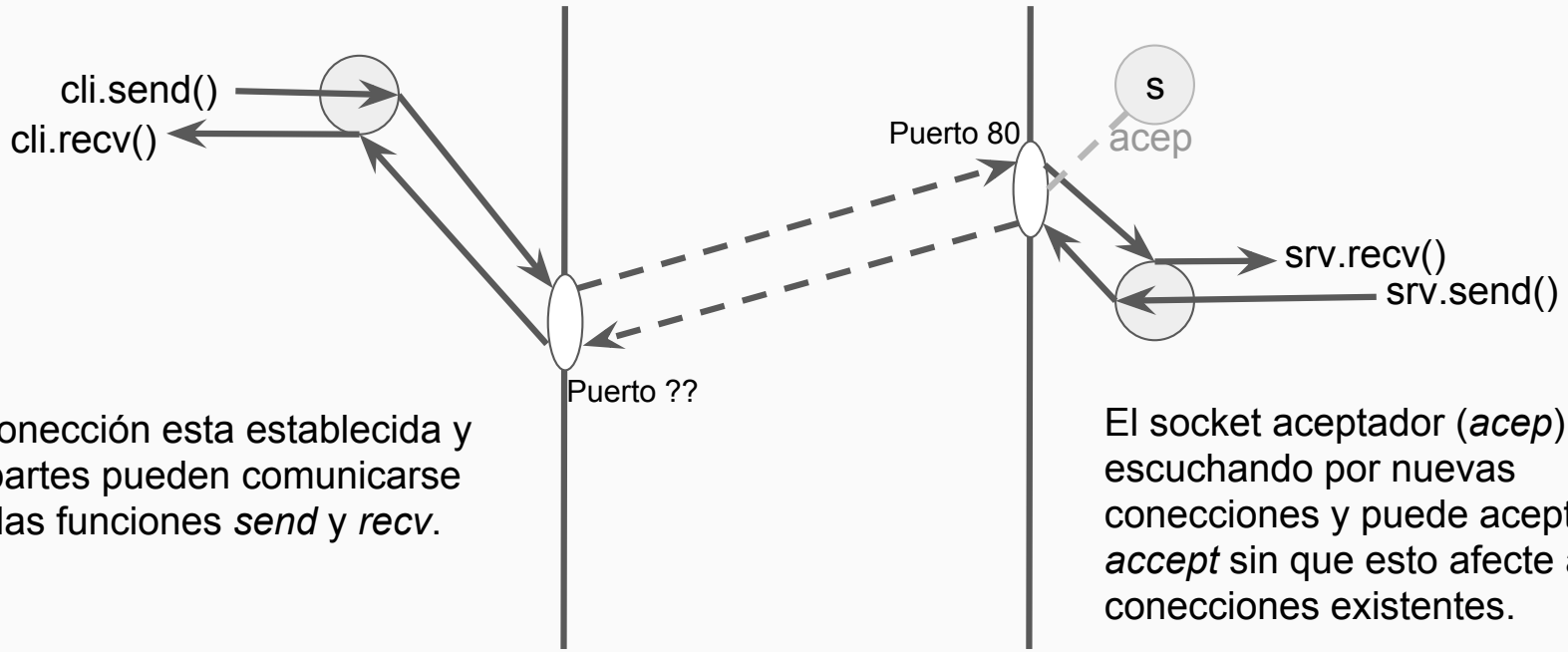
Máquina cliente
IP: 10.1.1.1

Máquina servidor
IP: 157.92.49.18

Canal de comunicación TCP

Envio y recepción de datos

Conección establecida



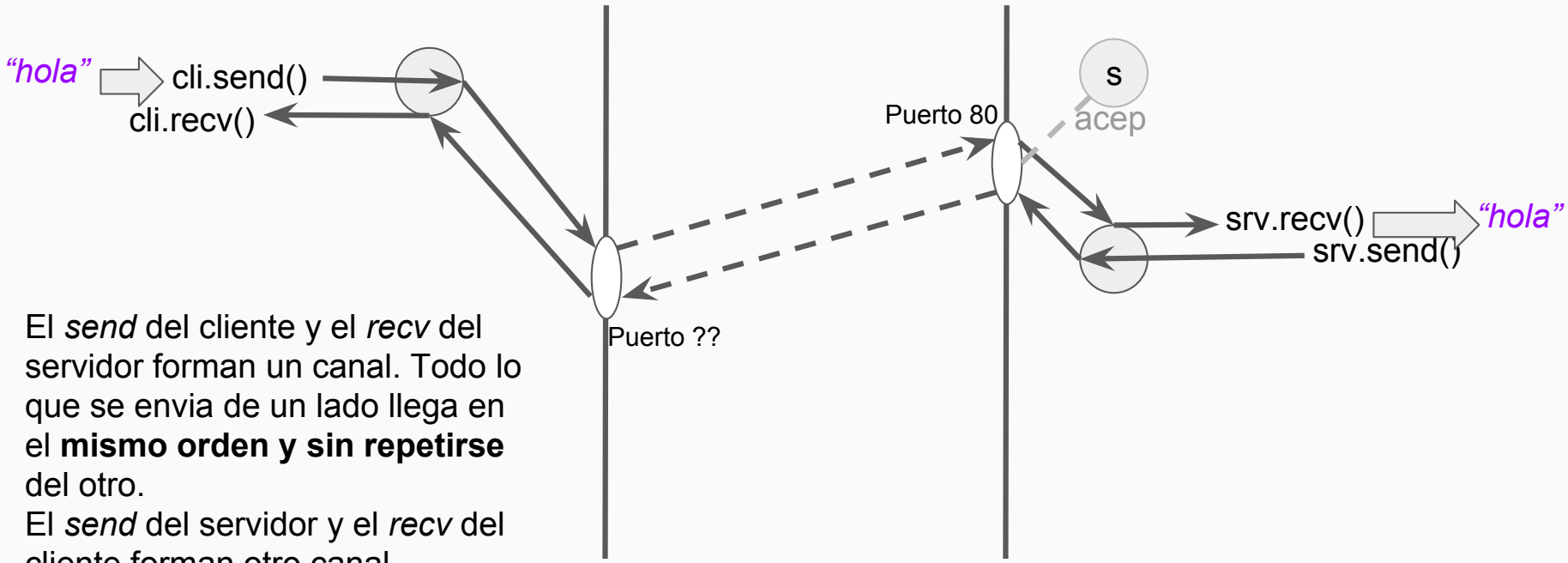
La conección esta establecida y las partes pueden comunicarse con las funciones *send* y *recv*.

El socket aceptador (*acep*) sigue escuchando por nuevas conecciones y puede aceptarlas con *accept* sin que esto afecte a las conecciones existentes.

Máquina cliente
IP: 10.1.1.1

Máquina servidor
IP: 157.92.49.18

Envio y recepción de datos

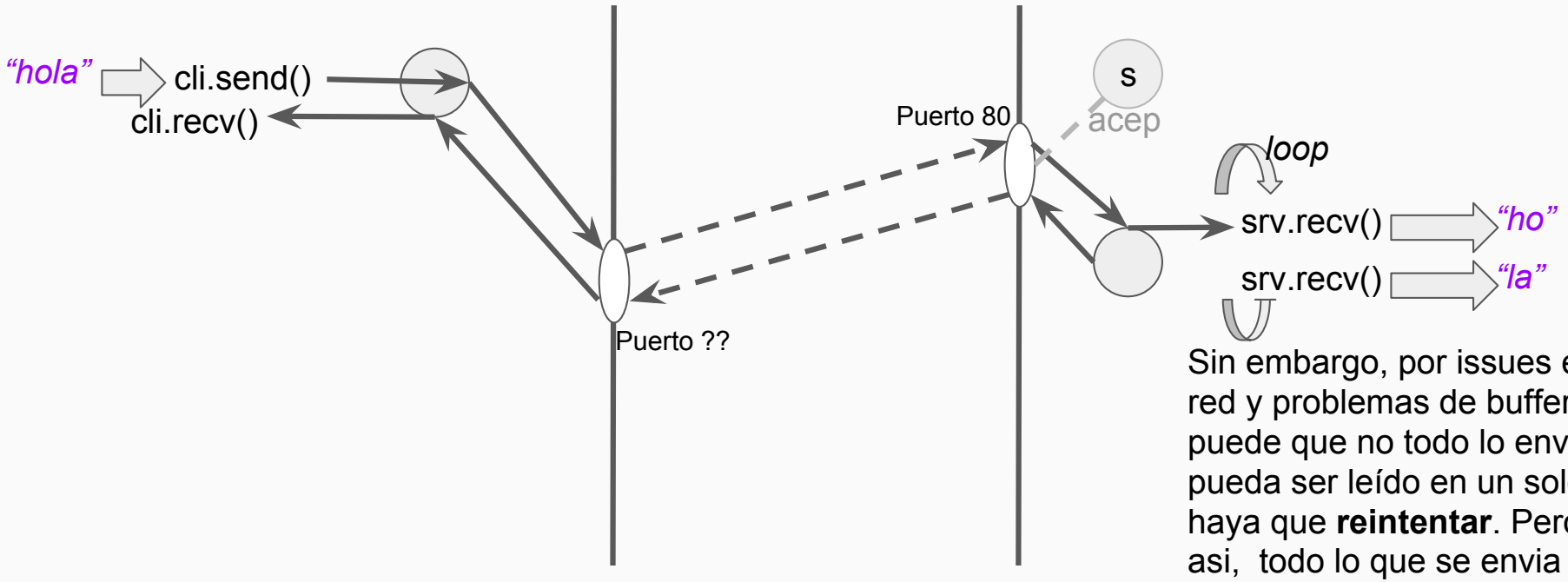


El *send* del cliente y el *recv* del servidor forman un canal. Todo lo que se envia de un lado llega en el **mismo orden y sin repetirse** del otro.
El *send* del servidor y el *recv* del cliente forman otro canal independiente. Esto se conoce como *Full Duplex*.

Máquina cliente
IP: 10.1.1.1

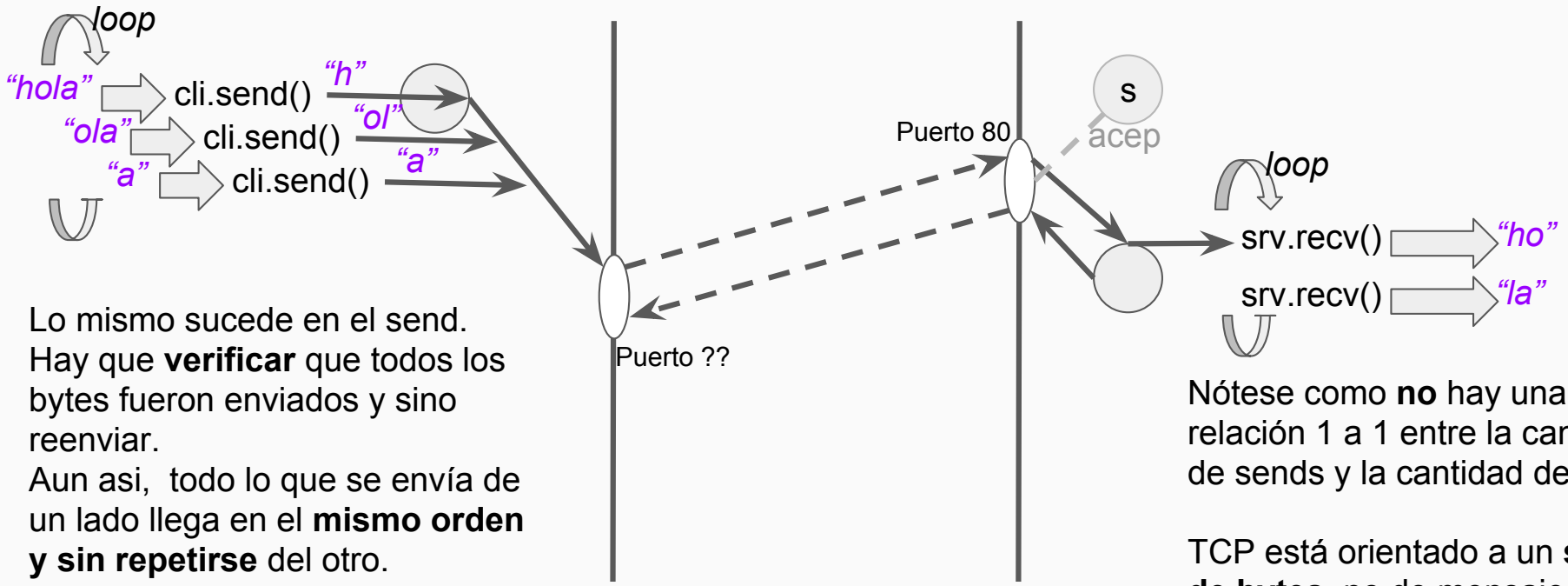
Máquina servidor
IP: 157.92.49.18

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 envia 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 asi, 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

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           // 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_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                                                  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_recv += s;
15   }
16 }
```

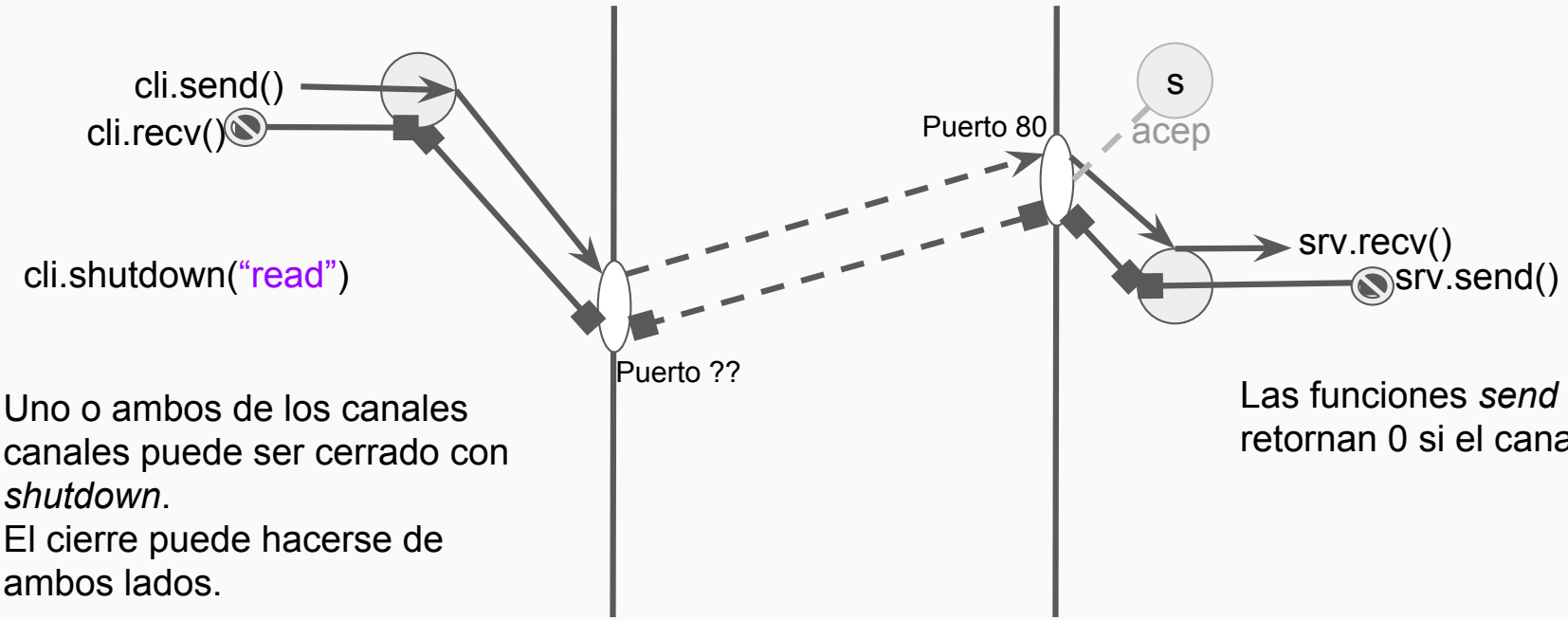
Envio de datos incremental

```
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 conección parcial



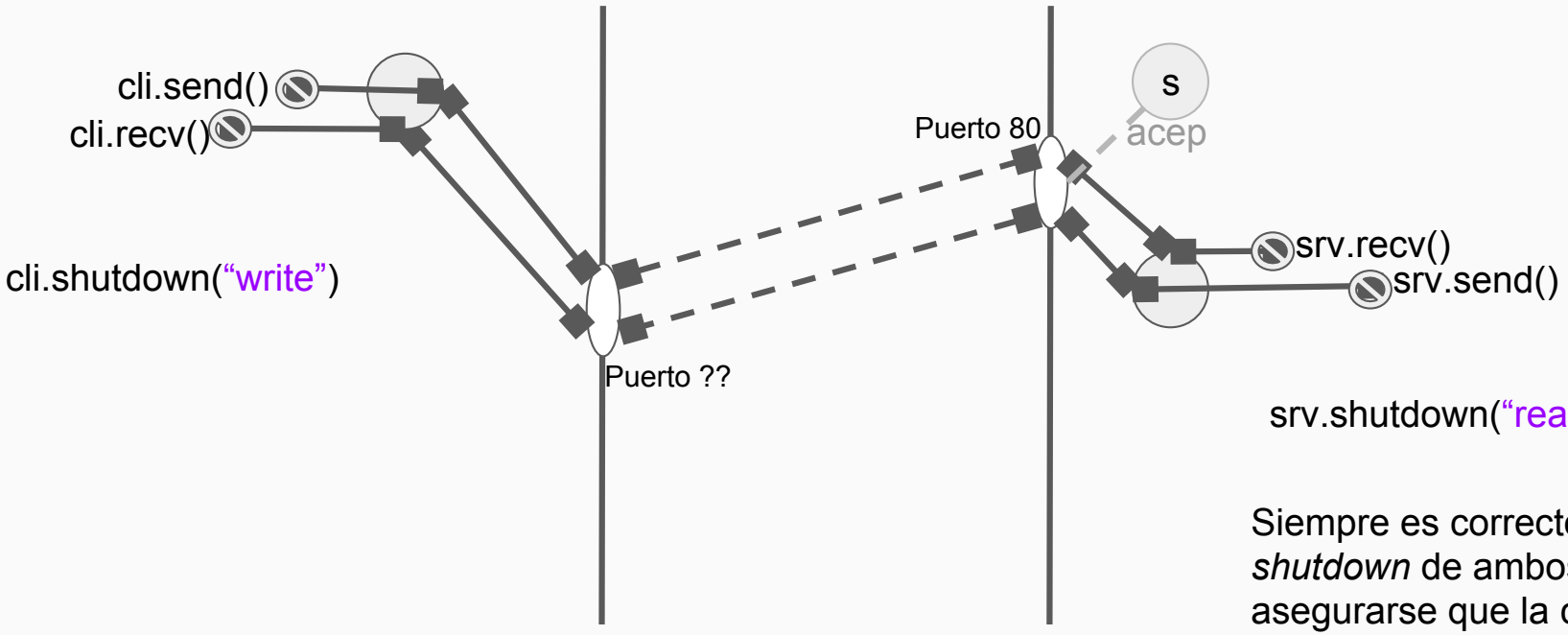
Uno o ambos de los canales canales puede ser cerrado con *shutdown*.
El cierre puede hacerse de ambos lados.

Las funciones *send* y *recv* retornan 0 si el canal fue cerrado.

Máquina cliente
IP: 10.1.1.1

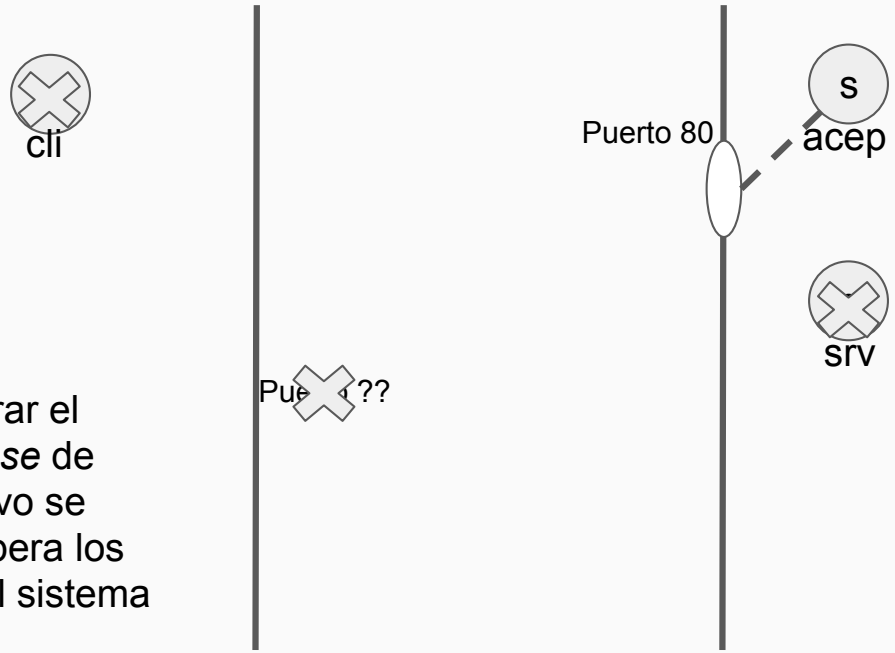
Máquina servidor
IP: 157.92.49.18

Cierre de conección total



Siempre es correcto hacer un *shutdown* de ambos lados para asegurarse que la conección esta cerrada.

Liberación de los recursos con close



cli.close()

srv.close()

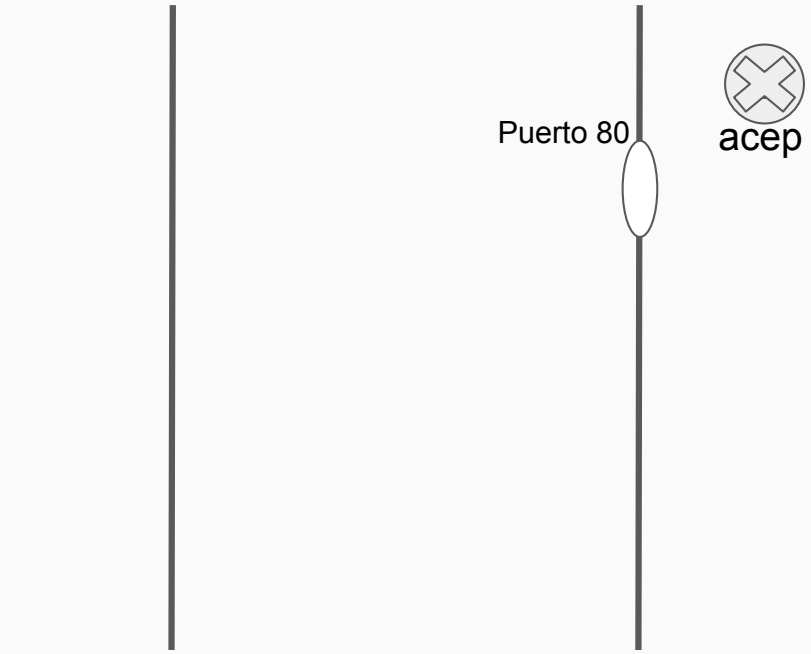
Luego, es necesario cerrar el socket con la función *close* de igual forma que un archivo se cierra con *fclose*. Esto libera los recursos asociados en el sistema operativo

Del lado del server hay que hacer lo mismo.

Máquina cliente
IP: 10.1.1.1

Máquina servidor
IP: 157.92.49.18

Cierre y liberación del socket aceptador



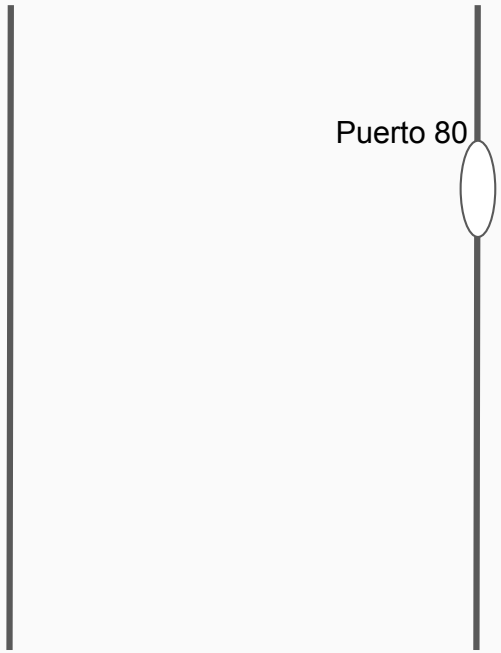
acep.shutdown(“read-write”)
acep.close()

Finalmente, para cerrar el socket aceptador hay que hacer un *shutdown* y un *close*.

Máquina cliente
IP: 10.1.1.1

Máquina servidor
IP: 157.92.49.18

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”

Despues 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
```

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 Binario: son simples y eficientes en terminos de memoria y velocidad de procesamiento. Son más difíciles de debuggear. Es necesario tomar en consideración items como el endianess, el padding, los tamaños y el signo.
- Protocolos en Texto: son la contracara de los protocolos binarios, son lentos, ineficientes y más difíciles de parsear pero más fáciles de debuggear.

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
9  msj.value = (char*) malloc(msj.length);
10 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`?
- Que pasa si el endianess no coincide? y si hay padding entre los dos primeros campos?

Netstat

Netstat

```
machineA$ nc -l 1234 &
machineA$ nc -l 8080 &
machineA$ nc 127.0.0.1 8080 &

machineA$ netstat -tauon
Active Internet connections (servers and established)
Proto Local Address           Foreign Address         State
tcp   127.0.0.1:1234          0.0.0.0:*               LISTEN
tcp   127.0.0.1:8080          127.0.0.1:33036         ESTABLISHED
tcp   127.0.0.1:33036         127.0.0.1:8080          ESTABLISHED
```

Netstat

```
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