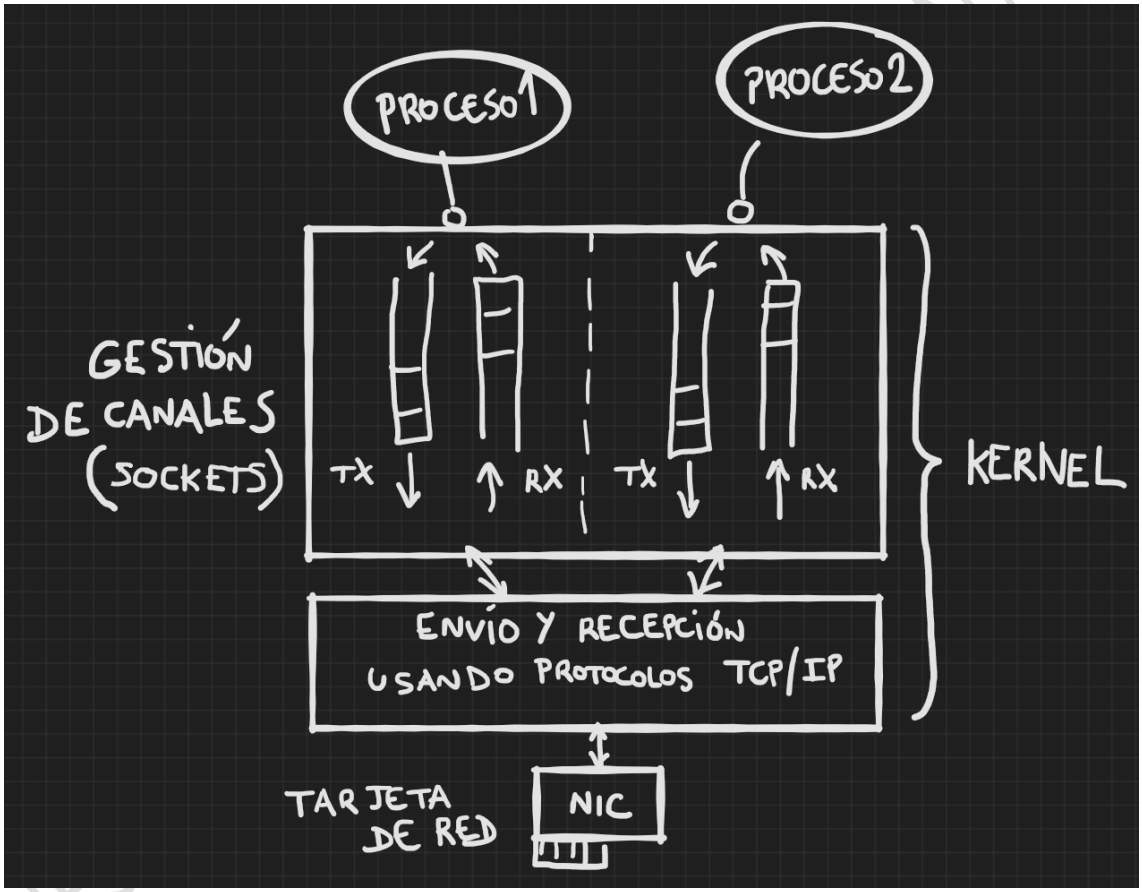
COMUNICACIÓN ENTRE PROCESOS (2ª PARTE): CANALES PARA COMUNICACIONES NO FIABLES POR INTERNET

Ha llegado el momento de que nuestros datos abandonen nuestra propia máquina y viajen a través de la red de redes. Cuando se diseñó Internet, se establecieron una serie de normas para que los datos de un proceso que se ejecute en un equipo conectado, puedan recibirse por cualquier otro proceso que esté corriendo en otro equipo también conectado.

Para nuestra profesión, es muy importante conocer bien las normas que permiten que los datos viajen por la red. Sin embargo, no todos los programadores que hacen código para comunicarse por Internet tienen esas necesidades, así que hay dos filosofías para enseñar cómo enviar y recibir datos por la red de redes. Mira la figura:



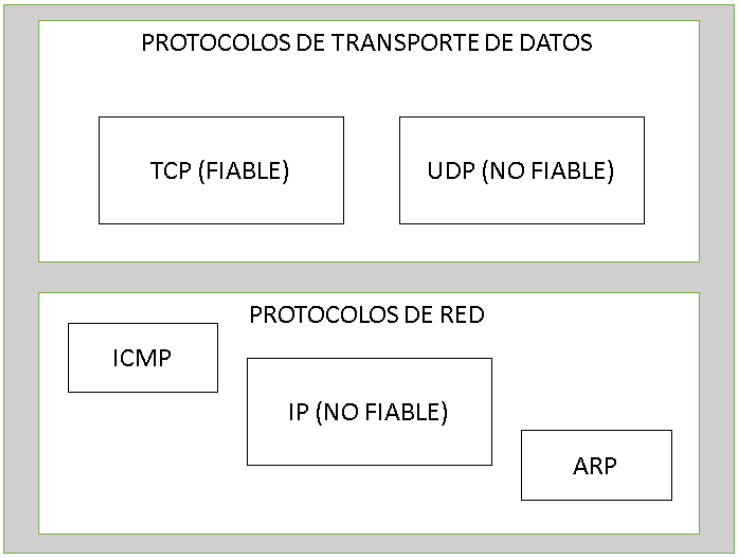
Lo que podéis ver es la infraestructura que se crea para las comunicaciones de Internet. Básicamente, un proceso le pedirá al sistema operativo que cree un CANAL que se llama socket, y lo identificará a través de un descriptor de fichero. Internamente, el kernel gestionará cada canal solicitado (un proceso puede pedir y utilizar más de un socket, por supuesto).

Si os dais cuenta, en este caso, los canales para trabajar en la red son BIDIRECCIONALES. ¿Qué quiere decir esto? Que el kernel mantiene tuberías separadas para los datos que se envían y los que se van recibiendo, pero el proceso utilizará el mismo descriptor para cuando quiera escribir o leer del socket. Esto está así para simplificar un poco más todo el procedimiento: en Internet se asume, como veremos más adelante, que el diálogo que establecen dos procesos es casi

siempre bidireccional. Se preguntan y se responden, así que en vez de que el programa tenga que abrir dos tuberías por sentido para eso, el kernel lo hace ya automáticamente, por defecto.

En una filosofía muy orientada al programador, se esconden todos los detalles de cómo se envían y reciben los datos utilizando la tarjeta de red (NIC: network interface card, en inglés). Esos "detalles" son algoritmos implementados dentro del kernel, que permiten seguir las normas de comunicación para que las máquinas se entiendan entre sí: los protocolos.

Estudiaremos los protocolos que permiten que las máquinas y procesos se comuniquen usando Internet en el Tema 4. Simplemente, como curiosidad, os diré que existen varios protocolos, y que cada uno de ellos se encarga de resolver un problema de la comunicación. Y tienen nombres: TCP, UDP, IP, ARP[1]... Cuando se habla de protocolos de Internet, se suele hablar de pila o *suite* TCP/IP, para referirnos a todos ellos (no solo a TCP e IP, pero son los más famosos y se erigen como "representantes" del resto). Esta funcionalidad TCP/IP es la que aparece representada en la figura anterior, como una caja entre la tarjeta de red y los sockets de envío y recepción del kernel creados para los procesos. Si queréis que echemos un vistazo a lo que hay dentro de la caja, es esto (atención, SPOILERS):



Pero, como os digo, revisitaremos el diseño y el software asociado a estos protocolos en el tema siguiente. Si hubiésemos seguido otra filosofía para aprender las comunicaciones de Internet, habríamos empezado por aquí, enseñando primero "las tripas" y, mucho después, hubiésemos empezado a programar. Realmente, esta sería la filosofía más acorde con la forma en que se articulan los planes de estudio de las Ingenierías de Telecomunicación, pero, ciertamente, retrasa el momento de creación de nuestros programas. Es cuestión de gustos porque, siempre que se explique todo bien, se debe llegar al mismo sitio.

[1] TCP: transmission control protocol; UDP: user datagram protocol; IP: internet protocol; ARP: address resolution protocol

COMO DISEÑAR PROGRAMAS PARA INTERNET: EL MODELO CLIENTE/SERVIDOR

Todavía no podemos crear los canales para intercambiar datos a través de la red. Aún nos queda aprender el modelo o esquema de deben seguir TODOS nuestros programas para que se puedan comunicar.

La razón de que existan procesos en máquinas de la red que esperan que alguien contacte y dialogue con ellos es que están ahí por la simple razón de que dan UN SERVICIO. ¿En qué consiste ese servicio? Pues imaginaos todo lo que vosotros hacéis con vuestros dispositivos con Internet: consulta de páginas web, flujo continuado (streaming) de audio/video, sincronización de relojes, intercambio de archivos, correo electrónico... Es curioso que, durante el diseño de Internet (que no deja de ser tecnología de los años 70 y 80 con multitud de parches), sus arquitectos no pudieron imaginar la cantidad de servicios que iba a ser capaz de suministrar. Pues bien, todos, absolutamente todos esos servicios se suministran a través de unos procesos que se llaman SERVIDORES DE INTERNET.

El servidor es un proceso que siempre está esperando peticiones de otros. Tan simple como eso. Aquellos procesos que solicitan un servicio a un servidor y esperan su respuesta (bien en forma de una página web, un correo electrónico, un fichero, un flujo de audio/video...), se llaman CLIENTES.

Se deben cumplir tres reglas siempre:

1. Un servidor siempre se ejecuta antes de un cliente, porque si el cliente se ejecuta primero y no encuentra al servidor, no le puede enviar datos.
2. Un cliente tiene que conocer DÓNDE se encuentra el servidor, para poder mandarle peticiones de servicio. Aquí tenemos nuestro primer problema, porque el proceso está en otra máquina.
3. Un cliente y un servidor deben hablar en el mismo idioma. O lo que es lo mismo, deben utilizar el mismo protocolo de comunicación. Así, no será posible la correcta comunicación entre un cliente de correo y un servidor de páginas web, por ejemplo. Esto es fácil de entender.

¿CÓMO SABEMOS DÓNDE SE ENCUENTRA UN PROCESO SERVIDOR?

En Internet, hay dos identificadores únicos, que se utilizan para llegar hasta el socket de un proceso y así depositar datos en su cola de recepción:

1) La dirección IP de la máquina donde se ejecuta el proceso. Estos identificadores son valores numéricos de tipo entero de cuatro bytes. Habitualmente, utilizamos una notación en la que cada byte se expresa en decimal, separado por puntos (así es más manipulable por una persona).

   Ejemplos de direcciones IP:

   192.168.0.1

   10.20.30.254

   127.0.0.1

Date cuenta de que en un byte (en este caso se emplean sin signo) solo caben valores entre 0 y 255. Una vez vi en un libro un ejemplo de dirección IP parecida a ésta: 100.200.300.1. ¡Qué barbaridad, ese 300! Aún me duelen los ojos.

Ya que estamos, os diré que la dirección 127.0.0.1 es muy importante, y se utiliza para indicar la dirección LOCAL de tu equipo[2]. Identifica a tu propio ordenador, y suele ser la que utilizaremos en las prácticas, pudiendo así ejecutar el servidor y el cliente en nuestra misma máquina para probar nuestras comunicaciones. Cuando se envían datos a la IP 127.0.0.1, dichos datos nunca llegan a la tarjeta de red y, por tanto, no hacen uso de ancho de banda de la red. Es el kernel el que deposita los datos en el socket de recepción adecuado.

2) El identificador del canal de recepción asociado a un proceso concreto se llama PUERTO. Es un valor numérico de tipo entero sin signo, de dos bytes. Es decir, puede tomar valores entre 0 y 65535.

   Esta es la forma que tiene un protocolo de transporte de datos de Internet de decir que ciertos datos van destinados a un proceso y no a otro, dentro de una máquina. Por tanto, un mismo puerto solo pueden utilizarse por un proceso a la vez.

   Cuando un proceso crea un canal bidireccional en el kernel y se prepara para ser servidor, antes de poder recibir datos de los clientes, necesita elegir un PUERTO y decirle al sistema operativo que asocie (vincule) su socket con ese valor. Si el puerto está libre, el sistema hará esa asociación mientras exista el canal.

Sabiendo A PRIORI estos dos identificadores, la IP y el PUERTO, un cliente escribirá datos por su socket, proporcionando además esta información concreta que identifica al destinatario, para que el kernel monte los paquetes (cabecera y datos) y los envíe a través de la red.

Un momento, ¿y si el servidor le quiere responder al cliente? Claro, una cosa es pedir y otra obtener el servicio. En ese caso, resulta que el proceso cliente también incluye la IP de su máquina y su puerto asociado junto a los datos que manda al servidor. Toda esta información llega al proceso servidor, que puede así montar la respuesta al destino adecuado.

¿Acabas de decir que un cliente también tiene un puerto asociado a su canal? Por supuesto. En el momento en que el cliente crea un canal, puede también vincularlo a un puerto (siempre que esté sin utilizar por otro socket). Muchas veces, esta operación es opcional, y puede delegar esta operación al sistema operativo, que elegirá uno que esté libre y se lo asignará cuando se envíe el primer mensaje al servidor (por tanto, el procedimiento de vinculación entre canal y puerto siempre existe, de una forma u otra).

Como el sentido del diálogo siempre es primero de cliente a servidor, está claro que el PUERTO del servidor tiene que ser conocido a la fuerza por el cliente antes de que se produzca la comunicación. Sin embargo, observa que esto no se cumple nunca en el sentido contrario: el servidor jamás conocerá a priori el puerto que identifica al cliente ¡puesto que el cliente existe

[2] Más técnicamente, 127.0.0.1 es la dirección IP de un interfaz de red "de pruebas en lazo". Muchos equipos de telecomunicaciones tienen un modo de pruebas que consiste en transmitir datos para recibirlos en el mismo equipo, sin hacer uso del canal. En ese modo de comunicación de pruebas en lazo, se debe cumplir que la recepción se produce sin errores. Así se pueden descartar fallos debidos al propio equipo, y achacárselos al canal.

(se crea) siempre después del servidor! Solo en el momento en el que el cliente contacta con el servidor, éste aprenderá cual es el puerto a donde debe encaminar su respuesta.

PREPARANDO DIRECCIONES DE SOCKETS

Vamos a ir empezando a preparar nuestra dirección de socket, para indicársela al kernel junto a los datos que queremos enviar. Ya sabemos que necesitamos indicar una dirección IP y un PUERTO, y hemos dicho que estos valores se pueden guardar en variables enteras sin signo, de cuatro bytes y dos bytes, respectivamente. Pero antes, tenemos que resolver dos gravísimos problemas que siquiera sabíamos que teníamos hasta ahora. ¡Maldita sea, con lo bien que íbamos!

LOS EXTRAORDINARIOS PROBLEMAS DE LA REPRESENTACIÓN DE DATOS ENTRE MÁQUINAS DISTINTAS

PRIMER PROBLEMA: ¡el tipo *int* es el demonio!

Ha llegado el momento de desechar para siempre el uso del tipo de dato int en nuestros programas. Adiós, querido amigo. Lo sentimos mucho, pero no eres un dato fiable para mandarte por la red ¿Por qué? Pues porque en el momento en el que estamos enviando un entero de una máquina a otra, puede ser que el tamaño en bytes de un entero en la máquina origen sea distinto al de la máquina destino, creando un problema monumental. Ejemplo:

Plataforma origen: envía un entero que, en su sistema operativo, ocupa 4 bytes (32 bits)

Plataforma destino: espera recibir un entero que, en su sistema operativo, ocupa 2 bytes (16 bits)

El destino recibe dos bytes y se queda conforme, ¡pero ese valor no está completo! Al revés también hay lío.

SOLUCIÓN:

Nunca vamos a enviar int por la red. A partir de ahora, vamos a utilizar nuevos tipos de datos enteros, definidos en la librería <stdint.h> (tipos de entero estándar).

#include <stdint.h>

Incluye estos tipos de datos:

int8_t, int16_t, int32_t, int64_t (que representan a un entero con signo de 8 bits, uno de 16 bits, uno de 32 bits y otro de 64 bits, respectivamente).

Y sus versiones sin signo:

uint8_t, uint16_t, uint32_t, uint64_t

De esta forma, os pongo un ejemplo de representación correcta de valores enteros:

uint16_t x = 60000; //este valor cabe en dos bytes

y cuando indiquemos el número de bytes a mandar al destino, pondremos sizeof(uint16_t) o, directamente, 2. En recepción, esperaremos a leer dos bytes, almacenándolos también en una variable de tipo uint16_t

¡Alto! Ahí hay otro error absolutamente terrorífico. No se puede mandar ni recibir ese entero de dos bytes así, tal cual. Eso nos lleva al siguiente problema...

SEGUNDO PROBLEMA: Gulliver, el graciosillo y ese asunto de los huevos

¿Os habéis preguntado alguna vez en qué orden en memoria se guardan los bytes de una variable? Nosotros ya sabemos en qué orden se guardan las variables en la pila, o en el montículo, pero ¿en qué posición concreta se guarda cada byte? Vamos a verlo con un ejemplo:

Voy a crear una variable entera de dos bytes con este contenido:

uint16_t num = 0xDEAD;

En esta ocasión, por cuestiones didácticas, he decidido guardar contenido en formato hexadecimal. Creo que todos estamos de acuerdo en que el byte más significativo contiene 0xDE, y el byte menos significativo contiene 0xAD;

Está claro que esta información está en la memoria del ordenador. Supongamos que la dirección de la pila asociada a la variable num es la 0x101010. Por lo tanto, en la posición 0x101010 estará uno de los bytes de la variable, y en la posición 0x101011 estará el otro ¿pero qué byte corresponde a qué posición?

La respuesta depende de la arquitectura del ordenador que ejecuta el proceso. Hay dos tipos de arquitecturas:

- Las que tienen el byte más significativo primero en la memoria, se denominan big endian. Los grandes servidores de Internet, históricamente, corrían en estas arquitecturas (Solaris, Motorola...).
- Las que tienen el byte menos significativo primero en la memoria se denominan little endian. Los ordenadores personales con procesadores Intel son un ejemplo de arquitectura little endian, y también los procesadores ARM de los teléfonos móviles.

Estos nombres tan extraños provienen del libro de Jonathan Swift "Los viajes de Gulliver", donde los liliputienses pertenecían a dos grupos enfrentados, según el sitio por donde rompían los huevos cocidos (una facción los cascaba por el lado más grande o big-end y el lado por el lado más estrecho, o little-end). Y así fue como a un ingeniero rumboso se le ocurrió en 1980 denominar a las dos arquitecturas de computadores ¡manda huevos!

Pero, aplicando esto del big endian y little endian a la variable de nuestro ejemplo, obtenemos dos posibilidades, claro:

Arquitectura	Dirección de memoria	
	0x101010	0x101011
Big-endian	0xDE	0xAD
Little-endian	0xAD	0xDE

Por si no habíais caído, esto afecta, ¡y de qué manera! a la forma en la que interpretamos los bytes que se intercambian entre dos máquinas. Por ahora, sabemos que si al sistema operativo le decimos que envíe (o que escriba, porque es lo mismo) dos bytes a partir de la dirección de memoria &num, le estamos diciendo que envíe primero el byte de la posición &num, y después el siguiente, en ese orden.

El lío está servido si enviamos el contenido de la variable num desde una arquitectura big-endian a una little-endian pasa esto:

Big-endian (emisor): envía 0xDE, envía 0xAD, en este orden

Little endian (receptor): recibe 0xDE y luego 0xAD. Claro, estos bytes se colocan en orden en una variable en memoria: 0xDE en su primera posición y 0xAD en la segunda. ¿Qué ha pasado? Pues que se han invertido el orden de los bytes ya que, para esta arquitectura, el 0xDE es el byte menos significativo, y el 0xAD el más significativo. Es decir, si imprimo el valor de la variable recibida obtengo el valor 0xADDE.

Vaya desastre: la inversión de orden provoca que no se interprete bien la información. Imagina que estás leyendo el valor de temperatura de una central nuclear, tu extracto bancario, la hora a la que empieza un examen del campus... Una auténtica pesadilla.

Por supuesto este problema no existe si el emisor y el receptor se ejecutan en la misma arquitectura, pero ¿quién sabe eso de antemano? Ya os lo digo yo. Nadie. Ni Gulliver.

SOLUCIÓN

Os voy a dar la solución clásica, porque es la que utilizan la mayoría de los protocolos con los que funcionan nuestras aplicaciones de Internet favoritas. Eso sí, no es la única.

Lo que vamos a hacer es que todas las variables enteras que tengan más de un byte las convertiremos a formato big endian antes de transmitirlas. Este es el "formato elegido para la red", porque en los albores de Internet se pensaba que las arquitecturas iban a ser, mayoritariamente, big endian (y fijaos que ahora ese tipo está casi desaparecido, casi todo es ya little endian).

¿Esto quiere decir que yo, como programador, tengo que saber si mi programa se está ejecutando en little endian, y darles la vuelta a los datos antes de mandarlos?

O que, si voy a recibirlos sabiendo que vienen en formato big endian, ¿tengo que saber si les tengo que dar la vuelta si mi programa se está ejecutando en little endian?

Pues no, menos mal. El sistema operativo se encarga, porque él sí que sabe en qué tipo de arquitectura se está ejecutando. Así que nos propone que utilicemos estas funciones ANTES DE ENVIAR:

```
uint16_t   num = 60000;
uint16_t num_en_big_endian = htons(num); //este es el que se transmite por la red
uint32_t num2 = 70000; //este valor cabe en cuatro bytes
uint32_t num2_en_big_endian = htonl(num2); //este es el que se transmite por la red
```

Estas dos funciones tienen un nombre extraño, pero rápidamente se resolverá su misterio. Vamos a leerlas de otra forma:

htons() significa **host to net for short int**. Devuelve, en formato de red, el valor introducido como argumento, que está en formato de host. La palabra host se utiliza en la terminología de Internet para referirse a un ordenador de la red. Por lo tanto, lo que hace esta función es devolver en big endian el valor del argumento que se le pasa. Este argumento es un entero corto que, históricamente, equivale a enteros de 16 bits (int16_t e uint16_t, en nuestro caso).

htonl() significa **host to net for long int**. Devuelve, en formato de red, el valor introducido como argumento, que está en formato de host. En este caso, el argumento es un entero largo que, históricamente, equivale a enteros de 32 bits (int32_t e uint32_t, en nuestro caso).

Y, por supuesto, el sistema operativo nos ofrece también las funciones para realizar las funciones de conversión inversas, que hay que utilizar DESPUÉS DE RECIBIR:

uint16_t num_en_big_endian; //supon que almacena la informacion recibida por la red
uint16_t num = **ntohs**(num_en_big_endian); //este es su valor correcto
uint32_t num2_en_big_endian; //supon que almacena la informacion recibida por la red
uint32_t num2 = **ntohl**(num2_en_big_endian); //este es su valor correcto

Traducimos de igual forma:

ntohs() significa **net to host for short int**. Devuelve, en el formato propio del host donde se ejecuta el proceso, el valor que se le pasa en formato de red. El argumento es un entero "corto" de 16 bits (int16_t e uint16_t, en nuestro caso).

ntohl() significa **net to host for long int**. Devuelve, en el formato propio del host donde se ejecuta el proceso, el valor que se le pasa en formato de red. El argumento es un entero "largo" de 32 bits (int32_t e uint32_t, en nuestro caso).

Y después de resolver estos dos extraordinarios problemas, por fin podemos preparar la dirección completa a la que mandar nuestros datos. Vamos a ello.

PREPARANDO DIRECCIONES DE SOCKETS (ESTA VEZ SÍ QUE SÍ)

ADVERTENCIA: no vas a tener que aprender nada de memoria de lo que veas a partir de esta sección. Asegúrate que entiendes, eso sí, por qué se hace así. Cuando implementamos programas que usan sockets, todos cortamos y pegamos, sabiendo lo que estamos haciendo.

Una dirección de socket para Internet es de tipo **struct sockaddr_in** (se puede consultar en man 7 ip), y tiene este aspecto:

Solo se utilizan los tres primeros campos:

- sin_family: es obligatorio que este campo tenga el valor AF_INET, que es una constante que identifica que esta dirección pertenece a la familia de direcciones (address family - AF- en inglés) utilizadas para Internet
- sin_port: es un campo de dos bytes, para poner el puerto en formato de red (big endian)
- sin_addr: es un campo de cuatro bytes para poner una IP en formato de red (big endian)
- sin_zero: nos aseguraremos de que está a cero, y no lo usaremos

Este es un ejemplo de cómo se rellena una variable de este tipo:

```
1  struct sockaddr_in dir_socket;
2
3  memset(&dir_socket,0, sizeof(dir_socket));
4  dir_socket.sin_family = AF_INET;
5  dir_socket.sin_port = htons(80); //en big endian
6  uint8_t dir_IP[4] = {127, 0, 0, 1}; //en big endian
7  memcpy(&dir_socket.sin_addr, dir_IP, 4);
```

En la línea 3, nos aseguramos de que toda la estructura está a cero. Para ello, utilizamos la función de biblioteca memset() que, permite poner todos los bytes indicados, a partir de una dirección de memoria, en el valor que se pasa como segundo argumento.

La línea 4 rellena el campo de familia de direcciones a AF_INET. Siempre se pondrá ese valor. En la línea 5 nos encontramos con una función conocida: htons(), para indicar que el identificador del puerto de la dirección es 80 ¡pero en big endian!

La línea 6 utiliza un truco para especificar una dirección IP, es utilizando un array de cuatro casillas tipo uint8_t (es decir, cuatro bytes). Fíjate que, tal y como están rellenas las casillas, ya está en formato big endian, puesto que el byte más significativo de la dirección corresponde a la primera casilla de la memoria, y así sucesivamente.

Por último, en la línea 7, se hace una copia de memoria del array al campo sin_addr.

Existe una forma de obtener la dirección en formato big endian, pero en un uint32_t, usando la función inet_addr() (man 3 inet_addr).

```
uint32_t ip = inet_addr("127.0.0.1");
```

Esta función convierte una cadena de caracteres con una dirección IP expresada en notación decimal con puntos a un uint32_t en big endian. En caso de que la IP esté mal escrita, devuelve -1.

El uso de inet_addr() es muy conveniente cuando la IP se pasa como argumento a un programa, por ejemplo, así que el equivalente a las líneas 6 y 7 anteriores podría ser:

```
6  uint32_t dir_IP = inet_addr("127.0.0.1");  //la devuelve en big endian
7  memcpy(&dir_socket.sin_addr, &dir_IP, 4);
```

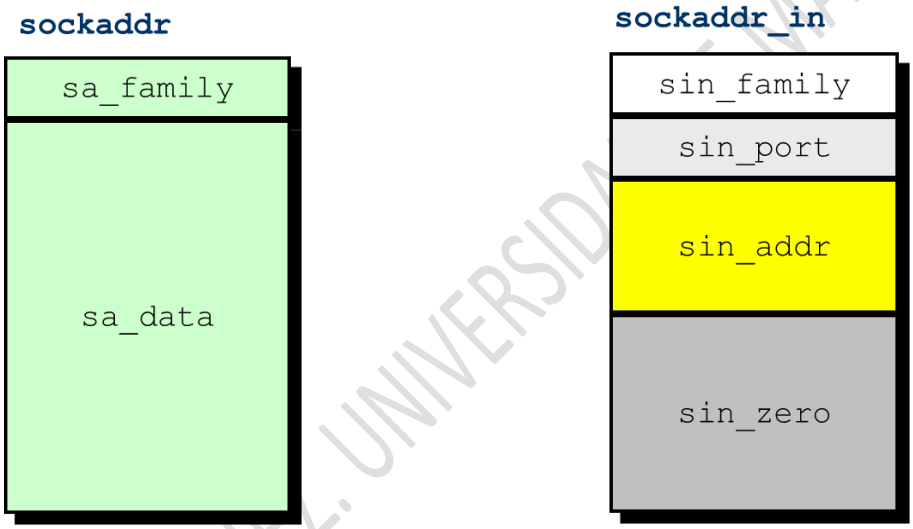
Seguro que te extraña que no haya intentado hacer esto:

```
dir_socket.sin_addr = inet_addr("127.0.0.1"); //ERROR, tipos incompatibles
```

¡Da error de compilación! La razón es muy sencilla, si miras en el manual (man 7 ip), verás que este campo no es de tipo uint32_t, sino una estructura con un único campo, esta vez sí, de tipo uint32_t. Así que esta es la forma correcta de hacerlo:

dir_socket.sin_addr.s_addr = inet_addr("127.0.0.1"); //OK

Y con todo esto, ya tenemos nuestra dirección de sockets rellena y lista para utilizar en las llamadas al sistema que la necesiten. Sin embargo, has de saber que estos canales bidireccionales, que llamamos sockets, fueron concebidos para poder utilizarse con distintas tecnologías de red, no solo para Internet. Por eso, de forma genérica, todas sus funciones utilizan las denominadas "direcciones genéricas", normalmente un **puntero** a una estructura de tipo **struct sockaddr**, que tiene este aspecto (a la izquierda, en la figura):



A la derecha, tienes la struct sockaddr_in de Internet. Si te fijas, ocupan el mismo tamaño en memoria, así que un casting de este tipo es seguro, y además necesario:

//La direccion genérica que necesitan las funciones de sockets se obtiene con un casting:
struct sockaddr * dir_generica = (struct sockaddr *)&dir_socket;

No te preocupes si todavía no tienes claro dónde utilizar esta dirección que hemos montado. Lo mejor será pasar ya a crear nuestro primer socket y ver estas líneas de código en acción.

CREACIÓN DE SOCKETS PARA COMUNICACIONES NO FIABLES

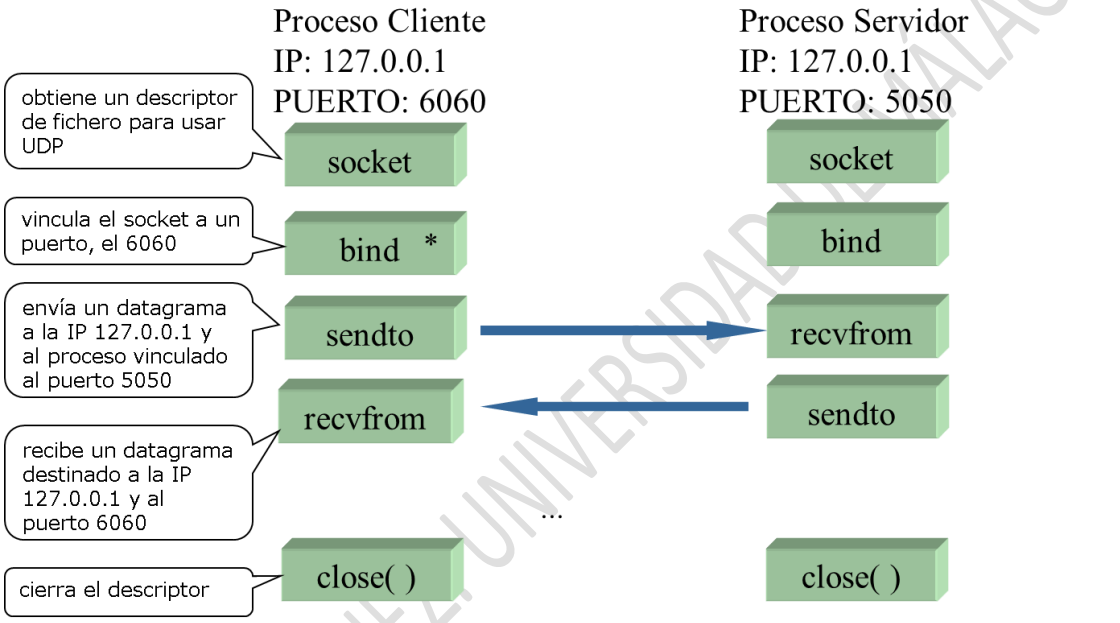
Internet permite que entre dos procesos se establezca un diálogo no fiable, lo que implica que los mensajes que se intercambian se pueden perder en la red, o pueden llegar desordenados o incluso duplicados[3]. ¿Y quién en su sano juicio implementaría servicios NO fiables? Pues hay bastantes, ya que garantizar que los mensajes no se pierden, ni llegan desordenados, ni

[3] Un mensaje puede llegar dos veces (duplicado) si se mandó la primera vez y luego se decidió retransmitirlo pensando que se había perdido. Sin embargo, si no estaba perdido pero terminó llegando con retraso, el receptor se encuentra en su cola con dos mensajes idénticos.

duplicados, implica complicar el protocolo con el que se comunican los procesos. Complicar es sinónimo de añadir chequeos extra para que todo vaya bien, y eso termina haciendo que los datos lleguen más lentos a su destino.

Así que, cuando la velocidad es un problema, se suele recurrir a los sockets no fiables. Los servicios de streaming suelen utilizar este tipo de comunicación. En Internet, el protocolo encargado de realizar el intercambio de mensajes entre procesos de manera no fiable se llama UDP (user datagram protocol).

A partir de aquí, vamos a crear un servicio de eco utilizando sockets UDP. El cliente envía un mensaje al servidor y éste le responde con el mismo mensaje. Para ello, os presento en esta figura la estructura de los dos procesos, donde cada caja corresponde a la llamada al sistema de sockets que se va a usar, en secuencia.



(*): En un cliente, bind() es opcional. Si no se usa, el sistema operativo asigna al socket un puerto efímero

Fijémonos en la parte cliente. La secuencia de llamadas al sistema para sockets es la siguiente:

socket(), bind() (que es opcional), sendto(), recvfrom(), close()

Y en el servidor:

socket(), bind() (aquí es obligatorio), recvfrom(), sendto(), close()

CREACIÓN DE UN SOCKET

Comenzamos, por fin, la creación del programa del cliente. Para ello, reservamos un canal bidireccional de este tipo:

int sd = socket(PF_INET, SOCK_DGRAM, 0);

El sistema operativo creará un canal bidireccional y preparará los mecanismos para que el envío y la recepción de mensajes se haga a través de la familia de protocolos de Internet (PF_INET), utilizando un servicio de datagramas (intercambio no fiable, SOCK_DGRAM), y el protocolo por

defecto que realice esta tarea (el valor cero del tercer parámetro indica que se seleccionará UDP).

NOTA: En muchos códigos verás que se usan de forma equivalente AF_INET o PF_INET. Hace mucho tiempo se pensó que una familia de protocolos (PF), conceptualmente, no tenía por qué ser lo mismo que una familia de direcciones (AF). Pero esto no terminó cuajando, y la realidad es que las dos se pueden utilizar sin problema.

Esta llamada al sistema socket() (man 2 socket), devuelve -1 si falla, o un descriptor de fichero que identifica al canal bidireccional creado por el kernel (el nombre de la variable, sd, la he elegido como abreviatura de socket descriptor).

ASOCIACIÓN DE UN SOCKET CON UN PUERTO CONCRETO

Sabemos que un canal de socket se vincula siempre a un puerto. Recuerda que, junto al mensaje del cliente, va a viajar información extra que ayude a los protocolos de Internet a entregar correctamente la información en la cola de recepción del destinatario.

El puerto donde el cliente va a poder recibir mensajes (vamos a denominarlo origen) es parte de la información extra que se envía. En un cliente, podemos hacer esta asociación a un puerto concreto elegido utilizando bind() (man 2 bind). Pero recuerda que, si no se hace, el sistema operativo hará ese vínculo con alguno que él elija y que esté libre en ese momento (no asociado a otro socket).

Así que estas líneas de abajo, en un cliente, son opcionales. Aun así, las explicamos ahora para que también sirvan para cuando se necesiten cuando el servidor use bind().

```
1  struct sockaddr_in vinculo;
2  memset(&vinculo, 0, sizeof(vinculo));
3  vinculo.sin_family      = AF_INET;
4  vinculo.sin_port        = htons(6060); //puerto en big endian
5  vinculo.sin_addr.s_addr = INADDR_ANY;
6
7  int resultado = bind(sd, (struct sockaddr *)&vinculo, sizeof(vinculo));
8  if(resultado < 0){
9    perror("Error en bind");
10   exit(1);
11 }
```

Las líneas 1 a 4 no suponen ninguna variación sobre lo conocido. Se crea una variable de tipo struct sockaddr_in y se va rellenando. Sin embargo, hay algo nuevo aquí en la línea 5. Fíjate que en esta ocasión SOLO estoy rellenando esta variable de dirección para indicarle al sistema operativo el puerto que me interesa vincular al socket, así que el único campo realmente interesante que va a tener en cuenta es el sin_port. ¿Y qué se pone en el campo destinado a las direcciones IP? Aunque no nos interese su valor, en el campo sin_addr se debe introducir la constante INADDR_ANY para que se pueda interpretar correctamente por parte del sistema[4]). Otra forma de rellenar este campo es:

[4] Técnicamente, lo que indica la constante INADDR_ANY es que el sistema operativo debe guardar mensajes destinados al puerto asociado y provenientes desde cualquier tarjeta de red con IP válida en el ordenador, en la cola de recepción del socket al que se vincule

```
uint32_t any_ip = INADDR_ANY;
memcpy(&vinculo.sin_addr,&any_ip, 4);
```

Si el puerto solicitado está libre, la asociación se hará correctamente introduciendo la variable vínculo en la llamada al sistema bind(), como se hace en la línea 7. El funcionamiento de esta llamada es sencillo: el primer argumento es el socket sobre el que se quiere hacer el vínculo con el puerto; el segundo argumento pasa la dirección a una estructura donde se haya rellenado el puerto, aunque respetando el casting necesario para convertirlo a **struct sockaddr ***; el tercer argumento indica el tamaño en bytes de la estructura introducida anteriormente.

ELIGIENDO PUERTOS PARA LA VINCULACIÓN

Lo normal es que la especificación de tu problema o norma te diga el puerto que tienes que utilizar para vincularlo al socket. Sin embargo, si te dan libertad para elegir uno, sobre todo cuando estás practicando tú solo, sigue esta recomendación.

No utilices los puertos del 0 a 1023, porque están reservados para procesos ejecutados por el superusuario de Unix/Linux, y bind() puede fallar si lo ejecuta cualquier otro usuario del sistema. Por lo demás, cualquier puerto a partir del 1024 se puede utilizar, siempre que no esté ocupado. Para saber si un puerto concreto está en uso en tu máquina, puedes ejecutar en consola este comando:

ss -tunl

Fíjate que hay una columna que se llama Local Address:Port. Lo que aparezca ahí son puertos que están siendo usados, esos no los vas a poder vincular (si lo intentas, bind fallará).

ENVÍO DE UN MENSAJE POR UDP

Esta parte es mucho más intuitiva: preparamos una nueva variable de dirección de socket de tipo struct sockaddr_in con el puerto y la IP del servidor a donde queremos enviar el mensaje. Después, utilizamos la función sendto() (man 2 sendto) para enviar los bytes que queramos a esa IP y puerto indicados.

```
 1  struct sockaddr_in dir_servidor;
 2  memset(&dir_servidor, 0, sizeof(dir_servidor));
 3  dir_servidor.sin_family = AF_INET;
 4  dir_servidor.sin_port = htons(5050);
 5  uint32_t ip_servidor = inet_addr("127.0.0.1");
 6  memcpy(&dir_servidor.sin_addr, &ip_servidor, 4);
 7
 8  char mensaje[] = "Hola eco!"
 9  int enviados = sendto(sd, mensaje, strlen(mensaje), 0,
10                                  (struct sockaddr *) &dir_servidor, sizeof(dir_servidor));
11  if(enviados < 0){
12    perror("Error sendto");
13    exit(1);
14  }
```

El relleno de las líneas 1 a 6 ya debe resultarnos familiar. En la línea 9 utilizamos la llamada a sendto() para colocar un mensaje en la cola de transmisión socket, y ya el kernel se encargará cuando pueda de enviarlo usando la tarjeta de red.

Los tres primeros argumentos de sendto() son idénticos a los de cualquier llamada a write(): el descriptor de socket, la dirección donde empiezan los bytes a enviar y el número de los mismos que forman el mensaje.

Los siguientes argumentos son nuevos:

- Cuarto argumento: opciones de envío (nosotros en esta asignatura no utilizamos ninguna opción, se deja a cero)
- Quinto argumento: dirección de memoria (en formato struct sockaddr *) de la estructura donde se han rellenado la IP y puerto destino de este mensaje.
- Sexto argumento: tamaño en bytes de la estructura introducida como quinto argumento.

Y eso es todo. Si la cola de mensajes asociada a este socket estuviera llena, sendto() bloquea al proceso hasta que el kernel deje sitio al enviar mensajes pendientes. Si hubo algún problema (algún parámetro no tiene la información correcta), la llamada devuelve -1. Y, en el caso normal, devuelve SIEMPRE el mismo número de bytes que se pretendieron enviar.

RECEPCIÓN DE UN MENSAJE POR UDP

Te habrás dado cuenta de que en estos canales de tipo UDP estoy utilizando siempre el término mensaje. Y lo digo totalmente a posta, porque las colas de transmisión y de recepción de los sockets UDP NO son tuberías de bytes, como vimos en el tema anterior. Eso quiere decir que en cada casilla de las dos colas del kernel, para cada sentido, se guarda un mensaje completo, y separado del resto.

Por lo tanto, cuando vayas a leer un mensaje de la cola del socket UDP, no tienes que saber a priori cuantos bytes necesitas leer ¡no hace falta nada parecido a readn()! Si hay un mensaje, se lee íntegro, con el tamaño correcto con el que se envió desde el otro extremo. Para esto, utilizamos la llamada al sistema recvfrom() (man 2 recvfrom):

```
1   char datos[512];
2   struct sockaddr_in dir_origen;
3   socklen_t longitud_direccion = sizeof(dir_origen);
4
5   int recibidos = recvfrom(sd, datos, 512, 0, (struct sockaddr *) &dir_origen,
6                                                                  &longitud_direccion);
7
8   if(recibidos < 0){
9     perror("Error recvfrom");
10    exit(1);
11  }
```

Esta llamada al sistema funciona de forma similar a un read(): bloquea al proceso si la cola de recepción del socket sd está vacía. El primer, segundo y tercer argumento tienen el mismo significado que en read(). El resto de argumentos se utilizan de esta manera:

Cuarto argumento: opciones de recepción (no las utilizamos, las dejamos siempre a cero)

Quinto argumento: dirección de una variable de dirección de socket (en formato genérico) donde se devuelve la información de la IP y el puerto del proceso que nos envió este mensaje. Este argumento puede ser interesante: supón que quién responde NO es el mismo proceso al que tú mandaste tu información (no coincide la IP y el puerto) ¿Esto podría pasar? Por supuesto, nadie te asegura que estás hablando siempre con el mismo interlocutor (esto ya se utiliza, desde tiempos inmemoriales, en primer curso de la carrera de hackers).

Sexto argumento: dirección de la longitud de la variable de dirección de socket. Aquí hay un matiz que suele provocar errores. Es un parámetro por referencia, pero necesita pasarse a la función inicializada correctamente con el número de bytes que ocupa la estructura sockaddr_in. En la línea 3 viene esta inicialización. El tipo de este argumento es socklen_t (es una manera de indicar una longitud en bytes, como size_t).

SIGO RECORDANDO QUE NADIE SE ESTUDIA ESTAS LÍNEAS DE MEMORIA. CUALQUIER EJERCICIO ASUME QUE VAMOS A HACER COPIADO Y PEGADO CON CRITERIO.

CIERRE Y LIBERACIÓN DE UN SOCKET

Hemos llegado al final del cliente de eco. Basta con sacar por pantalla el mensaje recibido y estaremos listos para salir del programa:

write(1, datos, recibidos);

Para indicar al kernel que libere los recursos y colas asociadas al socket, simplemente se utiliza la mítica llamada al sistema close():

close(sd);

CONSIDERACIONES FINALES

Sigue todos los pasos anteriores para construirte el servidor de eco asociado a este cliente, facilmente. La única diferencia notable es que el servidor realiza la asociación de su socket con el puerto 5050 (en vez de con el 6060), y que lo primero que hace es bloquearse en recvfrom() hasta recibir un mensaje de un cliente. Después, utiliza la dirección de socket que devuelve recvfrom() por referencia, para enviar el mensaje recibido de vuelta al mismo cliente, con sendto().

Otra cosa. Te has dado cuenta de que los arrays de bytes no están afectados por la polémica big endian/little endian, ¿verdad? Efectivamente, cada casilla, que es un byte, llega perfectamente ordenada al destino y se coloca en posiciones consecutivas de memoria de igual forma que estaba colocada en la memoria del emisor, sin que haya ningún posible error de interpretación.

Tampoco pasa nada si utilizo un socket para enviar un entero que ocupa un byte (int8_t o uint8_t). ¡No se malinterpreta porque es imposible darle la vuelta a un byte!

En el caso de que quiera enviar un entero que ocupe más de un byte, ahí sí hay que hilar fino. Este es un ejemplo para usar sendto() y recvfrom() con un valor que cabe en un int16_t:

```
1  int16_t valor = 500;
2  int16_t valor_en_big_endian = htons(valor);
```

```
3  int enviados = sendto(sd, &valor_en_big_endian, 2, 0,
4                                  (struct sockaddr *) &dir_servidor, sizeof(dir_servidor));
5  if(enviados < 0){
6    perror("Error sendto");
7    exit(1);
8  }
9  int16_t valor_rv_en_big_endian;
10 struct sockaddr_in dir_origen;
11 socklen_t longitud_direccion = sizeof(dir_origen);
12
13 int recibidos = recvfrom(sd, &valor_rv_en_big_endian, 2, 0,
14                                   (struct sockaddr *) &dir_origen, &longitud_direccion);
15 if(recibidos < 0){
16   perror("Error recvfrom");
17   exit(1);
18 }
19 int 16_t valor_correcto = ntohs(valor_rv_en_big_endian);
20 printf("El valor recibido es %d\n", valor_correcto);
21
```