# Anotaciones a los códigos usados durante la presentación

## addrInfo.c

En este programa se usa la función getaddrinfo() para retornar posibles combinaciones para acceder a un par (host, puerto). Se pueden pedir todas las posibles o aplicar algún filtro.

Recibe un nombre o IP de host y opcionalmente un servicio (nombre conocido o número de puerto)

Para comprender su uso primero se debe explicar la estructura

struct addrinfo {

int ai\_flags;

int ai\_family;

int ai\_socktype;

int ai\_protocol;

socklen\_t ai\_addrlen;

struct sockaddr \*ai\_addr;

char \*ai\_canonname;

struct addrinfo \*ai\_next;

};

* ai\_flags: vector de bits para seteo de distintos flags, por ejemplo el bit AI\_NUMERICSERV indica que se solicitará información de un servicio en base al número de puerto, el bit AI\_PASSIVE indica que queremos que getaddrinfo() nos informe no números de IP específico sino un “wildcard” que represente cualquier dirección posible. Para mayor detalle ver <https://linux.die.net/man/3/getaddrinfo>
* ai\_family: AF\_INET, AF\_INET6, AF\_UNSPEC.
* ai\_socktype: SOCK\_STREAM, SOCK\_DGRAM
* ai\_protocol: protocolo de transporte, puede ser 0, IPPROTO\_XXX (donde XXX puede ser UDP, TCP, SCTP. PGM, etc.)
* ai\_addrlen: sizeof de lo apuntado por ai\_addr
* ai\_canonname: nombre canónico
* ai\_next: puntero al siguiente

Línea 29: establecemos como único filtro el flag AI\_CANONNAME, para que el campo ai\_canonname del primer elemento de la lista retornada apunte al nombre oficial del host.

Línea 30: invocamos a getaddrinfo() y le pasamos:

* addrString: nombre o número de IP en formato de puntos. En caso de querer obtener los IPs del host local este valor puede ser NULL. Si es para obtener los pares IP-Puerto de otro host no puede ser NULL
* service: nombre o número de puerto (en caracteres) al cual queremos acceder
* hint: filtros (puntero a la estructura addrinfo explicada más arriba)
* aiList: en este parámetro de salida dejará el puntero al primer elemento de la lista

Línea 31: En caso de error addrinfo retorna un valor distinto de cero con el código de error, la función gai\_strerror() retorna un string con un mensaje relativo al código de error retornado.

Líneas 32 a 39: recorre la lista e imprime todos los datos asociados a cada entrada.

Línea 41: libera la lista

## Función tcpClientSocket en tcpClientUtil.c

Esta función recibe por parámetro la identificación de un host (puede ser el FQDN o un string con la IPv4 o IPv6 en formato “legible para el humano”) y un string con el nombre del servicio o el número de puerto en el cual escucha el servicio.

Líneas 14 a 18: Prepara el *hint* a usar con addrinfo. En este caso vamos a solicitar únicamente TCP ( SOCK\_STREAM e IPPROTO\_TCP), y a una IP que puede ser v4 o v6 ( AF\_UNSPEC)

Línea 22: se invoca a la función getaddrinfo para que nos devuelva todas las posibles IPs que “matcheen” con lo establecido en las líneas 14 a 18

Líneas 29 a 43: se recorren todas las direcciones recibidas, intentando establecer la conexión con alguna de ellas.

Línea 31: crear el socket TCP.

socket(): crea un socket. En este caso puede ser para IPv4 o IPv6 usando un protocolo de transporte basado en flujo (SOCK\_STREAM) llamado TCP (IPPROTO\_TCP). Retorna un entero (el “file descriptor”). En caso de error retorna -1, se deja asentado en el log y se sigue probando. ¿Por qué se sigue probando?

Línea 35: intentar la conexión. En este caso si falla hay que liberar el socket creado. Como addrinfo nos puede retornar varias direcciones IP asociadas al mismo host, pero el host estar escuchando sólo en alguna/s de esas direcciones IPs, al fallar la conexión seguimos probando con las otras IPs

Línea 37: usamos la función close() para liberar los recursos reservados para un socket que no se va a usar

Línea 45: liberamos la lista de direcciones creada por addrinfo

## tcpECHOClient.c

Este programa implementa un cliente TCP que se conecta a un servidor ECHO ( <https://tools.ietf.org/rfc/rfc862.txt> ). El protocolo es muy simple: el servidor lee una serie de octetos y responde al cliente con esos mismos bytes.

Líneas 27: crear el socket y establecer conexión TCP con el servicio.

Línea 35: enviar el texto al servidor

* send(): recibe el socket, la dirección del comienzo del buffer y la cantidad de bytes a transferir. Retorna la cantidad de bytes que pudo enviar, o -1 si hubo un error. El último parámetro es *flags*, que en este caso no usamos ninguno, por lo que es equivalente a la función write(). Cada bit de flags representa una opción, por lo que se pueden setear más de una “encendiendo” el bit usando un OR a nivel de bits, algunos son:
  + MSG\_DONTWAIT, para que no bloquee. Si la operación debería bloquear (la capa de transporte no está disponible para procesar toda la información) retorna EAGAIN o EWOULDBLOCK
  + MSG\_MORE: habrá más datos para enviar (también soportada por UDP desde Linux 2.6). Le pide al kernel que junte en un paquete esta y las siguientes escrituras con este flag.

Para más detalles sobre los flags y valores de respuesta ver la sección 16.5 del libro de Stevens o <https://linux.die.net/man/2/send>

Líneas 40 a 56: recibir la respuesta del servidor. TCP es un protocolo “*byte-stream*” (flujo de bytes o caracteres): los bytes enviados en una llamada a send() no necesariamente serán leídos en una única invocación a recv() (la contrapartida de send() ). **Nunca asumir que hay una correspondencia uno a uno entre el send() de origen y el recv() en destino** .

* recv(): es una función bloqueante, retorna la cantidad de octetos leídos y copiados al buffer, o -1 si hubo algún error. Si retorna cero significa que la aplicación en la otra punta cerró la conexión TCP. Al igual que send() recibe como parámetro un vector de bits como flags, entre ellos:
  + MSG\_DONTWAIT: habilita operación no bloqueante
  + MSG\_WAIT: esperar hasta que toda la información esté disponible

Para más detalles sobre los flags ver la sección 16.5 del libro de Stevens o https://linux.die.net/man/2/recv

Línea 58 y 59: terminar la conexión y finalizar la ejecución.

* close(): le informa al socket destino que la comunicación ha finalizado, y luego libera los recursos locales que se hayan reservado (similar a fclose).

## tcpEchoServer.c

Un servidor TCP crea un socket pasivo, que es utilizado para escuchar pedidos de conexión. Una vez que se recibe un pedido de conexión, se crea un socket “activo” para interactuar con el cliente. Notar que para TCP el concepto de cliente y servidor es sólo a los fines de establecer la conexión, una vez establecida cualquiera de los dos puede leer y/o escribir como así también cerrar la conexión; es la aplicación la que hace la distinción entre ambos.

Líneas 14 a 16: setup y análisis de parámetros. Sólo se recibe el puerto en el cual escuchar. Otros parámetros que se pueden agregar podría ser si se desea escuchar en alguna IP en particular, por ejemplo en la 127.0.0.1 para que sólo se pueda usar desde un cliente en el mismo host. O que use preferentemente IPv6

Línea 20: crear el socket pasivo. (ver detalles más adelante)

Línea 26: quedarse bloqueado hasta que se establezca una conexión con el cliente. La función retorna el id del socket activo.

Línea 30: la función handleTCPEchoClient se encarga de interactuar con el cliente y cerrar el socket activo cuando el cliente cierre la conexión. El servidor no podrá atender a nuevos clientes hasta que se termine la sesión con el actual, por eso es un servidor iterativo. Una vez finalizada la sesión vuelve a ejecutarse la línea 26, atendiendo al primer cliente que se haya quedado esperando o se quedará esperando a la llegada de un nuevo cliente.

## Función setupTCPServerSocket en tcpServerUtil.c

Esta función se encarga de crear el socket pasivo en el puerto recibido por parámetro

Líneas 21 a 26: seteamos el *hint* a usar con getaddrinfo buscando en qué IP crear el socket pasivo. No se especifica una dirección en particular

* Línea 23: podrá ser IPv4 o IPv6 (o alguna futura variante)
* Línea 24: para que en la lista incluya “socket address” en las que se pueda “bindear” un socket que pueda aceptar ( o “acceptar” ) conexiones
* Líneas 25 y 26: sólo TCP

Líneas 42 a 46: creación del socket pasivo. De acuerdo al filtro de la línea 24, será un socket pasivo.

Líneas 49 a 61:

* bind(): relaciona el socket pasivo con IP y puerto local. Puede fallar si otra aplicación está escuchando en el puerto o no se tiene permiso para ese puerto. Al no haber especificado una dirección, se aceptarán los pedidos que arriben a cualquiera de las direcciones IP (pero sólo v4 o v6, no en ambas)
* listen(): le indica a TCP que acepte conexiones entrantes. Antes de llamar a esta función, cualquier pedido a ese puerto TCP será ignorado. Es una función no bloqueante y además del socket recibe como parámetro la cantidad máxima de pedidos de conexión a encolar.
* Si bind o listen fallan debemos cerrar el socket pasivo para liberar los recursos asociados, y también quede disponible es file id

Línea 64: liberamos la lista creada por getaddrinfo

## Función acceptTCPconnection en tcpServerUtil.c

Recibe el socket pasivo y espera por pedidos de conexión

* Líneas 75 a 79
  + accept(): función bloqueante (se puede hacer no bloqueante como analizaremos más adelante). Espera hasta que se establece una nueva conexión en el socket pasivo. Retorna un descriptor para un nuevo socket (socket activo) que ya está conectado al socket remoto (se hizo el “*three way handshake*”), el segundo argumento es un puntero a la estructura conteniendo la información (IP origen, puerto origen) y el tercero el *sizeof* de la estructura. Este socket activo ya está conectado y listo para recibir y enviar datos. La única finalidad de clntAddr es para logs o control, no es necesario saber los datos del cliente para poder enviar o recibir datos (a diferencia de UDP), basta con clntSock
* Líneas 82 y 83: se implementa un “*caller id*”.

## Función handleTCPEchoClient en tcpServerUtil.c

Recibe datos de un socket activo y escribe esos mismos datos en el socket como respuesta al origen, mientras recv() retorne un valor positivo. Cuando el cliente cierra la conexión en forma normal, recv() retorna cero y el servidor cierra el socket activo.

## UDPEchoClient.c

A diferencia de TCP, al usar UDP como protocolo de transporte una vez creado el socket el cliente envía un datagrama (un único mensaje) y queda a la espera de recibir una respuesta.

En esta implementación dejamos como ejercicio realizar algo similar a tcpEchoClient: si no podemos enviar el datagrama a una de las IPs del host, intentar con otras.

Líneas 1 a 23: análogo al cliente TCP. A diferencia de TCP, se debe limitar la longitud del mensaje a enviar

Líneas 17 a 28: para el servidor se acepta una dirección IPv4, IPv6 o nombre a resolver. Para el puerto se acepta número o nombre de servicio. Se usa getaddrinfo() para determinar familia, dirección y número de puerto. Se acepta cualquier familia (AF\_UNSPEC) para UDP (SOCK\_DGRAM e IPPROTO\_UDP), por lo que sólo se aceptará IPv4 o IPv6.

Línea 31: creación y seteo del socket. Similar al cliente TCP sólo que se usa UDP y no es necesario connect().

Líneas 57 a 62: enviar un datagrama.

* sendto(): a través de un socket cliente UDP se envía un único datagrama a un IP y puerto. Notar que se puede utilizar el mismo socket para enviar otros datagramas a otros IP y/o puertos. La primera invocación a esta función hace que se elija una IP y puerto de origen, que si bien no nos interesa como aplicación los necesita el servidor para enviar la respuesta y la capa de transporte para pasarnos la/s respuesta/s (más adelante veremos otros usos, por ejemplo en firewalls). Nota: sendto(sock, buf, len, NULL, 0) es equivalente a send(sock,buf,len,0).

Líneas 69 a 75: establecemos un tiempo máximo para la espera de la respuesta. Con la función setsockopt() se pueden fijar diversas opciones al socket, más adelante veremos otras opciones para poder manejar sockets no bloqueantes, en este caso establecemos un timeout de 5 segundos para cualquier operación a llevar a cabo con el socket.

Líneas 77 a 89: recibir un mensaje.

* recvfrom(): función que bloquea hasta que un datagrama UDP arriba con destino al socket (IP y puerto destino coinciden con IP y puerto usados como origen). Entonces copia los datos al buffer y también el IP y puerto de origen, lo que permite verificar si la fuente dice que es (“dice que es” no es lo mismo que “es”) la misma a la que se envió el datagrama.
* Línea 85: al no haber una conexión, en un mismo socket se puede recibir datagramas de distintos orígenes. La función sockAddrsEqual() se encuentra en util.c.

## udpEchoServer.c

Es muy simple: dentro de un ciclo infinito espera recibir un datagrama y entonces enviarlo de vuelta para luego escuchar el siguiente. En esta implementación sólo aceptamos y retornamos hasta 64 bytes, fijados por la constante MAXSTRINGLENGTH.

Línea 21: similar a setupTCPServerSocket, solo que no es necesario ejecutar listen().

Se invoca a

int getaddrinfo(const char \*node, const char \*service,

const struct addrinfo \*hints, struct addrinfo \*\*res);

* node: con valor en NULL
* service: el puerto local, ya sea el número o nombre
* hints, con los siguientes valores
  + ai\_family: Se acepta cualquier familia de direcciones: AF\_UNSPEC
  + ai\_socktype y ai\_protocol: UDP como protocolo de transporte (SOCK\_DGRAM e IPPROTO\_UDP respectivamente)
  + ai\_flags: AI\_PASSIVE, que junto con node en null indica que escucharemos en cualquier dirección IP y usaremos bind() y accept(). La dirección retornada será un *wildcard* que referencie a todas las direcciones IP del host. Si node no es NULL, el flag AI\_PASSIVE es ignorado.

Líneas 42 a 62: creación del socket. Muy similar a TCP, excepto que crea un un socket para datagramas y no “*stream oriented*”.

Líneas 83 a 109: manejar mensajes entrantes en forma iterativa

* recvfrom(): bloquea hasta recibir un datagrama de un cliente. Como no hay conexión cada datagrama contiene la información necesaria para poder responder.
* sendTo(): toma la dirección y puerto obtenidos con recvfrom() y envía el mismo mensaje recibido.

## “Conectándose” con UDP

Es posible llamar a connect() con un socket UDP para fijar la dirección y puerto destino de futuros datagramas. En este caso no se debe usar sendto() sino send(). En forma similar usar recv() en vez de recvfrom().