

Taller de Programación I (75.42)

Clase 6: Sockets

Autor: Lautaro Mazzitelli

Revisión: 1. Lautaro Mazzitelli
2. Leandro H. Fernández

Alcance

El presente apunte de clase no pretende ser una guía exhaustiva para el aprendizaje de redes informáticas ni sockets. En cambio constituye una introducción básica al modelo de capas OSI y a los fundamentos de la programación sockets TCP sobre IP. No se tratará en el presente la programación con sockets UDP ni la modalidad de sockets no bloqueantes.

Introducción

Un *socket*, o *internet socket*, o *network socket* es una vía de comunicación o un canal entre dos computadoras que se comunican sobre una red.

La definición de socket a evolucionado para cumplir con especificaciones de diferentes estándares a lo largo de su historia. Para una definición más completa de qué es un socket se puede consultar la RFC 147 "*The Definition of a Socket*". Para el alcance del presente documento nos limitaremos a indicar que la definición dada en el párrafo anterior sirve desde un punto de vista global. Y agregaremos que visto desde un equipo en particular un socket es la representación lógica de un canal de comunicación con otro equipo.

Modelo OSI

En los inicios de las telecomunicaciones fueron desarrolladas distintas maneras de comunicar computadoras. A causa de esto surgió la necesidad de crear un estándar, para que, como todo estándar establezca una manera uniforme de implementar las cosas.

En el año 1984 la Organización Internacional para la Estandarización (ISO) estableció el Modelo de Referencia de Interconexión de Sistemas Abiertos (OSI, Open system Interconnection). Este modelo establece siete capas que determinan la arquitectura de comunicaciones que se recomienda seguir para la implementación de las comunicaciones. Dichas capas son: Física, Enlace, Red, Transporte, Sesión, Presentación, Aplicación. A continuación las describiremos brevemente.

Concepto de capas (layers)

Cada una de las capas se apoya en la inferior para hacer su trabajo. La capa inferior siempre le otorga servicios a la superior. Las dos terminales se comunican virtualmente capa a capa en este modelo de comunicaciones.

Capa Física

Es la encargada de la comunicación física con el medio (cables y conectores) y de la manera en que se codifica la información (niveles de tensión, modulación, etcétera). El medio físico se divide en medios guiados (cable coaxial, par trenzado, fibra óptica) y no guiados (radio, infrarrojo, microondas). En cuanto a la codificación existen varios tipos como manchester, manchester diferencial, AMI, y otras.

Capa Enlace

La capa de enlace de datos se ocupa del direccionamiento físico (dirección MAC), de la topología de la red, del acceso a la red, de la notificación de errores, de la distribución ordenada de tramas y del control del flujo.

Capa Red

El cometido de la capa de red es hacer que los datos lleguen desde el origen al destino, aún cuando ambos no estén conectados directamente. Es decir que se encarga de encontrar un camino manteniendo una tabla de ruteo y atravesando los equipos que sea necesario, para hacer llevar los datos al destino. Es la encargada también de gestionar la congestión en la red.

Capa Transporte

Su función básica es aceptar los datos enviados por las capas superiores, dividirlos en pequeñas partes si es necesario, y pasarlos a la capa de red. En el caso del modelo OSI, también se asegura que lleguen correctamente al otro lado de la comunicación. Se encarga primordialmente de subdividir un paquete recibido en partes mas pequeñas si es necesario.

Capa Sesión

Esta capa ofrece varios servicios que son cruciales para la comunicación, como son: 1 Control de la sesión a establecer entre el emisor y el receptor (quién transmite, quién escucha y seguimiento de ésta). 2 Control de la concurrencia (que dos comunicaciones a la misma operación crítica no se efectúen al mismo tiempo). 3 Mantener puntos de verificación (checkpoints), que sirven para que, ante una interrupción de transmisión por cualquier causa, la misma se pueda reanudar desde el último punto de verificación en lugar de repetirla desde el principio. Por lo tanto, el servicio provisto por esta capa es la capacidad de asegurar que, dada una sesión establecida entre dos máquinas, la misma se pueda efectuar para las operaciones definidas de principio a fin, reanudándolas en caso de interrupción. En muchos casos, los servicios de la capa de sesión son parcialmente, o incluso, totalmente rescindibles.

Capa Presentación

El objetivo de la capa de presentación es encargarse de la representación de la información, de manera que aunque distintos equipos puedan tener diferentes representaciones internas de caracteres (ASCII, Unicode, EBCDIC), números (little-endian, big-endian), sonido o imágenes, los datos lleguen de manera reconocible.

Capa Aplicación

Son las aplicaciones en sí: navegador, cliente ftp, cliente de correo, etcétera.

Conceptos de TCP/IP.

La familia de protocolos de *Internet* es un conjunto de protocolos de red que implementa la pila de protocolos en la que se basa Internet y que permiten la transmisión de datos entre redes de computadoras (con distintos sistemas operativos). En ocasiones se la denomina conjunto de protocolos TCP/IP, en referencia a los dos protocolos más importantes que la componen: Protocolo de Control de Transmisión (TCP) y Protocolo de Internet (IP), que fueron los dos primeros en definirse, y que son los más utilizados de la familia.

La familia de protocolos de internet puede describirse por analogía con el modelo OSI. El modelo de Internet fue diseñado como la solución a un problema práctico de ingeniería. El modelo OSI, en cambio, fue propuesto como una aproximación teórica y también como una primera fase en la evolución de las redes de computadoras. Por lo tanto, el modelo OSI es más fácil de entender, pero el modelo TCP/IP es el que realmente se usa. Sirve de ayuda entender el modelo OSI antes de conocer TCP/IP, ya que se aplican los mismos principios, pero son más fáciles de entender en el modelo OSI.

Diferencias TCP vs. UDP.

Como dijimos en el apartado anterior, TPC/IP contiene un conjunto de protocolos, entre los que podemos nombrar TCP (Protocolo de Control de Transmisión) y UDP (Protocolo de Datagrama de Usuario). Estos protocolos tienen varias diferencias que se pueden ver en la siguiente tabla:

Característica	TCP	UDP
Orientado a la conexión	Sí	No
Utilizado en tiempo real	No	Sí
Recuperación de errores	Sí	No
Control de flujo	Sí	No

De la tabla anterior podemos deducir que UDP se utiliza mayormente para transmitir audio y video, y para conexiones de juegos en red, ya que lo primordial en esos casos es la velocidad y no es tan importante si se pierde algún paquete. Asimismo se utiliza para enviar mensajes de broadcast en una red.

TCP se utiliza cuando la información a transmitir debe llegar por completo y se decide resignar un poco de velocidad a cambio de la confiabilidad. Es un protocolo fiable que se encarga de asegurar el correcto orden de llegada de los paquetes y la retransmisión ante una eventual pérdida. Por este motivo las transmisiones FTP, por ejemplo, utilizan TCP. Recordemos que es orientado a la conexión; lo que implica que determina de alguna manera si el otro extremo está presente.

Definición de IP y puerto

Cuando trabajamos con TPC/IP, cada computadora debe tener asociada una dirección IP única dentro de una red. Las direcciones **IPv4** se componen de 4 octetos binarios, que generalmente se representan en forma decimal para recordar las direcciones más fácilmente. Un ejemplo de una dirección IP sería 192.168.1.100

En la actualidad y desde hace ya algunos años se comenzó a utilizar la nueva versión del protocolo **IPv6** que comenzó a desarrollarse en el año 1996. Su principal característica es la

expansión del espacio de direcciones con respecto al anterior. Ya que se estima que las direcciones IPv4 disponibles se terminarán para Junio de 2011. La nueva versión dispone direcciones de 128 bits que nos deja con 5×10^{28} direcciones por cada habitante del planeta (datos del año 2010). Podemos dimensionar el número diciendo que se dispone de la misma cantidad de direcciones por persona que la cantidad de átomos en una tonelada de carbón. Para presentar una dirección se escriben ocho grupos de cuatro dígitos hexadecimales: **2001:db8:85a3:0:0:8a2e:370:7334** y se simplifica omitiendo los ceros a la izquierda de cada grupo. Aunque existen otras convenciones para reducir aún más la escritura de las mismas.

Además de la dirección que la identifica, cada computadora tiene interfaces con el exterior. Éstas son llamadas puertos (*ports*). Desde el punto de vista lógico, las conexiones de una computadora a otra se hacen a través de ellos. Por ejemplo, cuando nos conectarnos con el sitio web de nuestra materia (www.7542.fi.uba.ar) utilizando nuestro navegador, se abre socket apuntando a la dirección IP 157.92.44.19 y al puerto 80 (protocolo HTTP). El puerto generalmente tiene asociado un protocolo determinado por convención, aunque esto no es un estándar rígido. Por ejemplo FTP asociado al puerto 21, HTTP al 80, SSH al 22.

Es importante notar que una conexión socket completa se compone de la información de ambos extremos de la comunicación: dirección y puerto de cada uno de los dos equipos involucrados. De esta forma cuando un navegador web se conecta a un servidor, en ambos extremos existe un socket con la misma información. Tal como se ve en el ejemplo:

Cliente	Servidor
201.55.68.123 : 18455	200.49.150.78 : 80

Más adelante notaremos que en la conexión típica el cliente especifica sólo la dirección y puerto destino. Muy pocas veces indica la dirección local y casi nunca el puerto. Estos datos normalmente los determina el sistema operativo en forma automática. Y en particular los puertos menores a 1024 están reservados para ser utilizados sólo por un usuario con privilegios de administrador del equipo. Los puertos desde el 1024 hasta el 49151 están registrados por la IANA para determinados destinos. Los restantes desde el 49152 hasta el 65535 están destinados a uso privado, efímero o dinámico.

También hay que resaltar que el estándar IP define "big-endian" como el orden de bytes para los números en la red. Por lo tanto será necesario especificar los puertos de esa forma. La API de sockets proveer las funciones `htonl()` y `htons()` para convertir el orden de equipo a orden de red para los tipos *long* y *short* respectivamente. Y las antagónicas `ntohl()` y `ntohs()`.

Uso, configuración y puesta en marcha

Uso

A la hora de comunicar dos programas existen varias posibilidades para establecer la conexión inicialmente. Uno de los programas debe estar arrancado y en espera de que otro quiera conectarse a él. Nunca da "el primer paso" en la conexión. Al programa que actúa de esta forma se le conoce como **servidor**. Su nombre se debe a que normalmente es el que presta servicios a otros programas. Usualmente es quien tiene la información que se desea obtener desde otro punto o bien tiene la capacidad de realizar tareas según se lo soliciten. Por ejemplo, el servidor de páginas web tiene las páginas web y se las envía al navegador que se las requiere.

El otro programa es el que da el primer paso. En el momento de arrancarlo, o cuando lo necesite, intenta conectarse al servidor. Este programa se denomina **cliente**. Su nombre se debe a que es el que solicita información o la ejecución de tareas al **servidor**. El cliente de FTP por ejemplo puede

solicitar la creación de un directorio en el servidor una vez debidamente conectado.

Configuración

Podemos dividir asimismo a los sockets en bloqueantes y no bloqueantes según su variante de configuración más habitual. Los bloqueantes (que son los que utilizaremos en el curso) actúan de la siguiente forma: cuando realizamos una llamada a alguna función relacionada con sockets que no puede ser completada inmediatamente, nuestro proceso pasa al estado de dormido esperando que se satisfaga alguna condición que permita que se complete la llamada. Por ejemplo, el socket servidor se quedará esperando por una conexión entrante; bloqueará en la función *accept* hasta que el sistema operativo informe que hay una nueva conexión disponible.

Con sockets no bloqueantes las funciones de entrada y salida que no puede realizar su tarea por falta de datos retornan inmediatamente y devuelven un código de error.

Comparativamente la ventaja de utilizar sockets bloqueantes (en conjunto con programación multitarea (*multitasking*)) a utilizar no bloqueantes es que los primeros permiten esperar eventos de entrada salida fácilmente sin ocupar tiempo de CPU. Esta diferencia se hace mucho más evidente desde el punto de vista del servidor.

Puesta en marcha y operación

Los pasos para crear y configurar un socket del lado del cliente son:

1. Crear un socket con la función `socket()`
2. Enlazar el socket a la dirección del servidor utilizando `connect()`
3. Enviar y recibir datos con `send()` y `recv()`
4. Cerrar todos los sockets debidamente con `close()` al finalizar.

Los pasos para establecer un socket del lado del servidor son:

1. Crear un socket con la función `socket()`
2. Enlazar el socket a una dirección utilizando la función `bind()`. Para un socket en internet, la dirección consiste en un numero de puerto en la maquina donde es ejecutado.
3. Escuchar por conexiones con la función `listen()`
4. Aceptar conexiones con la función `accept()`
5. Enviar y recibir datos con `send()` y `recv()`
6. Finalizar la transmisión en el socket que escucha conexiones con `shutdown()`
7. Cerrar todos los sockets debidamente con `close()` al finalizar.

Importante

El cierre de una conexión TCP correctamente establecida es algo complejo. Ambas partes envían paquetes especiales para notificar y confirmar el cierre de la conexión. Y por la naturaleza segura del protocolo si alguna confirmación se pierde o se retrasa, es posible que una de las partes pueda volver a enviar un paquetes. Si un caso de retransmisión ocurre innecesariamente existirá un paquete duplicado. Adicionalmente ese duplicado podrá terminar siendo recibido por una nueva conexión, si esta se establece en el mismo puerto que la recientemente cerrada.

Para evitar este problema existe un estado para los sockets TCP llamado `TIME_WAIT`. El mismo es manejado en forma automática por el kernel del sistema operativo. Y si un socket que estuvo escuchando conexiones en un puerto determinado queda en ese estado. El puerto permanecerá inutilizado hasta que el kernel disponga el final del `TIME_WAIT`. Este comportamiento es correcto y

esperado. A menos que se produzca porque una de las partes no llamó a `close()` sobre el socket.

Es común tentarse a utilizar la modalidad de creación de socket que indica que se permita la reutilización del puerto. Y así poder iniciar inmediatamente el programa sin recibir un error originado por un socket anterior en estado `TIME_WAIT`. Esto **no es correcto** y no debe hacerse.

Sólo con fines de desarrollo se recomienda reiniciar el servicio de red para no tener que esperar la finalización del estado `TIME_WAIT`. En linux esto es posible normalmente con `"/etc/init.d/networking restart"`. Esto reiniciará TODAS las conexiones, utilizarlo con cuidado.

Resumen de funciones

<code>int socket(int domain, int type, int protocol)</code>	Crea el socket obteniendo un filedescriptor del sistema operativo. Recibe las características de configuración básicas.
<code>struct hostent * gethostbyname(const char *name)</code> <code>struct hostent * gethostbyaddr(const void *addr, socklen_t len, int type)</code>	Funciones para resolver direcciones de red en función del nombre o vice versa.
<code>int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)</code>	Enlaza el socket identificado por el filedescriptor con una dirección y puerto locales.
<code>int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)</code>	Conecta el socket a la dirección y puerto destino. Determina dirección y puertos locales si no se utilizó <code>bind()</code> previamente.
<code>int listen(int sockfd, int backlog)</code>	Configura el socket para recibir conexiones en la dirección y puerto previamente determinada mediante <code>bind()</code> .
<code>int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)</code>	Espera una conexión en el socket previamente configurado con <code>listen()</code> .
<code>ssize_t send(int s, const void *buf, size_t len, int flags)</code> <code>ssize_t sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)</code>	Recibe datos a través del socket. La variante <code>sendto()</code> se utiliza con UDP.
<code>ssize_t recv(int s, void *buf, size_t len, int flags)</code> <code>ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)</code>	Envía datos a través del socket. La variante <code>recvfrom()</code> se utiliza con UDP.
<code>int shutdown(int s, int how)</code>	Se utiliza para cerrar el envío y la recepción de datos en forma ordenada.
<code>int close(int fd)</code>	Se utiliza para cerrar el socket y liberar los recursos.

Errores típicos

A la hora de comenzar a programar sockets existen algunos errores comunes que vale la pena aclarar en forma anticipada:

1. Creer que el argumento **backlog** de la función **listen()** determina el máximo posible de conexiones.

El argumento hace referencia a la cantidad máxima de conexiones que el sistema operativo permitirá en espera de ser aceptadas. Entre una llamada a la función `accept()` y la siguiente

pueden recibirse varios pedidos de conexión. Si ese número supera al valor de backlog el sistema operativo comenzará a rechazar las conexiones sobrantes. Si luego se llama a la función `accept` se tomará una conexión de la cola liberando un nuevo lugar.

2. *Suponer que la función **send()** enviará el total de información que se le pasa en el argumento **buffer**.*

La función enviará la mayor cantidad posible de información y retornará un valor que indica justamente cuántos bytes pudieron ser enviados. Es responsabilidad del programador asegurarse de que se vuelva a llamar esta función con la porción restante de información. Y de que esto se repita hasta el envío completo.

3. *Asumir que la función **recv()** recibirá la cantidad de bytes especificada en el argumento **length** o bien que recibirá exactamente un paquete enviado desde el otro extremo.*

La función recibe la cantidad de bytes disponibles y la guarda en el buffer pasado por el programador. Luego retorna la cantidad de bytes recibidos, que desde luego no superará el tamaño del buffer. La próxima llamada a esta función puede retornar más información o bien bloquearse si no hay más datos.

El hecho de que un extremo de la comunicación haya realizado un envío de treinta bytes en una sola llamada a `send()`, no implica que los mismos sean recibidos en una sola llamada a `recv()` aunque el buffer tenga suficiente espacio para los treinta bytes. Esto puede o no ocurrir.

4. *Terminar la ejecución de un programa que recibe conexiones sin liberar correctamente el socket sobre el que se está ejecutando la función **accept()**.*

Normalmente existirá un hilo separado para la recepción de conexiones, y estará la mayor parte del tiempo bloqueado en `accept()`. Cuando el programa deba finalizar será necesario utilizar la función `shutdown()` sobre el socket para desbloquear la llamada a `accept()`.

Ejemplo de cliente

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <errno.h>
5. #include <string.h>
6. #include <netdb.h>
7. #include <sys/types.h>
8. #include <netinet/in.h>
9. #include <sys/socket.h>
10.
11. extern int errno;
12. #define PORT 4321 // puerto al que vamos a conectar
13. #define MAXDATASIZE 100 // máximo número de bytes que se pueden leer de una vez
14.
15. int main(int argc, char *argv[])
16. {
17.     int sockfd, numbytes;
18.     char buf[MAXDATASIZE];
19.     struct hostent *he;
20.     struct sockaddr_in their_addr; // información de la dirección de destino
21.
22.     if (argc != 2) {
```

```

23.     fprintf(stderr,"uso: cliente hostnamen P.ej.: cliente localhost");
24.     exit(1);
25. }
26. if ((he=gethostbyname(argv[1])) == NULL) { // obtener información de máquina
27.     perror("gethostbyname");
28.     printf("%s\n",strerror(errno));
29.     exit(1);
30. }
31. // Socket internet, y un Stream (TCP)
32. if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
33.     perror("socket");
34.     printf("%s\n",strerror(errno));
35.     exit(1);
36. }
37.
38. // Completamos la estructura con la dirección y puerto destino
39. their_addr.sin_family = AF_INET; // Orden de bytes de la máquina
40. their_addr.sin_port = htons(PORT); // short, Orden de bytes de la red
41. their_addr.sin_addr = *((struct in_addr *)he->h_addr);
42. memset(&(their_addr.sin_zero), 8, sizeof(int)); // poner a cero el resto
43.
44. if (connect(sockfd, (struct sockaddr *)&their_addr,sizeof(struct sockaddr)) ==
-1) {
45.     perror("connect");
46.     printf("%s\n",strerror(errno));
47.     exit(1);
48. }
49. if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
50.     perror("recv");
51.     printf("%s\n",strerror(errno));
52.     exit(1);
53. }
54. buf[numbytes] = '\0';
55. printf("Received: %s\n",buf);
56. close(sockfd);
57.
58. return 0;
59.}

```

Ejemplo de servidor

```

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <errno.h>
4. #include <string.h>
5. #include <netdb.h>
6. #include <arpa/inet.h>
7. #include <sys/types.h>

```

```

8.
9.  extern int errno;
10. #define SERVER_PORT 4321
11.
12. int main(int argc, char *argv[])
13. {
14.     unsigned int sockfd, newsockfd, clilen;
15.     struct sockaddr_in cli_addr, serv_addr;
16.
17.     printf("Creando socket...\n");
18.     // Creo el socket
19.     if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
20.         perror("creando socket");
21.         printf("%sn", strerror(errno));
22.         exit(1);
23.     }
24.
25.     // Lleno de ceros la estructura
26.     memset((char *)&serv_addr, 0, sizeof(serv_addr));
27.     serv_addr.sin_family = AF_INET; // Tipo internet
28.     serv_addr.sin_port = htons(SERVER_PORT);
29.     serv_addr.sin_addr.s_addr = INADDR_ANY;
30.
31.     // Enlazo la dirección al socket
32.     printf("Bindeando...\n");
33.     if ( bind( sockfd,
34.             (struct sockaddr *)&serv_addr,
35.             (socklen_t)sizeof(struct sockaddr)) < 0 ) {
36.         perror("bind");
37.         printf("%sn", strerror(errno));
38.         exit(1);
39.     }
40.
41.     printf("Escuchando...\n");
42.     listen(sockfd, 5); // Escucho por conexiones
43.     printf("Aceptando...\n");
44.     clilen = (socklen_t)sizeof(cli_addr);
45.
46.     // Acepto la conexión entrante
47.     newsockfd = accept(sockfd, (struct sockaddr *)&cli_addr, &clilen);
48.     if (newsockfd < 0) {
49.         perror("accept");
50.         printf("%sn", strerror(errno));
51.         exit(1);
52.     }
53.

```

```
54.     printf("Conexión establecida.n");
55.     write(newsockfd, "Recibí el mensajen", 19);
56.     close(sockfd);
57.
58.     return 0;
59. }
```