Práctica 2.1: Programación con Sockets

Objetivos

En esta práctica, nos familiarizaremos con la interfaz de programación de sockets como base para la programación de aplicaciones basadas en red, poniendo de manifiesto las diferencias de programación entre los protocolos UDP y TCP. Además, aprenderemos a programar servidores concurrentes en ambos transportes.

Contenidos

Preparación del entorno para la práctica

La práctica debe realizarse en el puesto físico del laboratorio, sin necesidad de arrancar ninguna máquina virtual. Las pruebas de funcionamiento de los programas se harán con las IP's configuradas en la máquina del laboratorio, normalmente la dirección de `loopback`.

Llamadas del API para la gestión de direcciones

El uso del API BSD requiere la manipulación de direcciones de red, y traducción de estas entre las tres representaciones básicas: el nombre de dominio, la dirección IP y binario (que finalmente se envía en la red como campo dirección origen en la cabecera del datagrama IP).

Ejercicio 1. Escribir un programa que obtenga todas las posibles direcciones con las que se podría crear un socket asociado a un host dado (primer argumento del programa). Para cada dirección mostrar la IP numérica, la familia y tipo de socket. Comprobar el resultado para:

- Una dirección IPv4 en formato punto (ej. "`147.96.1.9`")
- Un nombre de dominio (ej. "`www.google.com`")
- Un nombre en `/etc/hosts` (ej. "`localhost`")
- Una dirección o nombre incorrectos en cualquiera de los casos anteriores

El programa se implementará usando la función `getaddrinfo(3)` para obtener la lista de posibles conexiones (`struct sockaddr *`). Cada dirección se imprimirá en su valor numérico, usando la función `getnameinfo(3)` con el *flag* `NI_NUMERICHOST`, así como la familia de direcciones y el tipo de socket también en su equivalente numérico. Para obtener todos las familias posibles usar como opción de filtrado `AF_UNSPEC`;

Ejemplo de ejecución

```
>./gai www.google.com
66.102.1.147 2     1
66.102.1.147 2     2
66.102.1.147 2     3
2a00:1450:400c:c06::67    10     1
2a00:1450:400c:c06::67    10     2
2a00:1450:400c:c06::67    10     3
```

```
> ./gai localhost
::1    10     1
::1    10     2
::1    10     3
127.0.0.1    2      1
127.0.0.1    2      2
127.0.0.1    2      3
> ./gai noexiste.ucm.es
Error: Name or service not known
```

Nota: La ejecución correcta del programa necesita una configuración correcta del cliente DNS del sistema. Las constantes para el tipo de socket y familia están definidas en `/usr/include/bits/socket.h` y `/usr/include/bits/socket_type.h`: Los protocolos 2 y 10 son AF_INET y AF_INET6, respectivamente y los tipos 1, 2, 3 son SOCK_STREAM, SOCK_DGRAM y SOCK_RAW, respectivamente

Protocolo UDP - Servidor de hora

Ejercicio 2. Usando como base el servidor estudiado en clase, escribir un servidor que use el protocolo UDP, de forma que:

- La dirección y el puerto son el primer y segundo argumento del programa. Las direcciones (AF_INET) pueden expresarse en cualquier formato, p. ej. nombre de host o notación punto.
- El servidor recibirá un comando (codificado en un carácter), de forma que: `'t'` devuelva la hora, `'d'` la fecha y `'q'` termina el proceso servidor.
- En cada mensaje el servidor debe imprimir el nombre y puerto del cliente, usar la función `getnameinfo(3)`.

Probar el funcionamiento del servidor con el comando Netcat (`nc`).

Ejemplo servidor de hora UDP

```
$ ./time_server 0.0.0.0 3000
2 bytes de 127.0.0.1:58772
2 bytes de 127.0.0.1:58772
2 bytes de 127.0.0.1:58772
Comando no soportado X
2 bytes de 127.0.0.1:58772
Saliendo...
```

```
$ nc -u 127.0.0.1 3000
t
10:30:08 PMd
2018-01-14X
q
^C
$
```

Nota: El servidor no envía `'\n'` y `nc` muestra la respuesta y el siguiente comando (en negrita en el ejemplo) en la misma línea.

Nota: La hora se puede obtener con la función `localtime(3)` y formatearla (obtener la fecha o la hora) con `strftime(3)`

Ejercicio 3. Escribir el cliente para el servidor de hora, similar al funcionamiento del comando `nc`. El cliente tendrá por parámetros la dirección y el puerto del servidor y el comando. Por ejemplo, `./time_client 127.0.0.1 3000 t`, para solicitar la hora.

Protocolo TCP - Servidor de eco

TCP nos ofrece un servicio orientado a conexión y fiable. Una vez creado el socket, debe ponerse en estado LISTEN (apertura pasiva, `listen(2)`) y a continuación quedarse a la espera de conexiones entrantes mediante una llamada `accept(2)`.

Ejercicio 4. Utilizando sockets TCP, crear un servidor de eco que escuche por conexiones entrantes en una dirección IPv4 y puerto dados. Cuando reciba una conexión entrante, debe mostrar la dirección y número de puerto del cliente. A partir de ese momento, enviará al cliente todo lo que reciba de éste (eco). Comprobar su funcionamiento empleando el comando Netcat (`nc`) como cliente. Comprobar qué sucede si varios clientes intentan conectar al mismo tiempo.

Ejemplo servidor de eco TCP	

Servidor:
`$ ./echo_server 0.0.0.0 2222`
`Conexión desde 127.0.0.1 53456`
`Conexión terminada`
	Cliente:
	`$ nc 127.0.0.1 2222`
	Hola
	`Hola`
	Qué tal
	`Qué tal`
	^C
	`$`

Ejercicio 5. Escribir el cliente para conectarse con el servidor del ejercicio anterior. El cliente debe tomar la dirección y el puerto del servidor desde la línea de órdenes (pasados como parámetros) y una vez establecida la conexión con el servidor le enviará lo que el usuario escriba por teclado. Mostrará en la consola la respuesta recibida desde el servidor. Cuando el usuario escriba la letra `'Q'` como único carácter de una línea, el cliente cerrará la conexión con el servidor.

Ejemplo servidor de eco con cliente	

Servidor:
`$ ./echo_server 0.0.0.0 2222`
`Conexión desde 127.0.0.1 53445`
`Conexión terminada`
`$`
	Cliente:
	`$ ./echo_client 127.0.0.1 2222`
	Hola
	`Hola`
	Q
	`$`

Servidores UDP y TCP multi-threaded

Ejercicio 6. Convertir el servidor UDP del ejercicio 2 en multi-threaded siguiendo un modelo *pre-fork*, con las siguientes características:

- El thread principal inicializará el socket del servidor con `socket(2)` y `bind(2)`.
- El thread principal creará un conjunto de threads para tratar cada mensaje recibido.
- Los threads para tratar los mensajes se implementarán con su propia clase. El constructor únicamente necesita el descriptor del socket creado. Además tendrán un único método que permanecerá en un bucle recibiendo mensajes, `recvfrom(2)`.

- Para poder comprobar la concurrencia añadir una latencia artificial con la llamada `sleep(3)` en cada mensaje. Imprimir además el id del thread que está tratando el mensaje, por ejemplo `std::this_thread::get_id()`.
- El thread principal esperará leyendo de la terminal hasta que reciba el caracter 'q', que terminará todos los threads del servidor.

Ejercicio 7. Convertir el servidor TCP del ejercicio 4 en multi-threaded siguiendo una estrategia *accept-and-fork*:

- El thread principal inicializará el socket del servidor con `socket(2)` y `bind(2)`; y permanecerá en un bucle esperando conexiones, `accept(2)`.
- Cuando se reciba una conexión, el thread principal creará un thread para tratar esa conexión.
- Los threads para tratar las conexiones se implementarán con su propia clase. El constructor recibirá en este caso el descriptor del socket de la conexión, retornado por accept(2). Además tendrán un único método que permanecerá en un bucle recibiendo/enviando mensajes hasta que se desconecte el cliente.