Práctica 2.5: 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 aplicaciones independientes de la familia de protocolos de red (IPv4 o IPv6) utilizados.

Contenidos

Preparación del entorno para la práctica
Llamadas del API para la gestión de direcciones
Protocolo UDP - Servidor de hora
Protocolo TCP - Servidor de eco

Preparación del entorno para la práctica

Configuraremos la topología de red que se muestra en la Figura 1. Como en prácticas anteriores construiremos la topología con la herramienta `vtopol` y un archivo de topología adecuado. Antes de comenzar la práctica, configurar los interfaces de red como se indica en la figura y comprobar la conectividad entre las máquinas.

Nota: Observar que las VMs tienen un interfaz de red con pila dual IPv6 - IPv4.

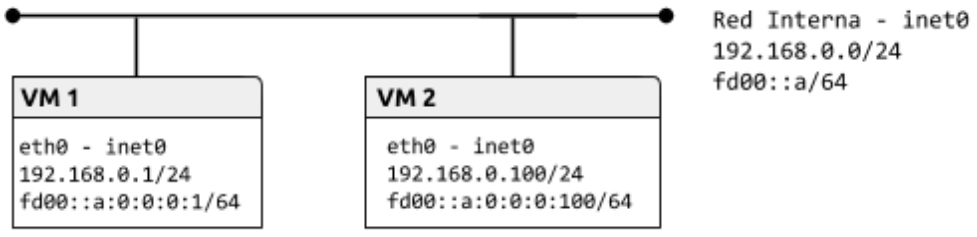

Figura 1: Topología y direccionamiento de las red usada en esta práctica

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 (tanto versión 4 como 6) y binario (que finalmente se envían 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`")
- Una dirección IPv6 en formato punto (ej. "`fd00::a:0:0:0:1`")
- 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.

Nota: Para probar el comportamiento de las funciones y el servicio DNS, realizar este ejercicio en el host anfitrión.

Ejemplo de ejecución

```
# Los protocolos 2 y 10 son AF_INET y AF_INET6, respectivamente (ver socket.h)
# Los tipos 1, 2, 3 son SOCK_STREAM, SOCK_DGRAM y SOCK_RAW, respectivamente
>./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 ::1
::1    10     1
::1    10     2
::1    10     3
> ./gai 1::3::4
Error getaddrinfo(): Name or service not known
> ./gai noexiste.ucm.es
Error: Name or service not known
```

Protocolo UDP - Servidor de hora

Ejercicio 1. 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 pueden expresarse en cualquier formato, esto es nombre de host, notación punto... Además, el servidor debe funcionar con direcciones IPv4 e IPv6 .
- 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`).

Nota: Dado que el servidor puede funcionar con direcciones IPv4 e IPv6, hay que usar una estructura que permita acomodar cualquiera de ellas, por ejemplo en `recvfrom(3)`. El API BSD define el tipo `sockaddr_storage` para estas situaciones.

Ejemplo servidor de hora UDP

$ **./time_server :: 3000** 2 bytes de ::FFFF:192.168.0.100:58772 2 bytes de ::FFFF:192.168.0.100:58772 2 bytes de ::FFFF:192.168.0.100:58772 Comando no soportado X 2 bytes de ::FFFF:192.168.0.100:58772 Saliendo...	$ **nc -u 192.168.0.1 3000** **t** 10:30:08 PM**d** 2014-01-14**X** **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.

Ejercicio 2. 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 192.128.0.1 3000 t`, para solicitar la hora.

Ejercicio 3. Modificar el servidor para que además de poder recibir comandos por red, los pueda recibir directamente del terminal, leyendo dos caracteres (comando y '\n') de la entrada estándar. Multiplexar el uso de ambos flujos de datos usando la función `select(2)`.

Ejercicio 4 *(Opcional)***.** Convertir el servidor UDP en multi-proceso siguiendo un modelo *pre-fork*. Una vez asociado el socket a la dirección después de la llamada a `bind(2)`, realizar la llamada `recvfrom` en procesos diferentes de forma que cada uno atenderá una conexión de un cliente distinto. Imprimir el PID del proceso servidor para comprobarlo.

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 1. Utilizando sockets sobre TCP, crear un servidor de eco que escuche por conexiones entrantes en una dirección (IPv4 o IPv6) 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 desde el mismo (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 :: 2222** Conexión desde fd00::a:0:0:0:1 53456 Conexión terminada	**Cliente**: $ **nc -6 fd00::a:0:0:0:1 2222** **Hola** Hola **Qué tal** Qué tal **^C** $

Ejercicio 2. Escribir el cliente para conectarse con el servidor del ejercicio 1. 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 :: 2222`
`Conexión desde fd00::a:0:0:0:1 53445`
`Conexión terminada`
`$`
	Cliente:
	`$ ./echo_client fd00::a:0:0:0:1 2222`
	Hola
	Hola
	Q
	`$`

Ejercicio 3. Tratar cada petición en un proceso diferente con `fork(2)`. Se debe modificar el código del servidor para que acepte varias conexiones simultáneas. Cada conexión se gestionará en un proceso hijo (modelo *accept-and-fork*). El proceso padre debe cerrar el socket devuelto por la llamada `accept(2)`.

Ejercicio 4. Añadir la lógica necesaria para que el servidor sincronice la finalización de los procesos hijos (por ejemplo, capturando la señal `SIGCHLD`) de forma que no quede ningún proceso en estado *zombie*.