Práctica 2.5. Sockets

Objetivos

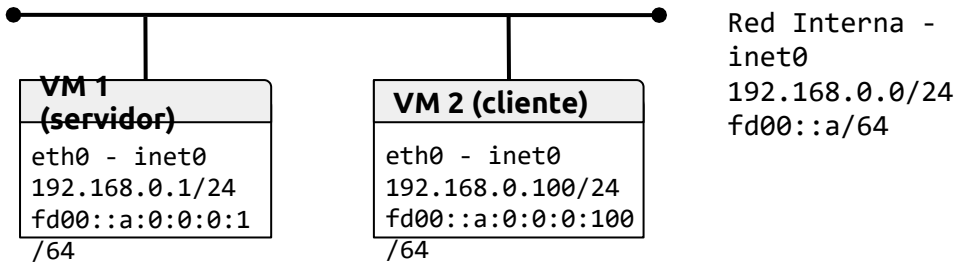
En esta práctica, nos familiarizaremos con la interfaz de programación de sockets como base para la programación de aplicaciones basadas en red, poniendo de manifiesto las diferencias de programación entre los protocolos UDP y TCP. Además, aprenderemos a programar aplicaciones independientes de la familia de protocolos de red (IPv4 o IPv6) utilizados.

Contenidos

Preparación del entorno de la práctica
Gestión de direcciones
Protocolo UDP - Servidor de hora
Protocolo TCP - Servidor de eco

Preparación del entorno de la práctica

Configuraremos la topología de red que se muestra en la figura. Como en prácticas anteriores construiremos la topología con la herramienta `vtopol`. 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.

Gestión de direcciones

El uso del API BSD requiere la manipulación de direcciones de red y la traducción de estas entre las tres representaciones básicas: nombre de dominio, dirección IP (versión 4 y 6) y binario (para incluirla 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 como primer argumento del programa. Para cada dirección, mostrar la IP numérica, la familia de protocolos y tipo de socket. Comprobar el resultado para:

- Una dirección IPv4 válida (ej. "`147.96.1.9`").
- Una dirección IPv6 válida (ej. "`fd00::a:0:0:0:1`").
- Un nombre de dominio válido (ej. "`www.google.com`").
- Un nombre en `/etc/hosts` válido (ej. "`localhost`").
- Una dirección o nombre incorrectos en cualquiera de los casos anteriores.

El programa se implementará usando `getaddrinfo(3)` para obtener la lista de posibles direcciones de socket (**`struct`** `sockaddr`). Cada dirección se imprimirá en su valor numérico, usando `getnameinfo(3)` con el *flag* `NI_NUMERICHOST`, así como la familia de direcciones y el tipo de socket.

Nota: Para probar el comportamiento con DNS, realizar este ejercicio en la máquina física.

Ejemplos:

```
# Las familias 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 getaddrinfo(): Name or service not known
```

Protocolo UDP - Servidor de hora

Ejercicio 2. Escribir un servidor UDP de hora de forma que:

- La dirección y el puerto son el primer y segundo argumento del programa. Las direcciones pueden expresarse en cualquier formato (nombre de host, notación de 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'` devuelve la fecha y `'q'` termina el proceso servidor.
- En cada mensaje el servidor debe imprimir el nombre y puerto del cliente, usar `getnameinfo(3)`.

Probar el funcionamiento del servidor con la herramienta Netcat (comando `nc` o `ncat`) como cliente.

Nota: Dado que el servidor puede funcionar con direcciones IPv4 e IPv6, hay que usar **`struct`** `sockaddr_storage` para acomodar cualquiera de ellas, por ejemplo, en `recvfrom(2)`.

```
                    DESDE: $ man getaddrinfo
                         EJERCICIO 2-5.c
#include <sys/types.h>
      #include <stdio.h>
      #include <stdlib.h>
      #include <unistd.h>
      #include <string.h>
      #include <sys/socket.h>
      #include <netdb.h>
```

```c
#include <time.h>

#define BUF_SIZE 500

int
main(int argc, char *argv[])
{
    struct addrinfo hints;
    struct addrinfo *result, *rp;
    int sfd, s;
    struct sockaddr_storage peer_addr;
    socklen_t peer_addr_len;
    ssize_t nread;
    char buf[BUF_SIZE];

    if (argc >3) {
        fprintf(stderr, "Usage: %s port\n", argv[2]);
        exit(EXIT_FAILURE);
    }

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;    /* Allow IPv4 or IPv6 */
    hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */
    hints.ai_flags = AI_PASSIVE;    /* For wildcard IP address
*/
    hints.ai_protocol = 0;          /* Any protocol */
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;

    s = getaddrinfo(argv[1], argv[2], &hints, &result);
    if (s != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
        exit(EXIT_FAILURE);
    }
   for (rp = result; rp != NULL; rp = rp->ai_next) {
        sfd = socket(rp->ai_family, rp->ai_socktype,
                rp->ai_protocol);
        if (sfd == -1)
            continue;

        if (bind(sfd, rp->ai_addr, rp->ai_addrlen) == 0)
```

```
                break;                  /* Success */

            close(sfd);
        }

        freeaddrinfo(result);           /* No longer needed */

        if (rp == NULL) {               /* No address succeeded */
            fprintf(stderr, "Could not bind\n");
            exit(EXIT_FAILURE);
        }

        /* Read datagrams and echo them back to sender */

        for (;;) {
            peer_addr_len = sizeof(peer_addr);
            nread = recvfrom(sfd, buf, BUF_SIZE, 0,
                    (struct sockaddr *) &peer_addr, &peer_addr_len);
            if (nread == -1)
                continue;               /* Ignore failed request */

            char host[NI_MAXHOST], service[NI_MAXSERV];

            s = getnameinfo((struct sockaddr *) &peer_addr,
                            peer_addr_len, host, NI_MAXHOST,
                            service, NI_MAXSERV, NI_NUMERICSERV);
            if (s == 0)
                printf("Received %zd bytes from %s:%s\n",
                        nread, host, service);
            else
                fprintf(stderr, "getnameinfo: %s\n",
gai_strerror(s));

          /*  if (sendto(sfd, buf, nread,
0,//----------------------------AQUI ENVIO RESPUESTA CLIENTE
                        (struct sockaddr *) &peer_addr,
                        peer_addr_len) != nread)
                fprintf(stderr, "Error sending response\n");*/



//-------------------------------------------------------------------
```

```
--------------------------------SACAR TIEMPO
/*
formato = %Y-%m-%d | %H:%M:%S
*/

   time_t t = time(NULL);
   struct tm tiempoLocal=*localtime(&t);


//-------------------------------------------------------------------------
--------------------------------------AQUÍ LEO Y ENVIO
       int bytes;
       switch(buf[0]){
           case 't':
               bytes = strftime(buf, sizeof(buf), "%H:%M:%S\n",
&tiempoLocal);
       sendto(sfd, buf, bytes, 0, (struct sockaddr *) &peer_addr,
peer_addr_len);
           break;

           case 'd':
               bytes = strftime(buf, sizeof(buf), "%d de %m del %Y \
n", &tiempoLocal);
       sendto(sfd, buf, bytes, 0, (struct sockaddr *) &peer_addr,
peer_addr_len);
           break;

           case 'q':
               printf("Saliendo...\n");
               return 0;
           break;

           default: printf("Error: Comando no soportado %c, debe ser
t,d o q\n",buf[0]);
           break;
       }

   }
}
```

Ejemplo:

$ **./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 **X** no soportado 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** $
DESDE VISUAL (SERVIDOR) **dani@dapire:~/Documentos/Practicas ASO$** ./2-5 :: 3500 Received 2 bytes from ip6-localhost:35141 Error: Comando no soportado x, debe ser t,d o q Received 2 bytes from ip6-localhost:35141 Received 2 bytes from ip6-localhost:35141 Received 2 bytes from ip6-localhost:35141 Saliendo...	**DESDE OTRO TERMINAL (CLIENTE)** **dani@dapire:~$** nc -u :: 3500 x d 01 de 12 del 2022 t 21:59:26 q ^C

Nota: El servidor no envía '\n', por lo que se muestra la respuesta y el siguiente comando (en negrita en el ejemplo) en la misma línea.

Ejercicio 3. Escribir el cliente para el servidor de hora. El cliente recibirá como argumentos la dirección del servidor, el puerto del servidor y el comando. Por ejemplo, para solicitar la hora, `./time_client 192.128.0.1 3000 t`.

Ejercicio 4. Modificar el servidor para que, además de poder recibir comandos por red, los pueda recibir directamente por el terminal, leyendo dos caracteres (el comando y '\n') de la entrada estándar. Multiplexar el uso de ambos canales usando `select(2)`.

Ejercicio 5. Convertir el servidor UDP en multi-proceso siguiendo el patrón *pre-fork*. Una vez asociado el socket a la dirección local con `bind(2)`, crear varios procesos que llamen a `recvfrom(2)` de forma que cada uno atenderá un mensaje de forma concurrente. Imprimir el PID del proceso servidor para comprobarlo. Para terminar el servidor, enviar la señal `SIGTERM` al grupo de procesos.

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 6. Crear un servidor TCP 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 la herramienta Netcat como cliente. Comprobar qué sucede si varios clientes intentan conectar al mismo tiempo.

Ejemplo:

$ **./echo_server :: 2222** Conexión desde fd00::a:0:0:0:100 53456 Conexión terminada	$ **nc -6 fd00::a:0:0:0:1 2222** **Hola** Hola **Qué tal** Qué tal **^C** $

Ejercicio 7. Escribir el cliente para conectarse con el servidor del ejercicio anterior. El cliente recibirá la dirección y el puerto del servidor como argumentos 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 el carácter 'Q' como único carácter de una línea, el cliente cerrará la conexión con el servidor y terminará.

Ejemplo:

`$ ./echo_server :: 2222`
`Conexión desde fd00::a:0:0:0:100 53445`
`Conexión terminada`
	`$ ./echo_client fd00::a:0:0:0:1 2222`
	Hola
	Hola
	Q
	$

Ejercicio 8. Modificar el código del servidor para que acepte varias conexiones simultáneas. Cada petición debe gestionarse en un proceso diferente, siguiendo el patrón *accept-and-fork*. El proceso padre debe cerrar el socket devuelto por `accept(2)`.

Ejercicio 9. Añadir la lógica necesaria en el servidor para que no quede ningún proceso en estado *zombie*. Para ello, se deberá capturar la señal `SIGCHLD` y obtener la información de estado de los procesos hijos finalizados.