PDI - TP 1 - A

Alumnos:
Alonso Facundo
Alvarez Poli Bautista
Kloster Agustin

1) Explicar qué es un socket y los diferentes tipos de sockets.

   Un socket es un descriptor de una conexión de red. Así como un descriptor de archivo es un identificador de tipo entero, asociado a un archivo abierto, un socket permite identificar una conexión de red activa entre un cliente y un servidor.

   Existen varios tipos de sockets, pero vamos a destacar tres en particular “Stream Sockets”, “Datagram Sockets” y “Raw sockets”.

   1) **Stream sockets**
      Stream sockets son flujos de comunicación conectados bidireccionalmente. Por lo tanto utilizan TCP y son confiables. Al utilizar TCP, este crea y mantiene una conexión además de proporcionar los servicios de detección de errores, y que los datos lleguen secuencialmente.
   2) **Datagram sockets**
      Los “datagram sockets” o sockets de datagrama .A diferencia de los anteriores, estos utilizan UDP, por lo tanto tienen una longitud máxima fija, no crean y mantienen una conexión, no mantienen un orden en el recibimiento de datos, no proporcionan una detección de errores pero su velocidad es mayor.
      Algunos ejemplos de su utilización son tftp, dhcpcd (cliente DHCP),bootp y broadcast.
      En tftp es necesario implementar un protocolo encima de UDP si se quiere asegurar que los paquetes lleguen (En este caso implementa ACK).

   3) Raw sockets

      Los sockets sin formato permiten que una aplicación tenga acceso directo a protocolos de comunicación de nivel inferior. No utilizan ningún protocolo. Ejemplos de utilización son para monitorear la red, para piratear, entre otros.

      Los sockets sin formato normalmente están orientados a datagramas, aunque sus características exactas dependen de la interfaz proporcionada por el protocolo.

2) Cuáles son las estructuras necesarias para operar con sockets en el modelo C-S y cómo se hace para ingresar los datos requeridos. Explicar con un ejemplo.

Para poder operar con sockets en el modelo C-S, se utiliza la estructura de datos “sockaddr”, con la cual se define la información de dirección de sockets para múltiples tipos de socket:

```
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};
```

La variable “sa_family” define la familia de direcciones del socket (se utiliza “AF_INET” para IPV4, y “AF_INET6” para IPV6).

El campo “sa_data” contiene la dirección de destino y el número de puerto de dicho socket. Sin embargo, cabe destacar, que su implementación puede resultar ineficiente ya que su uso implica indicar individualmente las direcciones en dicho campo.

Para resolver este inconveniente, se utiliza la estructura “sockaddr_in” (uso exclusivo en IPV4; “in”=internet), que es un derivado de la estructura “sockaddr”. Un puntero de tipo “sockaddr_in” puede forzarse (es decir, hacer un “cast”) al tipo de una estructura “sockaddr” y viceversa:

```
struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

Mediante su uso, se simplifica la administración de direcciones, pues, en este caso, la referencia a la dirección del socket se hace a través de una estructura correspondiente, en lugar de una indicación manual.

El campo “sin_zero” provee compatibilidad con la estructura “sockaddr” en cuanto a definir el mismo tamaño de datos, y se debe rellenar a ceros utilizando la función “memset()”.

La estructura utilizada en el campo “sin_addr” es “in_addr”:

```
// Internet address (a structure for historical reasons)
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
```

3) Explicar modo bloqueante y no bloqueante en sockets y cuáles son los “sockets calls” a los cuales se pueden aplicar estos modos. Explicar las funciones necesarias.

El modo bloqueante y no bloqueante son modos que se pueden aplicar a determinadas funciones para permitir que se bloqueen o no (que sigan “escuchando” o dejen hacerlo).

Las socket calls a las cuales se les puede aplicar estos modos son:

- recv()
- recvfrom()
- accept()

Por defecto estas “calls” están definidas como bloqueantes, si se quisiera cambiar al modo no bloqueante se debe llamar a la función fcntl().

```
#include <unistd.h>
#include <fcntl.h>
.
.
.
sockfd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

El modo bloqueante debe ser más controlado, dado que si se quiere leer de un socket no bloqueante y no hay informacion ahi, retornara -1 y errno se setea como EWOULDBLOCK o EAGAIN.

4) Describir el modelo C-S aplicado a un servidor con Concurrencia Real. Escribir un ejemplo en lenguaje C.

SERVIDOR

En un servidor con concurrencia real se utiliza la función fork() para crear procesos hijo, los cuales son una copia exacta del proceso padre, para así poder manejar simultáneamente varios clientes.

La implementación es de la siguiente forma:

```c
// Función principal
int main()
{
    int sockfd, connfd, len;
    struct sockaddr_in servaddr, cli;
    pid_t pid_hijo;

    // socket create and verification
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("Falla la creación del socket...\n");
        exit(0);
    }
    else
        printf("Socket creado...\n");
    bzero(&servaddr, sizeof(servaddr));

    // asigno IP, PORT
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);

    // Bind del nuevo socket a la dirección IP y lo verifico
    if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
        printf("Falla socket bind ...\n");
        exit(0);
    }
    else
        printf("Se hace el socket bind ..\n");
```

Primero, se crean las variables necesarias y se procede a crear y verificar la creación del socket. Para esto se utiliza la función socket() pasando por parámetros la familia, el tipo y el protocolo.

Luego asigno la IP y el puerto, en el caso de la IP se asigna usando la función htonl(INADDR_ANY), donde el parámetro indica que se utilice la dirección por donde entró tráfico la última vez.

Por último, se hace el bind() del socket creado y la dirección IP resultante.

```
// El servidor está en modo escucha (pasivo) y lo verifico
if ((listen(sockfd, 5)) != 0) {
    printf("Falla el listen ...\n");
    exit(0);
}
else
    printf("Servidor en modo escucha ...\n");
```

A continuación se hace el listen() y el servidor se pone en modo escucha, esperando una conexión del cliente.

```
len = sizeof(cli);
int num = 0;

while(1)
{
    num++;
    // Acepto los paquetes del cliente y verifico
    connfd = accept(sockfd, (SA *)&cli, &len);
    if (connfd < 0)
    {
        printf("Falla al aceptar datos en el servidor...\n");
        exit(0);
    }
    else
        printf("El servidor acepta al cliente[%d] ...\n",num);

    // Creo el proceso hijo
    if ((pid_hijo = fork()) == 0)
    {
        // Estamos en el proceso hijo
        close(sockfd); // El hijo no necesita el socket de escucha

        // Procesamos la conexión con el cliente
        func(connfd,num);

        // Cerramos la conexión del cliente
        close(connfd);

        // Salimos del proceso hijo
        exit(0);
```

Luego se procede al bucle principal del servidor, en donde se acepta la conexión con un accept() y luego se utiliza la función fork(), para crear un proceso hijo y

así poder seguir recibiendo clientes. Se cierra el socket padre dado que el hijo no lo necesita para funcionar y se procede a realizar una función.

Terminada la función se cierra la conexión y se sale del proceso hijo.

```
}
// Cierro el socket al terminar el chat
close(sockfd);

return 0;
```

Finalmente se procede a cerrar el socket, finalizadas todas las conexiones.

CLIENTE

Del lado del cliente el proceso de creación del socket es similar, la única diferencia es a la hora de conectarse al servidor, para lo cual se usa la función connect().

```
// Conecto los sockets entre cliente y servidor
if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr))
    != 0) {
    printf("Falla de conexión con servidor...\n");
    exit(0);
}
else{
    printf("Conectado al servidor..\n");
    printf("Cliente conectado desde: %s:%d\n", inet_ntoa(servaddr.sin_addr), ntohs(servaddr.sin_port));
}
// Función para el chat
func(sockfd,cli);

//Cierro el socket
close(sockfd);
```

5) ¿Cómo se cierra una conexión C-S ?. Métodos que eviten la pérdida de información.

Una conexión C-S se cierra cuando se libera el socket asociado, evitando futuras lecturas y escrituras de información a través de dicha conexión. Para ello, se puede utilizar la función Unix “close()”, con la que se cierran los descriptores de archivo.

Sin embargo, su uso podría ocasionar errores de flujo, tal como pérdidas de información por operaciones inconclusas. Para mitigar este riesgo, primero se debe llamar a la función: “int shutdown(int sockfd, int how)”.

Su implementación provee un mayor control sobre la finalización de la conexión, ya que admite tres valores en “how” (“0” impide recibir datos; “1” impide enviar datos; “2” impide la lectura y escritura, similar al “close”), y además devuelve “0” si tiene éxito o “-1” en caso de error.

En otras palabras, shutdown() no cierra el descriptor, sino que modifica sus condiciones de uso, de tal manera que se puedan implementar algoritmos de control de flujo antes de cerrar el socket con close().