Escuela Politécnica Superior de Elche

SISTEMAS INFORMÁTICOS EN TIEMPO REAL

2º Ingeniería Industrial

PRÁCTICAS DE TCP-IP
Comunicación mediante SOCKETS

Luis Miguel Jiménez

Departamento de Ingeniería de Sistemas Industriales
Área de Ingeniería de Sistemas y Automática

ISA-UMH ©

1 OBJETIVO

Las prácticas de comunicación entre procesos mediante ***sockets*** tienen como objetivo asimilar los conceptos de programación distribuida, así como el manejo a nivel de programador de los protocolos de comunicaciones **TCP/IP** basándose en la interfaz de ***sockets***.

La interfaz de **sockets** facilita la comunicación entre procesos, programando dicha comunicación de manera similar a como se maneja cualquier otro dispositivo de entrada salida. Independiza así mismo la localización de los procesos, que pueden estar ubicados en el mismo o en diferentes computadores.

2 MATERIAL EMPLEADO

Las prácticas se realizan en grupos de dos personas disponiendo de 2 PCs. Los equipos disponen de S.O. Windows XP y el software de edición y conexión remota ***Putty/Crimson Editor*** para acceder al servidor ***SUN Solaris***. Para realizar las prácticas, el alumno debe disponer también de los apuntes de la asignatura.

Los programas correrán en el servidor **SUN Solaris** (**lorca.umh.es**).

3 PROTOCOLOS DE TRANSPORTE

La comunicación entre dos procesos distribuidos se realiza, según la arquitectura **TCP/IP**, apoyándose sobre los protocolos de nivel transporte (en algunos casos se puede saltar esta capa pero esto supone un esfuerzo adicional en la gestión de la comunicación). La arquitectura nos proporciona dos protocolos de transporte con características claramente diferenciadas, y ajustadas a diferentes tipos de necesidades de comunicación. Los dos protocolos proporcionados son el protocolo **UDP** y el protocolo **TCP**.

UDP: es un protocolo orientado a la no conexión (no exige ningún proceso previo a la comunicación), sin garantía de secuencia (los datos pueden llegar desordenados), sin garantía de fiabilidad (los datos pueden contener errores o perderse), y sin control de flujo (no controla si el receptor es capaz de aceptar datos a al misma velocidad que el emisor los envía).

TCP: es un protocolo orientado a la conexión, precisando un etapa previa de conexión (transmisión de mensajes entre emisor y receptor que garanticen el éxito de la comunicación posterior). Se trata de un protocolo fiable y con control de flujo, es decir comprueba si existen errores en la transmisión retransmitiendo aquellos mensajes incorrectos, evitando al mismo tiempo que un receptor lento pueda ser desbordado.

En ambos protocolos se utiliza el concepto de ***puerto*** para identificar a los procesos origen y destino de la información. El conjunto formado por la **dirección IP** y el **puerto** de comunicaciones constituye la información necesaria en una transmisión de datos entre procesos. El ***puerto*** es un número entero de 16 bits que debe ser único para cada proceso. Generalmente existe un conjunto de valores reservados para los procesos estándar del sistema operativo, disponiendo un rango de valores para los procesos del usuario. Estos

últimos pueden ser gestionados de forma manual o de forma automática por el propio sistema operativo.

4 LA INTERFAZ DE SOCKETS

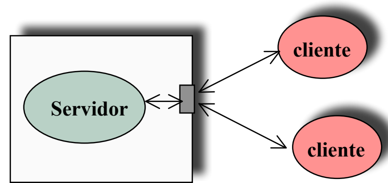
Para facilitar y unificar la programación de la comunicación entre procesos distribuidos, la Universidad de Berkley desarrollo en los año 70 lo que se conoce como ***interfaz de sockets***. Se trata de una librería que permite el manejo de la comunicación entre procesos distribuidos de forma muy similar a como se programa el sistema de Entrada/Salida estándar.

4.1 Modelo Cliente-Servidor

Es importante hacer una mención a la forma de programar una aplicación distribuida en red en la arquitectura TCP/IP. El modelo utilizado para la comunicación de datos es el modelo **Cliente-Servidor**. Este modelo plantea un comportamiento asimétrico en los procesos involucrados en la comunicación.

Uno de los procesos actúa como **servidor.** Se trata de un proceso que debe tener asignado un identificador de puerto fijo y perfectamente conocido (así mismo debe conocerse la dirección IP). Dicho proceso debe estar continuamente esperando datos o peticiones de transmisión (*estado de escucha*).

Por otro lado, uno o varios procesos actúan como **cliente/s**. El cliente es el que inicia la comunicación enviando datos o solicitando datos del servidor. El identificador del *puerto* de cliente no es tan importante y lo asigna automáticamente el S.O. A partir de la cabecera del paquete de datos, el servidor tendrá conocimiento de dicho identificador cuando el cliente le envíe un mensaje (también podrá conocer la dirección IP del cliente).

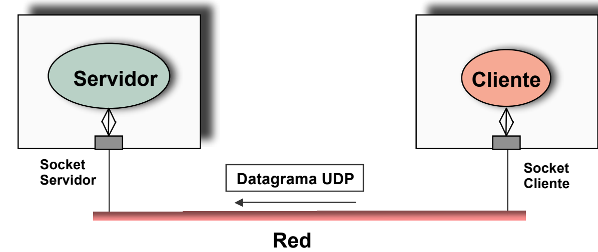


El elemento que canaliza esta comunicación lo denominaremos **socket**, y está constituido por una estructura de datos que maneja los recursos de comunicación.

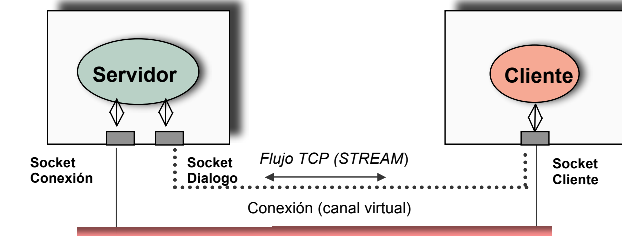
4.2 Diseño de programas basados en sockets

El diseño de aplicaciones distribuidas basadas en la interfaz de sockets está descrito de forma detalla en el capítulo 12 del libro. Los ejemplos que veremos en esta práctica analizan en detalle cada uno de los pasos, y serán la base común de cualquier aplicación avanzada que queramos desarrollar. La forma de trabajar varía de forma importante según usemos *sockets* de tipo ***Datagrama (UDP)*** o de tipo ***Stream (TCP)***.

Sockets de tipo Datagrama:



Sockets de tipo Stream:



En esta práctica nos centraremos en el uso de sockets de tipo STREAM que utilizan el protocolo **TCP**. Inicialmente la estructura socket debe ser creada e inicializada en ambos extremos de la comunicación con la función ***socket*** que nos devuelve un manejador de tipo entero (similar a un manejador de la E/S estándar).

En el lado del servidor es preciso adicionalmente nombrarlo (asignar IP y Puerto fijo) para que pueda ser accedido por otro proceso. El socket del cliente se nombra de forma automática cuando se establece la conexión. La función ***bind*** nos permite realizar esta tarea en el proceso servidor. Los parámetros son: el identificador del socket, la estructura de tipo **sockaddr** rellena con la dirección IP y el puerto, y por último el tamaño de dicha estructura (recordemos que este tamaño varia según el protocolo de red). La dirección IP se puede obtener automáticamente colocando la etiqueta **INADDR_ANY** en lugar de una dirección IP. El valor del puerto se puede asignar automáticamente si ponemos su valor a

0. En caso de que deseemos escoger un valor de puerto de forma manual debemos asegurarnos que está por encima del valor **IPPORT_RESERVED**. La función ***getsockname*** nos permite conocer la IP y el puerto asignados al socket.

En el protocolo TCP, antes de proceder a la transmisión de datos, es preciso establecer una ***conexión***. Para ello, por un lado el **servidor** debe quedarse a la escucha de peticiones de conexión. Esto se logra mediante las funciones ***listen*** y ***accept***. La primera establece el tamaño de la *cola de espera* e inica la admisión solicitudes de conexión. La segunda función (***accept***) es la que propiamente dicho realiza la aceptación de conexión dejando al proceso en espera (*escucha*). Al socket se le denomina de este modo ***Socket de Conexión***.

Cuando se establece una conexión la función ***accept*** devuelve el control creando un nuevo socket (***Socket de Dialogo***) ya '*nombrado'*, por el cual puede empezar la comunicación de datos. Adicionalmente devuelve la información del socket cliente que solicitó la conexión de forma que la transmisión pueda ser bidireccional.

Vemos por tanto que en este caso, en el lado servidor existen más de un socket: uno que acepta conexiones (el que creamos nosotros) y otro que crea el S.O por cada conexión que se establezca (*Socket de diálogo*). Si nuestro programa es multiThread podemos aceptar tantas conexiones simultáneas como hayamos especificado con la función ***listen***.

Por el lado del proceso **cliente**, este es el que tiene la responsabilidad de establecer la *conexión*. Para ello se utiliza la función ***connect*** que establece una conexión entre el socket que hemos creado y el especificado en la estructura sockaddr (dirección IP y puerto).

Una vez establecida la *conexión*, ambos procesos (threads) pueden comenzar la comunicación a través del *socket de diálogo*. En este caso, las funciones no precisan más información que el identificador del socket ya que el resto de datos están asociados a la conexión. Se pueden utilizar las funciones específicas ***recv***, ***send*** o las funciones genéricas de entrada salida estándar ***read / write***.

En todos los casos las funciones devuelven el número de bytes leídos o enviados, o en caso de error un –1. El campo **flags** nos permite controlar varios aspectos de la comunicación (normalmente se pone a **0**). En la ayuda del S.O. (`man`) podemos encontrar todas las opciones disponibles.

Una vez que nuestro proceso no necesita comunicarse a través de él (o va a terminar) debe liberar los recursos asociados cerrando el socket. Para ello se dispone de la función ***close***. Esta función no elimina los posibles datos que se puedan encontrar pendientes de transmisión en una conexión. Si queremos que se borren estos datos debemos ejecutar previamente la función ***shutdown***. Esta función permite también eliminar una conexión de forma parcial.

5 TAREAS

Cuestión 1. Comunicación unidireccional mediante sockets de tipo STREAM (TCP)

Para empezar de forma sencilla introduciremos un ejemplo de comunicación simple unidireccional para estudiar el uso de la librería de sockets. Se recomienda estudiar detalladamente todas las funciones y etapas descritas en el código fuente, haciendo especial hincapié a las variables utilizadas y parámetros de las funciones.

Cada alumno introducirá el código de ejemplo mostrado en el anexo, creando dos aplicaciones: **cliente** y **servidor**. Una vez introducido se pasará a comprobar el funcionamiento correcto de la comunicación.

Nota: El código fuente se puede descargar de la página web de la asignatura

Antes de compilar y probar el código del ejemplo debemos fijar algunos aspectos prácticos:

1. Cada proceso debe tener un puerto asignado. Este puede pasarse como parámetro al proceso servidor (debe asignarse un número de puerto no usado), o bien dejar que el S.O. lo asigne automáticamente.

2. El programa servidor muestra por pantalla su puerto y dirección IP y se queda a la espera de una transmisión.

3. Al programa cliente se le deben pasar como parámetros los datos del **servidor** (nombre simbólico, en este caso **lorca.umh.es**), el **puerto** del servidor (mostrado al ejecutar el proceso servidor) y opcionalmente el mensaje a transmitir **entrecomillado**:

   tcpclt1 ip_servidor puerto_servidor "mensaje"

4. Para la compilación se deben utilizar las siguientes librerías:

SUN-Solaris:

```
gcc tcpclt1.c –o tcpclt1 -lposix4 -lsocket -lnsl
gcc tcpsrv1.c –o tcpsrv1 -lposix4 -lsocket -lnsl
```

Linux:

```
gcc tcpclt1.c –o tcpclt1 -lrt -lnsl
gcc tcpsrv1.c –o tcpsrv1 -lrt -lnsl
```

Cuestión 2. Comunicación bidireccional mediante sockets de tipo STREAM

Modificar el código, tanto en el cliente como en el servidor, de forma que el servidor devuelva el mensaje recibido como un eco, y el cliente espere y lea la respuesta de eco a través de la conexión mostrándolo por consola.

tcpsrv2.c
tcpclt2.c

Cuestión 3. Comunicación bidireccional mediante sockets de tipo STREAM Multithread

Modificar el código del servidor TCP de forma que pueda atender varias conexiones de forma simultanea. Cada vez que reciba una conexión debe crear y lanzar un ***thread*** para atenderla y continuar la escucha (el identificador de thread se almacenará en el array **`hilos`**). Todos los threads tienen asignada la misma función (**`AtenderCliente`**) ya que todos los usuarios reciben el mismo servicio A cada thread se le pasará como parámetro una puntero a la estructura (**`DatosConexion`**) que contenga el socket de diálogo asignado a cada conexión, el contador de conexión y los datos del cliente.

El Thread **`main`**() deberá esperar a que terminen su ejecución cada una de los threads de diálogo antes de eliminar el socket de conexión.

Debido a que las funciones de la librería de sockets en Solaris son bloqueantes, los Threads deberán configurarse como **Threads de Núcleo**.

Deberán declararse los siguientes tipos de datos y variables:

```
void * AtenderCliente( void *);    // Thread de dialogo

pthread_t hilos[CONEXIONES];       // Tabla de threads

// Estructura paso de parámetros a los threads de Diálogo
typedef struct {
     int cont;             //número de conexión
     int socketDialogo;  // socket de dialogo
     struct sockaddr_in datosSocketClt;      // datos cliente
} DatosConexion;
```

Para compilar utilizar:

SUN-Solaris: gcc tcpsrv3.c –o tcpsrv3 -lposix4 -lsocket -lnsl -lpthread
Linux: gcc tcpsrv3.c –o tcpsrv3 -lrt -lnsl -lpthread

Cuestión 4. [OPTATIVA] Implementar un servidor y cliente de chat multiusuario.

Con base al programa anterior modificar el código implementado un servidor de ***chat*** multiusuario y multihilo. Deberá crearse un thread (**`LeerMensaje`**) por cada conexión que reciba los datos de cada usuario conectado (incluyendo su nombre). Existirá así mismo otro thread (**`EnviarMensaje`**) que remita a todos los usuarios conectados cada texto recibido en el servidor.

Para coordinar todos los threads deberemos disponer variables globales compartidas: un array (**`mensaje`**) con el texto que debe ser remitido a todos y un array (**`sockets`**) con los identificadores de sockets. Al tratarse de variables globales comunes deben ser accedidas en una sección crítica bloqueando un mutex (**`candado_mensaje`**), (**`candado_sockets`**).

```
#define CONEXIONES    10    // máximo numero de conexiones
#define DIM      1024       // Tamaño buffer mensajes

void * LeerMensaje( void *);           // Threads de escucha
void * EnviarMensaje( void *);   // Thread reenvio

char mensaje[DIM];          // buffer global de nuevo mensaje
int sockets[CONEXIONES];               // Tabla sockets dialogo

pthread_mutex_t candado_mensaje;
pthread_mutex_t candado_sockets;

pthread_t hilos[CONEXIONES+1];   // Tabla de threads
                                  // hilo[0] -> EnviarMEnsaje
```

Cada **thread de escucha** leerá los datos del cliente en un buffer interno. Intentará bloquear el acceso al array de **`mensaje`** con el mutex (**`candado_mensaje`**). Una vez accedida la sección crítica, si **`mensaje`** está vacío (==""), copiará la cadena leída y liberará el mutex. Si `mensaje` no está vacío liberará en mutex y lo intentará de nuevo. El Thread termina cuando el cliente envía la cadena "**`exit`**"

El **thread de reenvío**, intentará bloquear el mutex (**`candado_mensaje`**) y si la cadena no está vacía ('**\0**') enviará la cadena a todos los clientes. Para ello necesita conocer todos los sockets de diálogo creados que deberán estar almacenados en un array (**`sockets`**) (-1 significa socket no asignado o desconectado) protegida por el candado (**`candado_sockets`**). Una vez enviado borrará la cadena mensaje (**'\0'**) y liberará sendos mutex.

El **thread principal** únicamente acepta conexiones. Cada conexión aceptada devuelve un manejador de socket que debe ser almacenado en un array compartido (**`sockets`**) con el thread de reenvío. Para evitar conflictos usaremos también un mutex para su acceso (**`candado_sockets`**). Los valores del array se inicializaran a **-1** indicando que no hay ningún cliente. Si un cliente se desconecta su entrada en (**`sockets`**) se pondrá a **-1.**

El cliente transmitirá los mensajes introducidos por teclado hasta pulsar la cadena "**`exit`**" y mostrará los mensajes enviados por el servidor.