Laboratorio de
Sistemas Operativos

Tecnológico
de Monterrey

# Práctica 9: Networking: Cliente-Servidor.

## ÍNDICE



## Objetivos

Siguiendo con los elementos de programación en un ambiente operativo GNU/Linux, trabajarás en esta ocasión con los mecanismos utilizados por aplicaciones del tipo "**Cliente/Servidor**", utilizando protocolos de comunicación denominados **"AF_UNIX", y "TCP/IP" (AF_INET)**

## Conceptos

### Comunicación entre procesos

Hoy en día es fundamental que un proceso pueda comunicarse con otros (incluyendo procesos entre aplicaciones de usuarios y entre procesos del propio S.O.) A este tipo de comunicación se le denomina **IPC** por su escritura en inglés ("Inter-Process Communication").

Puede llevarse a cabo de diferentes formas, por ejemplo compartiendo un espacio específico de memoria RAM (donde se almacenan variables o buffers compartidos) y los accesos a esas áreas compartidas se realizan mediante las rutinas de IPC. Estas rutinas proveen los mecanismos necesarios para que la comunicación sea siempre sincronizada y exitosa (sin riesgo de una "*Race condition*" o problemas similares).


**Autor**: Dr. Juan A. Nolazco Flores **Co-autores**: M.C. Jorge Villaseñor, M.C. Roberto Aceves,
Ing.Raúl Fuentes, jose i. icaza, claudia.

Este tipo de comunicación se requería desde antes de que una computadora ocupará comunicarse con otra computadora y por lo mismo existe una familia de protocolos en los sistemas UNIX que son heredados en los sistemas operativos GNU/Linux. A esta familia se le denomina "**AF_UNIX**" y se utiliza para comunicar procesos que residen en una misma máquina.

En la modernidad el conjunto de protocolos para Internet denominado "**TCP/IP**"es probablemente la forma más popular para comunicarse con procesos residentes en varias computadoras, aunque también puede utilizarse para establecer comunicación entre procesos residentes en un mismo equipo.

Cuando un proceso Servidor central es accedido por varios procesos Cliente, la comunicación se denomina "cliente-servidor". El proceso cliente puede comunicarse con el proceso servidor utilizando la IP (Internet protocol address) de la máquina donde reside el proceso servidor, o bien su dirección lógica, típicamente (127.0.0.1). En este laboratorio nos concetraremos en este tipo de comunicación entre pocesos cliente y un proceso servidor.

Hoy en día los S.O.contemporáneos GNU/Linux ya reconocen versiones de IP denominadas versión 4 IP(v4) y versión 6 (IPv6), por lo tanto manejan dos familias de protocolos por separado denominadas: **AF_INET** y **AF_INET6** Un solo proceso servidor para utilizar ambas familias simultáneamente con diferentes clientes.

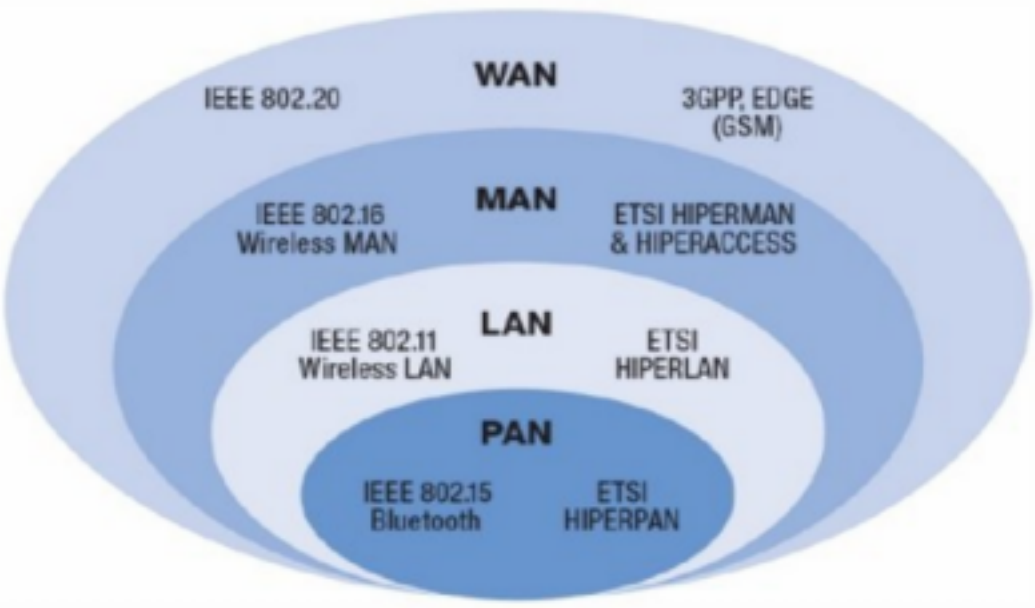
## Breve repaso sobre tipos de redes

**Una red de área personal (Personal Area Network or PAN)** es una red de computadoras utilizada para la comunicación entre los dispositivos electrónicos (teléfonos inteligentes, tabletas, computadoras) cercanos a una persona. Los dispositivos pueden no necesariamente pertenecer a la persona en cuestión.

El alcance de un PAN es típicamente algunos metros; rara vez se exceden 5 metros. Los PAN se pueden utilizar para la comunicación entre los mismos dispositivos personales (comunicación intrapersonal), o para que uno de los dispositivos de la red pueda conectarse a una red de más alto nivel y al Internet (un enlace ascendente), dotando de este acceso a otros dispositivos.Las redes de área personal puede estar conectadas con tecnologías tales como USB y FireWire.

Una red inalámbrica de área personal (WPAN) se puede también hacer posible con tecnologías de red tales como IrDA, Bluetooth, UWB, Z-Wave y ZigBee. La diferencia clave entre cada una de ellas es el consumo que exigen.

La siguiente figura indica algunos de los protocolos utilizados por Personal, Local, Metropolitan and Wide area networks:




## Sockets

En los orígenes de Internet, surgió la necesidad de intercomunicar procesos entre sí, por lo que la Universidad de Berkeley propuso la primera especificación e implementación de los ***sockets***.

El ***socket*** es una interfaz que se compone de una serie de llamadas a sistema y que permite que dos procesos (posiblemente situados en computadoras distintas) intercambien un flujo de datos de forma fiable y ordenada. El elemento clave del concepto esta en que los programas puedan comunicarse entre sí. Es necesario que un programa sea capaz de localizar al otro y que ambos puedan intercambiar cualquier secuencia de datos.

Cada *socket* requiere la definición de tres recursos:

- Un ***protocolo*** *de comunicación* que permita el intercambio de información
- Una ***dirección*** que conozca el protocolo de comunicación para identificar a los dispositivos comunicantes
- Un **número** de *puerto* que identifique a un proceso dentro de los dispositivos.

A continuación se muestran los encabezados y la llamada a sistema **socket(),** necesarios para la creación de un socket en C:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int af, int type, int protocol);
```

Esta llamada a sistema regresa un ***file descriptor (fd)***, mediante el cual se puede acceder a dicho socket. Recordemos que los fd's pueden utilizarse no únicamente para escribir y leer archivos, sino también para enviar y recibir datos entre procesos

**Autor**: Dr. Juan A. Nolazco Flores **Co-autores**: M.C. Jorge Villaseñor, M.C. Roberto Aceves,
Ing.Raúl Fuentes, jose i. icaza, claudia.

- El parámetro ***af*** **(address family)** especifica la familia de protocolos que se desea utilizar. Dos de estas familias, presentes en la mayoría de los sistemas, son:

→ **AF_UNIX** - protocolos internos Unix, para comunicar procesos que se ejecutan en la misma máquina,

→ **AF_INET** - protocolos Internet, para comunicar procesos en diferentes máquinas mediante protocolos de red, como TCP,otros: AF_ISO, AF_IPX, AF_APPLETALK, etc…

- El parámetro ***type*** indica la semántica de la comunicación; algunos de los valores que puede tomar son:

→ **SOCK_STREAM** - orientado a conexión, secuencial, confiable, 2 vías. Recordemos que con un protocolo "orientado a conexión" los datos viajan por la red en el mismo orden en que son enviados

→ **SOCK_DGRAM** - no orientado a conexión (datagrama)- los datos viajan en paquetes numerados que pueden viajar por distintas rutas antes de llegar al receptor, quien se encarga de ponerlos en orden

→ **SOCK_SEQPACKET** - no orientado a conexión, secuencial, confiable, para Xerox Network System

- El parámetro ***"protocol"*** indica el protocolo en particular que se va a utilizar; si se deja con valor de 0 el sistema se encarga de elegirlo.

# Arquitectura/modelo Cliente-Servidor.

Esta arquitectura consiste en un proceso (Cliente) que realiza peticiones a otro proceso (Servidor), y este último le responde. Esta idea se puede aplicar a tanto a procesos que se ejecutan en la misma computadora como a aquellos que se encuentran corriendo en diferentes computadoras conectadas por una red.

- Un ***servidor*** es un proceso que se está ejecutando en alguna computadora conectada a la red y que gestiona el acceso a un determinado recurso.
- Un ***cliente*** es un proceso que se ejecuta en la misma o diferente máquina y que realiza peticiones de algún recurso que gestiona el servidor.

El servidor está continuamente esperando peticiones de servicio. Cuando se produce una petición, el servidor despierta y atiende al cliente. Cuando el servicio concluye, el servidor vuelve al estado de espera.

De acuerdo con la forma de prestar el servicio, podemos considerar dos tipos de servidores:
→ ***Interactivos***, que atienden una sola petición de servicio a la vez; y
→ ***Concurrentes***, que toman cada petición y crean otros procesos para que se encarguen de atenderla, permitiendo atender varias peticiones a la vez.

El siguiente diagrama de flujo (Fig. 1) muestra la estructura de los procesos servidores y clientes.

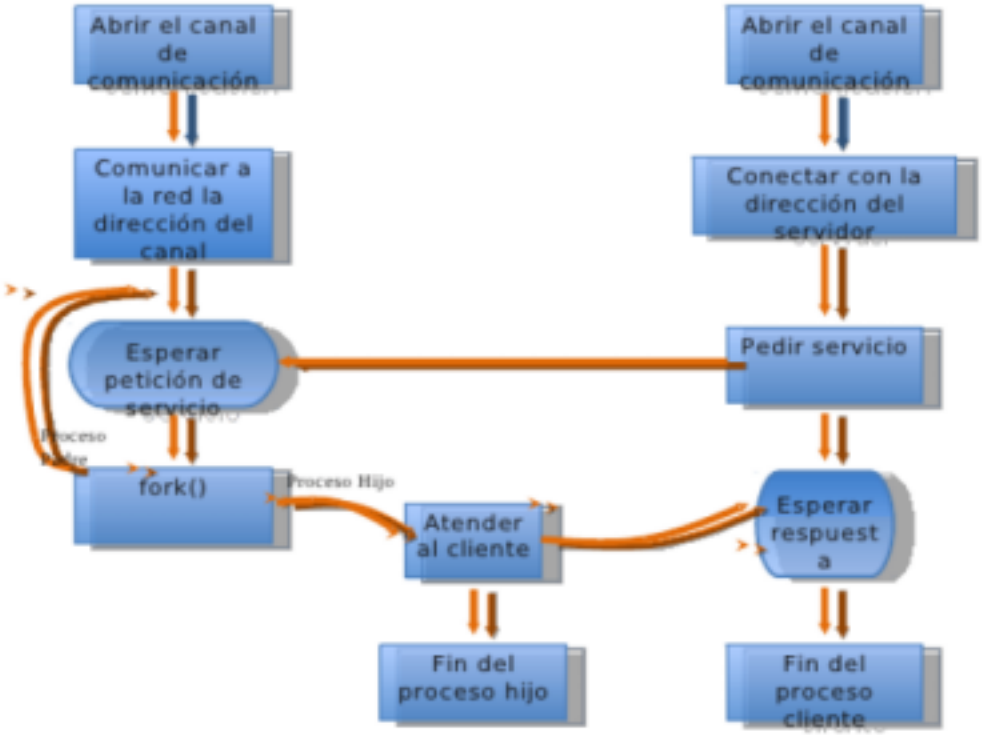



Fig. 1. Estructura de los programas servidores y clientes.

***NOTA**: El servidor y el cliente deben "entenderse"; esto significa que si el servidor espera que la comunicación se enviada de un cierto modo, el proceso cliente debe de hacerlo de esa forma. Esto es ajeno al lenguaje de programación con el cual se desarrollan los procesos. Tomen como ejemplo a los navegadores más populares: Chrome, Safari, Internet Explorer, Firefox. Cada uno fue hecho de forma diferente, pero todos saben que transacciones realizar con los servidores web para recibir los archivos y códigos html para desplegar ante el usuario la página web solicitada.*

Los servicios que intercambian el cliente y el servidor varían según el lenguaje de programación y el nivel dentro del Modelo OSI que estemos utilizando. Para esta práctica estaremos utilizando la familia más sencilla que deja los datos en nivel "Presentación" de OSI (nivel Aplicación de TCP/IP) y se le recomienda al alumno utilizar variables tipo "char" para no tener problemas de conversión de valores.

En este nivel prácticamente solo contamos con funciones de enviar (usualmente denominadas "write") y funciones de recibir (usualmente denominadas "read") que pueden recibir cualquier tipo de dato, y queda en responsabilidad del diseñador realizar la conversión o casting de dicha información.

## Cliente-Servidor Unix.

Para implementar un esquema de comunicación cliente-servidor entre procesos, se utilizan diversas llamadas al sistema, tales como write() o read(). Si te resultan familiares es debido a que en en Unix "everything is a file", asi que la comunicación entre dos computadoras distintas también se lleva a cabo por medio de un "File" que en realidad es un canal de comunicación

entre las dos procesos. En el siguiente diagrama (Fig. 2) puedes observar la secuencia de llamadas a sistema usadas para una comunicación orientada a conexión:

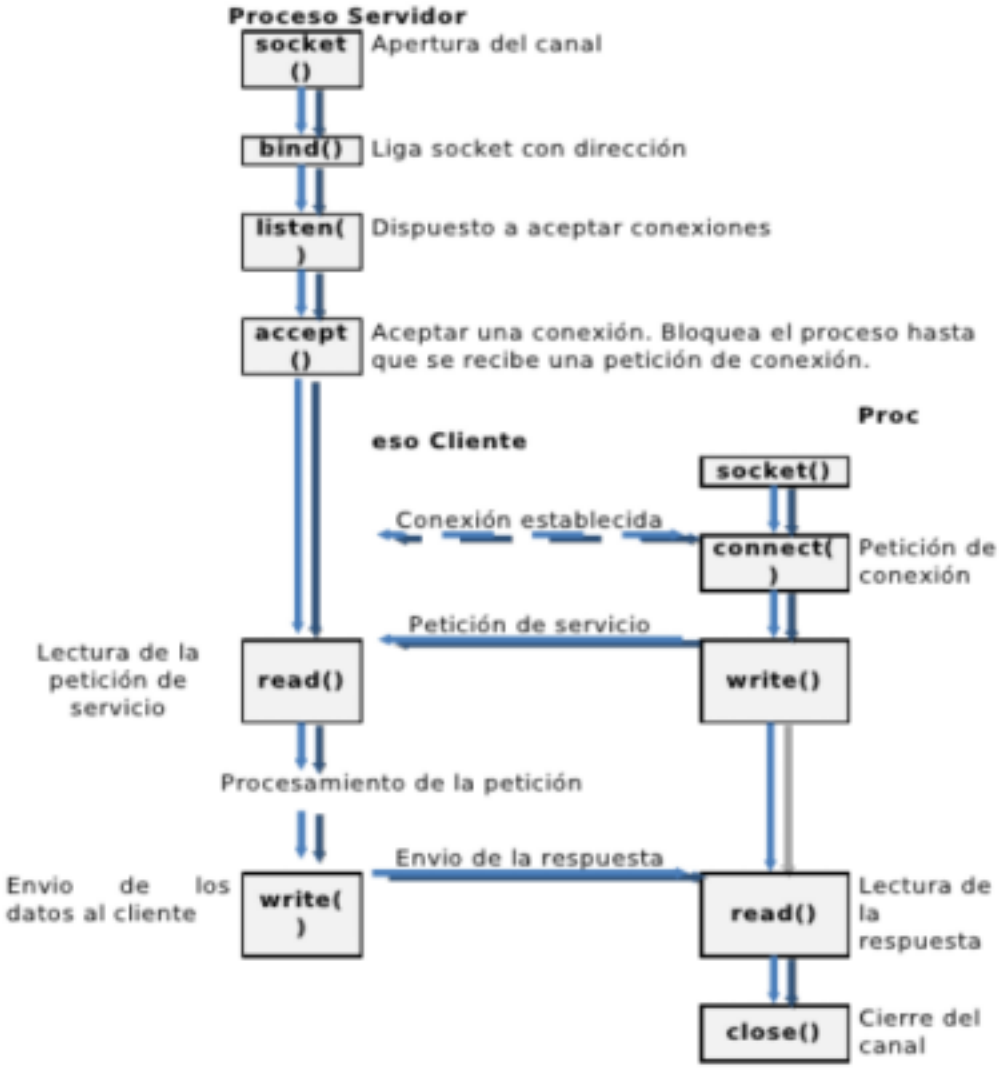



**Fig. 2**. Secuencia de llamadas para una comunicación orientada a conexión

La Figura 3 muestra la secuencia de llamadas para una comunicación no orientada a conexión.

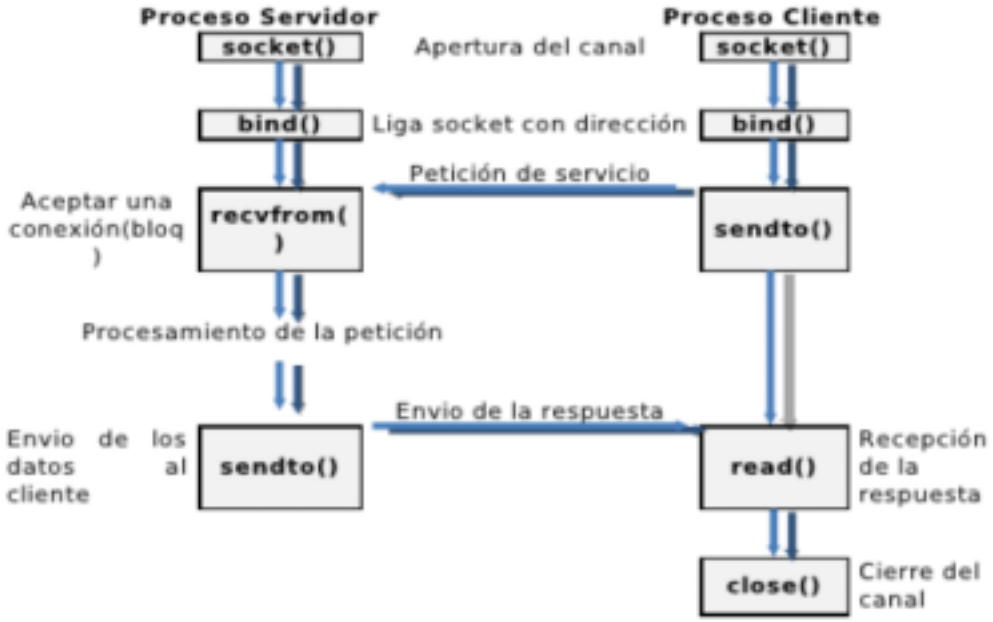



**Fig. 3**. Secuencia de llamadas para una comunicación **no** orientada a conexión.

El **Anexo A** contiene la definición de las llamadas a sistema utilizadas en el proceso de conexión por sockets, que son parte de la API definida por la biblioteca de sockets de Berkeley.

## Ejemplo de socket local (de la familia AF_UNIX)

Este ejemplo muestra la comunicación entre dos programas que corren en la misma máquina, por lo que no necesitan una conexión de red. Trabaja con un protocolo orientado a conexión (flujo de datos).

- El *Código 1* muestra el ***cliente***
- El *Código 2* el ***servidor.***

Observa las llamadas a sistema que se utilizan, en particular la creación del socket donde se especifica la familia de direcciones de protocolos del socket y su semántica: `socket(AF_UNIX, SOCK_STREAM, 0);`

```
/*
* client1.c - a simple local client program.
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

int main()
{
   int sockfd;
   int len;
   struct sockaddr_un address;
   int result;
```

```
   char ch = 'A';

/*  Create a socket for the client.  */
    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);

/*  Name the socket, as agreed with the server.  */
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "server_socket");
    len = sizeof(address);

/*  Now connect our socket to the server's socket.  */
    result = connect(sockfd, (struct sockaddr *)&address, len);


   if(result == -1) {
        perror("oops: client1");
        exit(1);
    }

/*  We can now read/write via sockfd.  */
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    return 0;
}
```

***Código 1.*** *Cliente local.*

```
/*
* server1.c - a simple local server program.
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

int main()
{
   int server_sockfd, client_sockfd;
   int server_len, client_len;
   struct sockaddr_un server_address;
   struct sockaddr_un client_address;

/*  Remove any old socket and create an unnamed socket for the server.  */
    unlink("server_socket");
    server_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);

/*  Name the socket.  */
    server_address.sun_family = AF_UNIX;
    strcpy(server_address.sun_path, "server_socket");
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);

/*  Create a connection queue and wait for clients.  */
    listen(server_sockfd, 5);
   while(1) {
     char ch;

        printf("server waiting\n");
```

**Autor**: Dr. Juan A. Nolazco Flores **Co-autores**: M.C. Jorge Villaseñor, M.C. Roberto Aceves,

Ing.Raúl Fuentes, jose i. icaza, claudia.

```
/*  Accept a connection.  */
        client_len = sizeof(client_address);
        client_sockfd = accept(server_sockfd,
            (struct sockaddr *)&client_address, &client_len);

/*  We can now read/write to client on client_sockfd.  */
        read(client_sockfd, &ch, 1);
        ch++;
        write(client_sockfd, &ch, 1);
        close(client_sockfd);
    }
}
```

***Código 2.*** *Servidor local***)**

**Autor**: Dr. Juan A. Nolazco Flores **Co-autores**: M.C. Jorge Villaseñor, M.C. Roberto Aceves,

Ing.Raúl Fuentes, jose i. icaza, claudia.

## ❖ Ejemplo de Socket Remoto (Familia AF_INET)

Este ejemplo muestra la comunicación entre dos programas que corren en distintas máquinas, ambas conectadas a una red. Trabaja con un protocolo orientado a conexión (flujo de datos).

- El *Código 3* muestra el ***cliente***
- El *Código 4* el ***servidor.***

Observa la creación del socket con la familia AF_INET y el uso de la estructura *sockaddr_in* para identificar al socket donde se definen la *familia de direcciones*, la *dirección* de red y el *puerto*.

```c
/*
* client2.c - a simple network client program.
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main()
{
   int sockfd;
   int len;
   struct sockaddr_in address;
   int result;
   char ch = 'A';

/*  Create a socket for the client.  */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

/*  Name the socket, as agreed with the server.  */
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("127.0.0.1");
    address.sin_port = 9734;
    len = sizeof(address);

/*  Now connect our socket to the server's socket.  */
    result = connect(sockfd, (struct sockaddr *)&address, len);


   if(result == -1) {
        perror("oops: client2");
        exit(1);
    }

/*  We can now read/write via sockfd.  */
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    return 0;
}
```

***Código 3.*** *Cliente remoto.*

**Autor**: Dr. Juan A. Nolazco Flores **Co-autores**: M.C. Jorge Villaseñor, M.C. Roberto Aceves,
Ing.Raúl Fuentes, jose i. icaza, claudia.

```c
/*
* server2.c - a simple network server program.
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main()
{
   int server_sockfd, client_sockfd;
   int server_len, client_len;
   struct sockaddr_in server_address;
   struct sockaddr_in client_address;

/*  Create an unnamed socket for the server.  */
    server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

/*  Name the socket.  */
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
    server_address.sin_port = 9734;
    server_len = sizeof(server_address);
    bind(server_sockfd, (struct sockaddr *)&server_address, server_len);

/*  Create a connection queue and wait for clients.  */
    listen(server_sockfd, 5);
   while(1) {
     char ch;

        printf("server waiting\n");

/*  Accept a connection.  */
        client_len = sizeof(client_address);
        client_sockfd = accept(server_sockfd,
            (struct sockaddr *)&client_address, &client_len);

/*  We can now read/write to client on client_sockfd.  */
        read(client_sockfd, &ch, 1);
        ch++;
        write(client_sockfd, &ch, 1);
        close(client_sockfd);
    }
}
```

***Código 4.*** *Servidor remoto.*

Para compilar cualquiera de estos códigos puedes utilizar la sintaxis que hemos manejado en otras prácticas:

```
gcc -o nombre_ejecutable nombre_codigo.c
```

## Anexo A

**Autor**: Dr. Juan A. Nolazco Flores

**Co-autores**: M.C. Jorge Villaseñor, M.C. Roberto Aceves,

Ing.Raúl Fuentes, jose i. icaza, claudia.

# Laboratorio de Sistemas Operativos

## Llamadas a sistema utilizadas en la arquitectura cliente-servidor

A continuación, una lista con las llamadas a sistema utilizadas en el proceso de conexión por sockets, que son parte de la API definida por la librería de sockets de Berkeley:

- **socket()** - creates a new socket of a certain socket type, identified by an integer number, and allocates system resources to it. Creates an endpoint for communication and returns a file descriptor for the socket. socket() takes three arguments:

➔ ***domain***, which specifies the protocol family of the created socket. For example: PF_INET for network protocol IPv4 or PF_INET6 for IPv6. PF_UNIX for local socket (using a file).

➔ ***type***, one of:
- SOCK_STREAM (reliable stream-oriented service or Stream Sockets),
- SOCK_DGRAM (datagram service or Datagram Sockets),
- SOCK_SEQPACKET (reliable sequenced packet service)
- SOCK_RAW (raw protocols atop the network layer).

➔ ***protocol***, specifying the actual transport protocol to use. The most common are IPPROTO_TCP, IPPROTO_SCTP, IPPROTO_UDP, IPPROTO_DCCP. These protocols are specified in <netinet/in.h>. The value "0" may be used to select a default protocol from the selected domain and type.

The function returns -1 if an error occurred. Otherwise, it returns an integer representing the newly-assigned descriptor. Prototype:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

- **bind()** - is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local port number and IP address. Assigns a socket an address. When a socket is created using socket(), it is only given a protocol family, but not assigned an address. This association with an address must be performed with the bind() system call before the socket can accept connections to other hosts. bind() takes three arguments:

➔ ***sockfd***, a descriptor representing the socket to perform the bind on

➔ ***my_addr***, a pointer to a sockaddr structure representing the address to bind to.

➔ ***addrlen***, a socklen_t field specifying the size of the sockaddr structure.

Bind() returns 0 on success and -1 if an error occurs. Prototype:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *my_addr,
socklen_t addrlen);
```

**Autor**: Dr. Juan A. Nolazco Flores **Co-autores**: M.C. Jorge Villaseñor, M.C. Roberto Aceves,
Ing.Raúl Fuentes, jose i. icaza, claudia.

- **listen()** - is used on the server side, and causes a bound TCP socket to enter listening state. After a socket has been associated with an address, listen() prepares it for incoming connections. However, this is only necessary for the stream-oriented (connection-oriented) data modes, i.e., for socket types (SOCK_STREAM, SOCK_SEQPACKET). listen() requires two arguments:

➔ ***sockfd***, a valid socket descriptor.

➔ ***backlog***, an integer representing the number of pending connections that can be queued up at any one time. The operating system usually places a cap on this value.

Once a connection is accepted, it is dequeued. On success, 0 is returned. If an error occurs, -1 is returned.

```
Prototype
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

- **accept()** - is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection. When an application is listening for stream-oriented connections from other hosts, it is notified of such events (cf. select() function) and must initialize the connection using the accept() function. Accept() creates a new socket for each connection and removes the connection from the listen queue. It takes the following arguments:

➔ ***sockfd***, the descriptor of the listening socket that has the connection queued.

➔ ***cliaddr***, a pointer to a sockaddr structure to receive the client's address information.

➔ ***addrlen***, a pointer to a socklen_t location that specifies the size of the client

➔ **address structure passed to accept().** When accept() returns, this location indicates how many bytes of the structure were actually used.

The accept() function returns the new socket descriptor for the accepted connection, or -1 if an error occurs. All further communication with the remote host now occurs via this new socket. Datagram sockets do not require processing by accept() since the receiver may immediately respond to the request using the listening socket. Prototype:

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t
*addrlen);
```

- **connect()** - is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection. The connect() system call connects a socket, identified by its file descriptor, to a remote host specified by that host's address in the argument list. Certain types of sockets are connectionless, most commonly user datagram protocol sockets. For these sockets, connect takes on a special meaning: the default target for sending and receiving data gets set to the given address, allowing the use of functions such as send() and recv() on connectionless sockets.

connect() returns an integer representing the error code: 0 represents success, while -1 represents an error. Prototype:

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr,
socklen_t addrlen);
```

- **gethostbyname()** and **gethostbyaddr()** - The gethostbyname() and gethostbyaddr() functions are used to resolve host names and addresses in the domain name system or the local hosts other resolver mechanisms (e.g., /etc/hosts lookup). They return a pointer to an object of type struct hostent, which describes an Internet Protocol host. The functions take the following arguments:

➔ ***name*** specifies the name of the host. For example: www.wikipedia.org

➔ ***addr*** specifies a pointer to a struct in_addr containing the address of the host.

➔ ***len*** specifies the length, in bytes, of addr.

➔ ***type*** specifies the address family type (e.g., AF_INET) of the host address.

The functions return a NULL pointer in case of error, in which case the external integer h_errno may be checked so see whether this is a temporary failure or an invalid or unknown host. Otherwise a valid struct hostent * is returned.These functions are not strictly a component of the BSD socket API, but are often used in conjunction with the API functions. Furthermore, these functions are now considered legacy interfaces for querying the domain name system. New functions that are completely protocol-agnostic have been defined.

These new function are getaddrinfo() and getnameinfo(), and are based on a new addrinfo data structure. Prototypes:

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const void *addr, int len, int
type);
```

- **send**() and **recv()**, or **write()** and **read()**, or **recvfrom()** and **sendto()**, - are used for sending and receiving data to/from a remote socket.
- **close**() - causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.

## = = = = Laboratorio = = = =

read ( sockaddr_in , char* char, int size )
write( sockaddr_in , char* char, int size )

**Autor**: Dr. Juan A. Nolazco Flores **Co-autores**: M.C. Jorge Villaseñor, M.C. Roberto Aceves,
Ing.Raúl Fuentes, jose i. icaza, claudia.

**char* char -** Recibe una cadena de caracteres; para que no batallen, utilicen un arreglo del tipo Char. La longitud del arreglo debe estar indicado también en "int size" (que es el tamaño del buffer de datos que se va a enviar). No olviden que tendrán que utilizar ampersand ( & ) en la función de read ya que el buffer leído se depositará en "char* char)

## Primera parte: Cliente-Servidor en Procesos (Familia UNIX)

### Ejercicio 1

Modifique el programa del cliente para que envíe cualquier carácter (un solo Char) al servidor. Dicho carácter debe ser dado por el usuario al invocar el programa (Como argumento de entrada)

***Nota:* Reportarás en el formulario el código fuente del cliente ya modificado.**

### Ejercicio 2

Compile los códigos de cliente y servidor local, ejecute el servidor en el fondo y ejecute un par de clientes en la misma terminal.

> ***Tip**: Si utiliza un comando como:*
> *$ **Servidor &&; Cliente [Argumento]; Cliente [Argumento]...***
> *estarás enviando el servidor a ejecutar en el background, seguido de invocaciones a los clientes. O bien simplemente **Servidor &&** para ejecutar el servidor en background y liberando la terminal para aceptar comandos adicionales*

***Nota:* Reportarás en el formulario el despliegue de la terminal del arranque del servidor y la ejecución de los clientes (Con las respectivas respuestas del servidor).**

### Ejercicio 3

Confirme el funcionamiento del servidor mediante Netstat

**Netstat** es una herramienta incluida en sistemas GNU/Linux la cual permite revisar qué recursos están separados para la comunicación entre procesos de AF_UNIX y para la comunicación con TCP/IP (AF_inet). Si se ejecuta el comando sin argumentos se recibirá una gran cantidad de información, por lo tanto tenemos que elaborar algo de filtrado.

Como el objetivo es validar que una aplicación servidor para la familia AF_UNIX está registrado y en estado de escucha (Listening), se espera que verifiques (man….) qué argumentos te permiten desplegar los servicios con esas características. Seguramente saldrán múltiples servicios, la mayoría internos del S.O. por lo que debes asegurar que la salida esté filtrada (grep?) y solo muestre el proceso que nos interesa. Para facilitar esto último puedes extraer el PROCESS-ID del servidor de antemano (comando $ ps con ciertos argumentos…) y utilizarlo en este punto.

***NOTA:* El alumno reportará en el formulario el comando introducido para validar la existencia del servidor y lo desplegado en la terminal.**

## Segunda parte: Cliente-Servidor (Familia INET)

Se cambiarán ambos programas para que el Servidor funcione como un Log. En general el servidor desplegará el siguiente mensaje cada vez que un cliente se comunique:

**Autor**: Dr. Juan A. Nolazco Flores **Co-autores**: M.C. Jorge Villaseñor, M.C. Roberto Aceves,
Ing.Raúl Fuentes, jose i. icaza, claudia.

<Dirección IP del cliente>: <Mensaje entregado por el cliente >

El mensaje tendrá una longitud máxima de 50 caracteres.
Una corrida ejemplo del servidor sería como la siguiente:

```
$ ./Servidor
10.17.112.90: Mensaje 1
10.17.112.70: Mensaje 2
10.17.112.80: Mensaje 3
```

La corrida ejemplo del cliente 10.17.112.90 sería:

```
$ ./Cliente  Mensaje1
Servidor:  Recibido
```

***NOTA**: El alumno puede decidir cómo pedir el mensaje a enviar y que desplegar en pantalla, pero siempre deberá desplegar la respuesta "Recibido" del servidor.*

**Entregas:**

- *Entregar el código fuente del **servidor** ya modificado.*
- *Entregar el código fuente del **cliente** ya modificado.*
- *Entregar un **ejemplo de corrida del servidor y dos o más clientes.***

***Tip**: El alumno puede utilizar la dirección lógica 127.0.0.1 para conectar al servidor de tal forma puede evitar la necesidad de usar otros equipos de computo.*

## ❖ Laboratorio: Modelo Ciente-Servidor.

not connection-oriented...


**Autor**: Dr. Juan A. Nolazco Flores **Co-autores**: M.C. Jorge Villaseñor, M.C. Roberto Aceves,

Ing.Raúl Fuentes, jose i. icaza, claudia.