



# INICIO GRABACIÓN




**SANJOSÉ**  
FUNDACIÓN DE EDUCACIÓN SUPERIOR

The background features a photograph of several hands stacked in a huddle, symbolizing teamwork. A large, semi-transparent blue circle is overlaid on the image, and a smaller, solid blue circle is positioned at the bottom right of the larger circle's edge.

# Procesos y comunicación




# Comunicación entre procesos



La comunicación entre procesos es un factor clave para construir sistemas distribuidos, los paradigmas de comunicación más usados en sistemas distribuidos son:

- Cliente - servidor.
- Llamada a un procedimiento remoto (RPC).
- Comunicación en grupo.

Los conceptos fundamentales que deben ser considerados para la comunicación son:

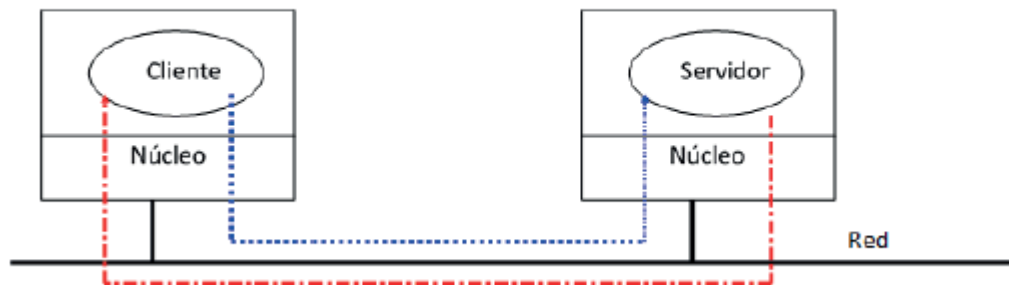
- Los datos tienen que ser aplanados antes de ser enviados.
  - Los datos tienen que ser representados de la misma manera en la fuente y destino.
  - Los datos tienen que empaquetarse para ser enviados.
  - Usar operaciones de *send* para enviar y *receive* para recibir.
  - Especificar la comunicación, ya sea en modo bloqueante o no bloqueante.
  - Abstracción del mecanismo de paso de mensaje.
  - La confiabilidad de la comunicación. Por ejemplo, usar TCP en lugar de UDP.
- 

# Modelo cliente - servidor



El término *cliente - servidor* (C-S) hace referencia a la comunicación en la que participan dos aplicaciones. Es decir que está basado en la comunicación de uno a uno. La aplicación que inicia la comunicación enviando una petición y esperando una respuesta se llama cliente. Mientras los servidores esperan pasivos, aceptan peticiones recibidas a través de la red, realizan el trabajo y regresan el resultado o un código de error porque no se generó la petición. Una máquina puede ejecutar un proceso o varios procesos cliente. La transferencia de mensaje en el modelo C-S se ejecuta en el núcleo. Una operación general del funcionamiento del *cliente - servidor*, donde un servidor espera una petición sobre un puerto bien conocido que ha sido reservado para cierto servicio. Un cliente reserva un puerto arbitrario y no usado para poder comunicarse.

Figura 4.2. Modelo cliente - servidor



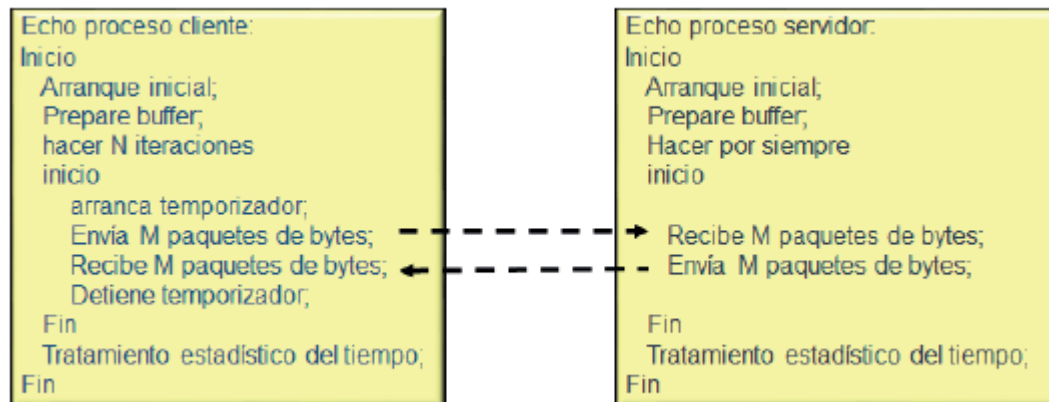
# Modelo cliente - servidor



La comunicación *cliente - servidor* puede ser implementada a través de librerías PVM (Parallel Virtual Machine) o sockets que permiten implementar comunicaciones bloqueantes y no bloqueantes, además de que también pueden usar TCP para tener comunicaciones confiables.

## Deficiencias del modelo cliente - servidor

- El paradigma de comunicación es la entrada/salida (E/S), ya que estos no son fundamentales en el cómputo centralizado.
- No permite la transparencia requerida para un ambiente distribuido.
- El programador debe de atender la transferencia de mensajes o las E/S, tanto del lado del cliente como del lado del servidor.



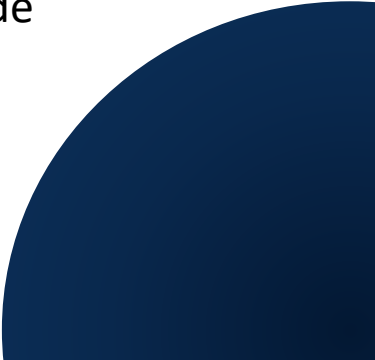


# Llamada de procedimiento remoto (RPC)



La llamada de procedimiento remoto, mejor conocido como RPC, es una variante del paradigma *cliente - servidor* y es una vía para implementar en la realidad este paradigma. En el RPC, un programa llama a un procedimiento localizado en otra máquina. El programador no se preocupa por las transferencias de mensajes o de las E/S. La idea de RPC es que una llamada a un procedimiento remoto se parezca lo más posible a una llamada local, esto le permite una mayor transparencia. Para obtener dicha transparencia, el RPC usa un *resguardo de cliente*, que se encarga de empacar los parámetros en un mensaje y le solicita al núcleo que envíe el mensaje al servidor, posteriormente se bloquea hasta que regrese la respuesta.

Algunos puntos problemáticos del RPC son:


- Que se deben de usar espacios de direcciones distintos, debido a que se ejecutan en computadoras diferentes.
  - Como las computadoras pueden no ser idénticas, la transferencia de parámetros y resultados puede complicarse.
  - Ambas computadoras pueden descomponerse.
- 

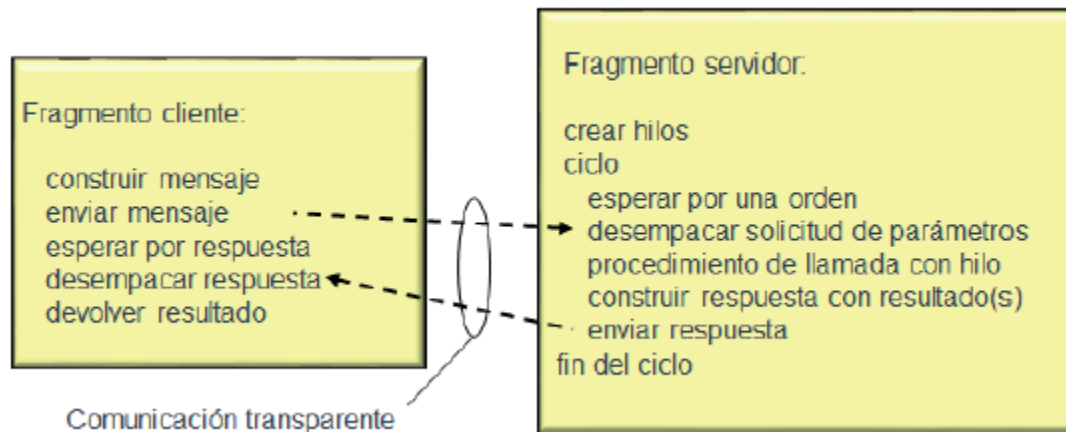
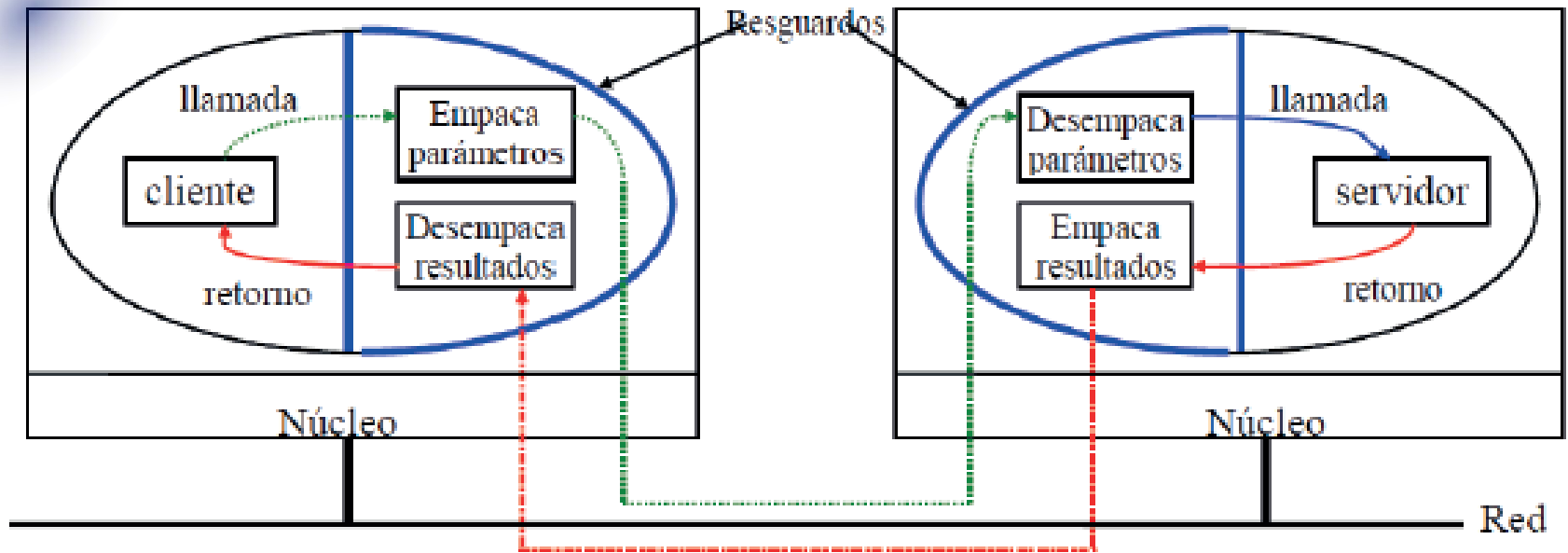


# Llamada de procedimiento remoto (RPC)



se muestra una llamada a un procedimiento remoto, el cual se realiza considerando los siguientes pasos:

1. El procedimiento cliente llama al resguardo del cliente de la manera usual.
  2. El resguardo del cliente construye un mensaje y realiza un señalamiento al núcleo.
  3. El núcleo envía el mensaje al núcleo remoto.
  4. El núcleo remoto proporciona el mensaje al resguardo del servidor.
  5. El resguardo del servidor desempaca los parámetros y llama al servidor.
  6. El servidor realiza el trabajo y regresa el resultado al resguardo.
  7. El resguardo del servidor empaca el resultado en un mensaje y hace un señalamiento mediante el núcleo.
  8. El núcleo remoto envía el mensaje al núcleo del cliente.
  9. El núcleo del cliente da el mensaje al resguardo del cliente.
  10. El resguardo desempaca el resultado y lo entrega al cliente.
- 





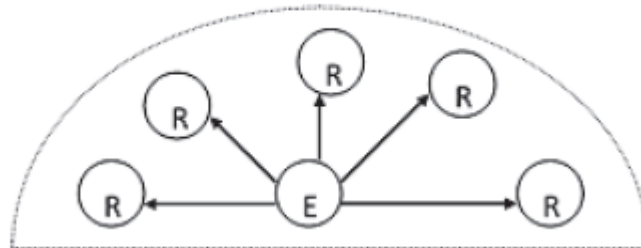
# Comunicación en grupo



En un sistema distribuido puede haber comunicación entre procesos de uno a muchos o de muchos a uno. Los grupos son dinámicos, lo cual implica que se pueden crear nuevos grupos y destruir grupos anteriores. Las técnicas para implantar la comunicación en grupos pueden ser:

- Transmisión de multidifusión (multicast)
- Transmisión completa
- Transmisión punto a punto

un emisor (E) y varios receptores (R)





# Comunicación en grupo



**Grupos cerrados:** Por ejemplo, el procesamiento paralelo.

**Grupos abiertos:** Por ejemplo, el soporte de servidores redundantes.

**Grupo de compañeros:** Permite que las decisiones se tomen de manera colectiva.

**Ventaja:** el grupo de compañeros es simétrico y no tiene punto de falla.

**Desventaja:** la toma de decisiones por votación es difícil, crea retraso y es costosa.

**Grupo jerárquico:** Existe un coordinador y varios trabajadores.

**Membresía de grupo:** Permite crear, eliminar grupos, agregar o eliminar procesos de grupos.

Utiliza técnicas como la del servidor del grupo o membresía distribuida.

Existen problemas de detección cuando un miembro ha fallado.





# Comunicación en grupo



*Direccionamiento al grupo:* Los grupos deben de poder direccionarse. Estas pueden ser:

- Dirección única grupal
- Dirección de cada miembro del grupo
- Direccionamiento de predicados

**Primitivas:** Las primitivas de comunicación son:

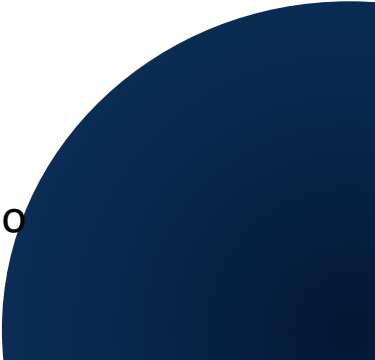
- send y receive, de la misma forma que en una comunicación puntual
- group\_send y group\_receive para comunicación en grupo

**Atomicidad:** Se refiere a que cuando se envía un mensaje a un grupo, este debe de llegarles a todos los miembros o a ninguno, así como garantizar la consistencia.

**Ordenamiento de mensajes:** Está conjuntada con la atomicidad y permite que la comunicación en grupo sea fácil de comprender y utilizar. Los criterios son:

- Ordenamiento con respecto al tiempo global
- Ordenamiento consistente
- Ordenamiento con respecto a grupos traslapados

**Escalabilidad:** Permite que el grupo continúe funcionando, aun cuando se agreguen nuevos miembros.



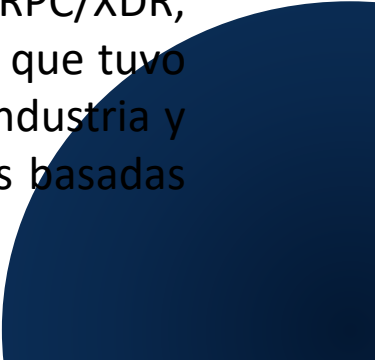


# Interfaz de programación de aplicaciones (API)



Las aplicaciones clientes - servidor se valen de protocolos de transporte para comunicarse. Las aplicaciones deben de especificar los detalles sobre el tipo de servicio (cliente o servidor), los datos a transmitir y el receptor donde situar los datos. La interfaz entre un programa de aplicación y los protocolos de comunicación de un sistema operativo se llama interfaz de programación de aplicaciones (API).

Un API define un conjunto de procedimientos que puede efectuar una aplicación al interactuar con el protocolo. Por lo general, un API tiene procedimientos para cada operación básica. Los programadores pueden escoger una gran variedad de APIs para sistemas distribuidos. Algunas de estas son BSD socket, Sun's RPC/XDR, librería de paso de mensajes PVM y Windows Sockets. La API de *socket*, que tuvo sus orígenes en el UNIX BSD, se convirtió en una norma de facto en la industria y muchos sistemas operativos la han adoptado, tanto para computadoras basadas en Windows (winsock) como para algunos sistemas UNIX.



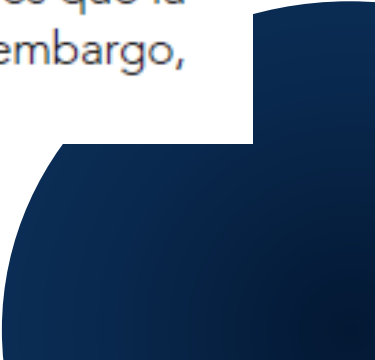


# La interfaz de socket



Un socket es un punto de referencia hacia donde los mensajes pueden ser enviados, o de donde pueden ser recibidos. Al llamar la aplicación a un procedimiento de socket, el control pasa a una rutina de la biblioteca de sockets que realiza las llamadas al sistema operativo para implementar la función de socket. UNIX BSD y los sistemas derivados contienen una biblioteca de sockets, la cual puede ofrecer a las aplicaciones una API de socket en un sistema de cómputo que no ofrece sockets originales.

Los sockets emplean varios conceptos de otras partes del sistema, pero en particular están integrados con la E/S, por lo que una aplicación se comunica con un socket de la misma forma en que se transfiere un archivo. Cuando una aplicación crea un socket, recibe un descriptor para hacer referencia a este. Si un sistema usa el mismo espacio de descriptor para los sockets y otras E/S, es posible emplear la misma aplicación para transferir datos localmente como para su uso en red. La ventaja del método de socket es que la mayor parte de las funciones tienen tres o menos parámetros, sin embargo, se debe llamar a varias funciones a este método.





# Funciones de la API de sockets



Las funciones relacionadas a la API sockets que permiten establecer una comunicación entre dos computadoras son:



`socket( )`

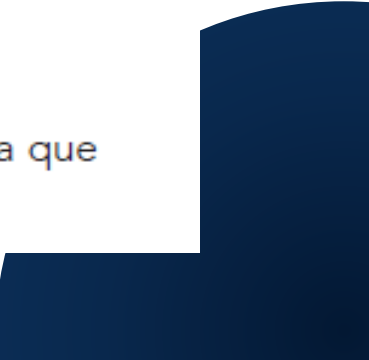
Esta rutina se usa para crear un socket y regresa un descriptor correspondiente a este socket. Este descriptor es usado en el lado del cliente y en el lado del servidor de su aplicación. Desde el punto de vista de la aplicación, el descriptor de archivo es el final de un canal de comunicación. La rutina retorna -1 si ocurre un error.

`close( )`

Indica al sistema que el uso de un socket debe de ser finalizado. Si se usa un protocolo TCP (orientado a conexión), `close` termina la conexión antes de cerrarlo. Cuando el socket se cierra, se libera al descriptor, por lo que la aplicación ya no transmite datos y el protocolo de transportación ya no acepta mensajes de entradas para el socket.

`bind( )`

Suministra un número a una dirección local a asociar con el socket, ya que cuando un socket es creado no cuenta con dirección alguna.





### `listen( )`

Esta rutina prepara un socket para aceptar conexiones y solo puede ser usada en sockets que utilizan un canal de comunicación virtual. Esta rutina se deberá usar del lado del servidor de la aplicación antes de que se pueda aceptar alguna solicitud de conexión del lado del cliente. El servidor encola las solicitudes de los clientes conforme estas llegan. La cola de solicitudes permite que el sistema detenga las solicitudes nuevas mientras que el servidor se encarga de las actuales.

### `accept( )`

Esta rutina es usada del lado del servidor de la aplicación para permitir aceptar las conexiones de los programas cliente. Después de configurar una cola de datos, el servidor llama `accept`, cesa su actividad y espera una solicitud de conexión de un programa cliente. Esta rutina solo es válida en proveedores de transporte de circuito virtual. Cuando llega una solicitud al socket especificado `accept( )` llena la estructura de la dirección, con la dirección del cliente que solicita la conexión y establece la longitud de la dirección, `accept( )` crea un socket nuevo para la conexión y regresa su descriptor al que lo invoca, este nuevo socket es usado por el servidor para comunicarse con el cliente y al terminar se cierra. El socket original es usado por el servidor para aceptar la siguiente conexión del cliente.





### `connect( )`

Esta rutina permite establecer una conexión a otro socket. Se utiliza del lado del cliente de la aplicación permitiendo que un protocolo TCP inicie una conexión en la capa de transporte para el servidor especificado. Cuando se utiliza para protocolos sin conexión, esta rutina registra la dirección del servidor en el socket, esto permite que el cliente transmita varios mensajes al mismo servidor. Usualmente el lado cliente de la aplicación enlaza a una dirección antes de usar esta rutina, sin embargo, esto no es requerido.

### `send( )`

Esta rutina es utilizada para enviar datos sobre un canal de comunicación tanto del lado del cliente como del lado servidor de la aplicación. Se usa para sockets orientados a conexión, sin embargo, podría utilizarse para datagramas pero haciendo uso de `connect( )` para establecer la dirección del socket.

### `sendto( )`

Permite que el cliente o servidor transmita mensajes usando un socket sin conexión (usando datagramas). Es exactamente similar a `send( )` solo que se deberán especificar la dirección destino del socket al cual se quiere enviar el dato. Se puede usar en sockets orientados a conexión pero el sistema ignorará la dirección destino indicada en `sendto( )`.





`recv( )`

Esta rutina lee datos desde un socket conectado y es usado tanto en el lado del cliente como del lado del servidor de la aplicación.

`recvfrom( )`

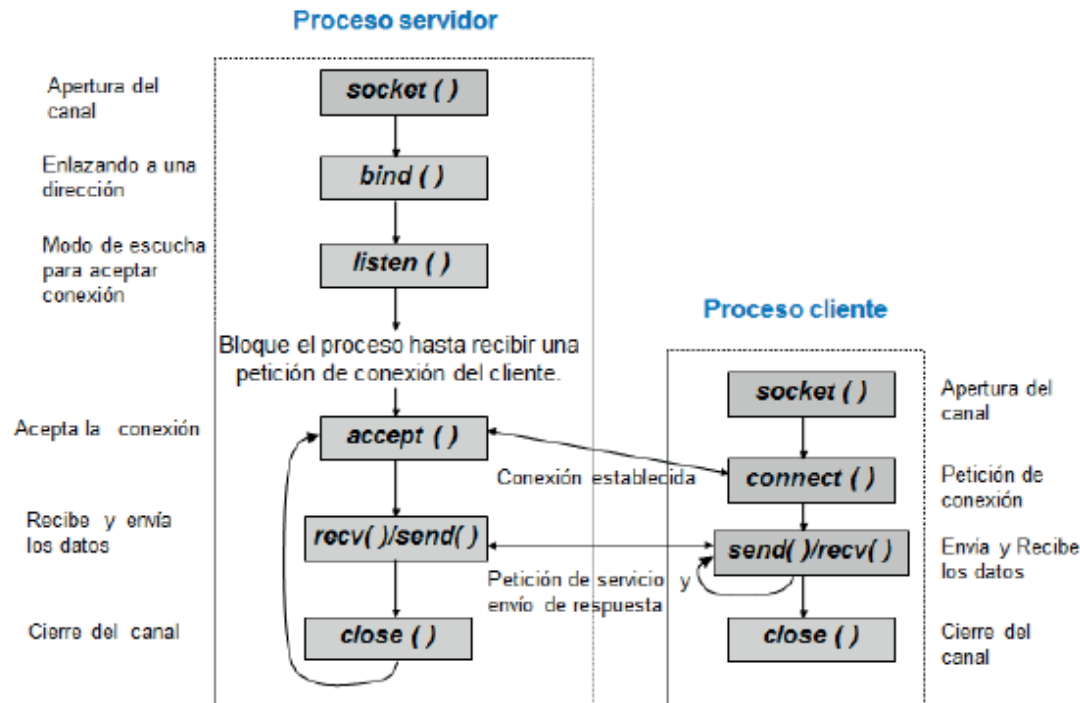
Esta rutina lee datos desde un socket sin conexión. En este caso, el sistema regresa la dirección del transmisor con los mensajes de entrada y permite registrar la dirección del socket transmisor en la misma forma que espera `sendto( )`, por lo que la aplicación usa la dirección registrada como destino de la respuesta.



# Comunicación orientada a conexión (TCP)



Un servicio orientado a conexión requiere que dos aplicaciones establezcan una conexión de transportación antes de comenzar el envío de datos. Para establecer la comunicación, ambas aplicaciones primero interactúan local-mente con protocolo de transporte y después estos protocolos intercambian mensajes por la red. Una vez que ambos extremos están de acuerdo y se haya establecido la conexión, las aplicaciones podrán enviar datos. La secuencia de llamadas para un escenario orientado a conexión.






# Comunicación orientada a conexión (TCP)



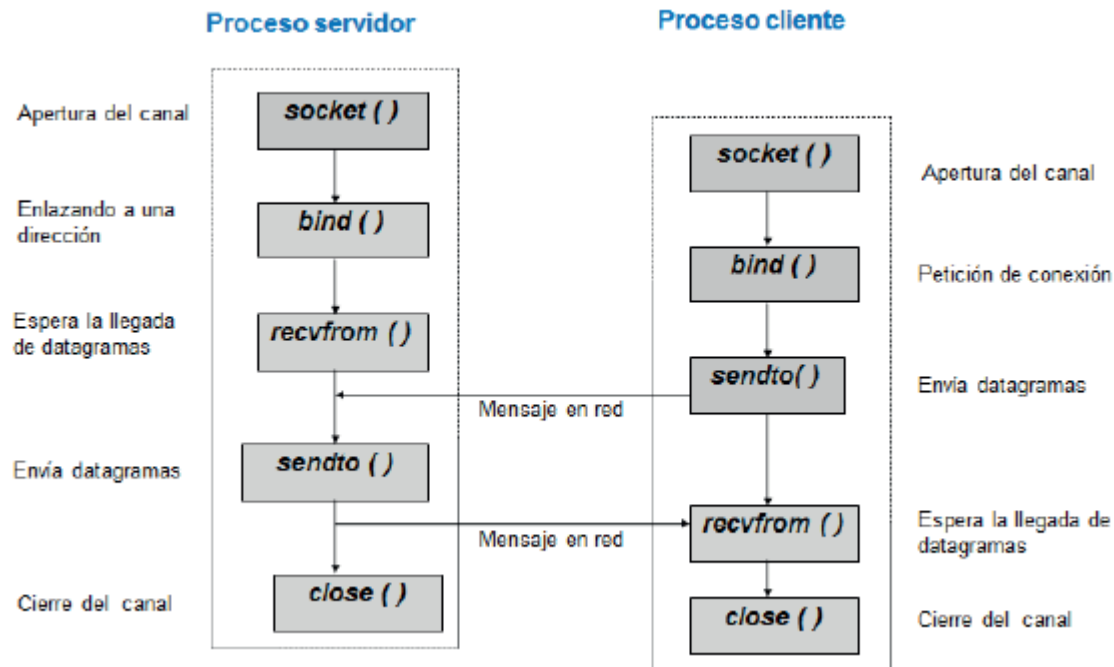
se observa que, después de llamar a *listen()*, el servidor se pone en modo pasivo hasta que recibe una solicitud del cliente. Cuando la solicitud de servicio del cliente llega al socket monitoreado por la función *accept()*, se crea de modo automático un socket nuevo, que será conectado inmediatamente con el proceso cliente. Este socket se llama de servicio y se cierra al terminar la transferencia de datos. El socket original que recibió la solicitud de servicio permanece abierto en estado de escucha para seguir aceptando nuevas solicitudes.





# Comunicación sin conexión o servicio de datagrama

Es similar a enviar una carta postal. El lado cliente actúa como la persona que envía la carta y el lado servidor como la persona que lo recibe. El servidor usa las llamadas a *socket()* y *bind()* para crear y unir un socket. Como el socket es sin conexión, se deberán usar las llamadas a *recvfrom()* y *sendto()*. Se llama a la función *bind()* pero no a *connect()*, dejando esta última como opcional. La dirección destino se especifica en la función *sendto()*. La función *recvfrom()* no espera conexión sino que responde a cualquier dato que llegue al puerto que tiene enlazado.





FUNDACIÓN DE EDUCACIÓN SUPERIOR

**SAN JOSÉ**

INSTITUCIÓN TECNOLÓGICA

FIN DE  
GRABACIÓN