UNIDAD 3. PROGRAMACIÓN DE COMUNICACIONES EN RED.

PROGRAMACIÓN DISTRIBUÍDA.

PROGRAMACIÓN DISTRIBUÍDA. Múltiples ordenadores colaboran entre sí comunicándose a través de una red.

MODELO DE PROGRAMACIÓN DISTRIBUÍDA O COMPUTACIÓN DISTRIBUÍDA:

Las características de un **sistema distribuido** son:

1.- El sistema está formado por **más de un elemento computacional**. Los elementos computacionales son independientes entre sí, ninguno de estos elementos comparte memoria con el resto. Un elemento computacional puede ser un ordenador dentro de una red, un procesador dentro de una máquina, una aplicación funcionando a través de internet.

2.- Los elementos computacionales no están sincronizados entre sí.

3.- Los elementos computacionales del sistema distribuido están conectados a una red. Se comunican entre ellos.

Es decir, elementos independientes, no comparten memoria interna, no están sincronizados, cada uno tiene su propio estado, contador de programa, a través de la red se comunican entre sí para realizar una tarea u objetivo común.

CONCEPTOS FUNDAMENTALES EN UN PROCESO DE COMUNICACIÓN

1.-**Mensaje:** Información que se intercambia entre las aplicaciones que se comunican.

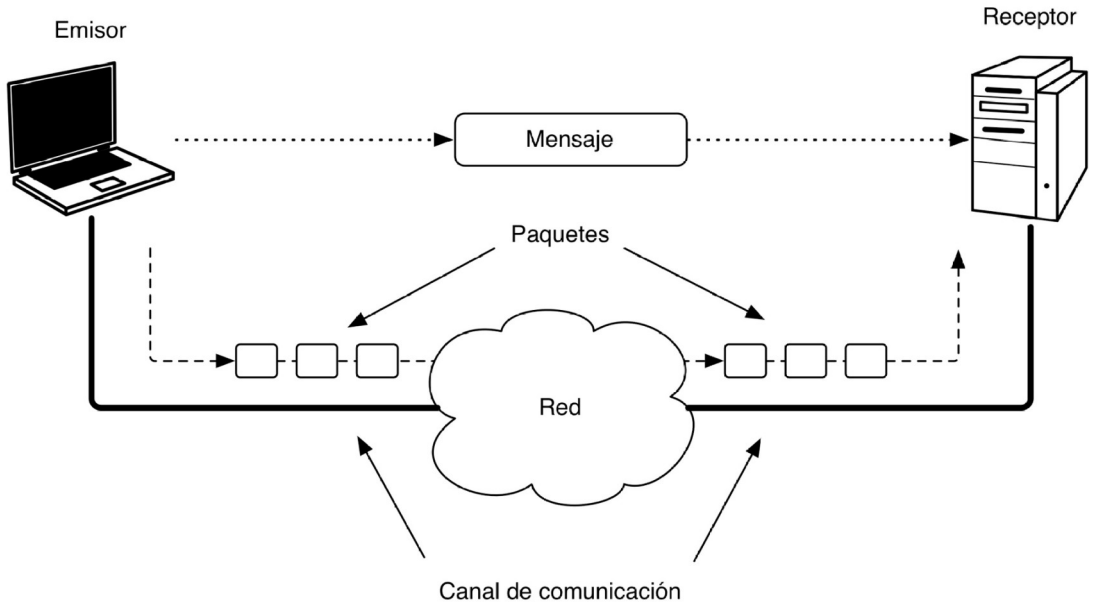
2.- **Emisor**: Aplicación que envía el mensaje.

3.- **Receptor**: Aplicación que recibe el mensaje. Dependiendo del modelo de comunicación (cliente/servidor, comunicación en grupo,), un mensaje puede tener más de un receptor.

4.- **Paquete**: El mensaje, para transmitirse a través de la red, se divide en uno o más paquetes. Definimos paquete como la unidad básica de información que se intercambia entre dos dispositivos.

5.- **Canal de comunicación**: Es el medio por el que se transmiten los paquetes, el canal de comunicación conecta al emisor con el/los receptores.

6.- **Protocolo de comunicaciones**: Conjunto de reglas que determinan o definen cómo se intercambian los paquetes. Un protocolo de comunicaciones define, por un lado, la secuencia de paquetes y, por otro, el formato de los mensajes.



Tanto los **elementos hardware** (interfaces de red, routers o encaminadores, etc.) como los **elementos software** (bibliotecas de programación, componentes del sistema operativo, etc.) que son necesarios para que las aplicaciones se comuniquen, **se organizan en una jerarquía o pila de protocolos.**

La pila de protocolo, contiene todos los elementos hardware y software necesarios para la comunicación y establece como trabajan juntos. La pila de protocolo utilizada en la mayoría de los sistemas distribuidos es la **pila IP.**

PILA DE PROTOCOLOS IP (**I**nternet **P**rotocol)

El protocolo IP es el protocolo estándar de comunicaciones en internet. Define los siguientes niveles:

1.- Nivel de red: Es el nivel más bajo, compuesto por los **elementos hardware**. Se encarga de transmitir los paquetes de información. El nivel de red puede estar construido sobre una red de comunicaciones de área local **LAN** (**L**ocal **A**rea **N**etwork) o una red de área extendida **WAN** (Wide **A**rea **N**etwork), o una red combinación de ambas.

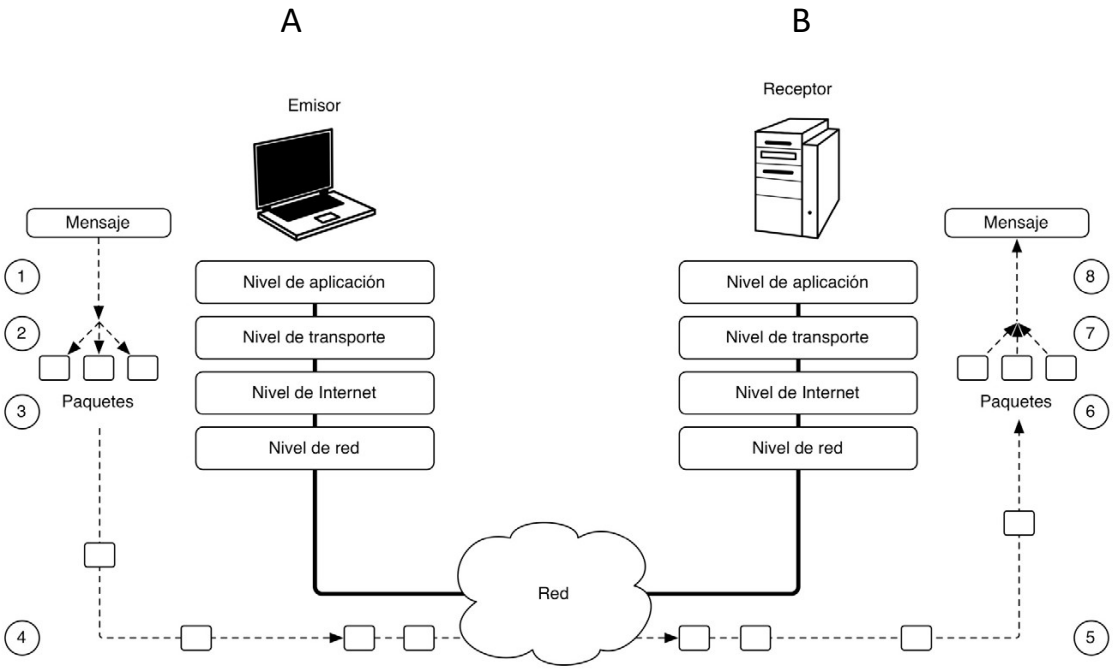
2.- Nivel de Internet: Por encima del nivel de red, compuesto por los elementos software que se encargan de dirigir los paquetes por la red.

3.- Nivel de Transporte: Por encima del nivel de internet, compuesto por los elementos software encargados de crear el canal de comunicación, descomponer el mensaje en paquetes, gestionar su transmisión entre el emisor y el receptor. Existen dos tipos de protocolos de transporte:

Protocolo TCP (**orientado a conexión**)

Protocolo UDP (**No orientado a conexión**)

4.- Nivel de Aplicación: El último nivel en la cima de la pila IP, compuesto por las aplicaciones que forman el sistema distribuido.



A1.- La aplicación que envía el mensaje (emisor, situada en el nivel de aplicación) entrega el mensaje al nivel inferior, al nivel de transporte.

A2.- El protocolo del nivel de transporte (TCP o UDP) descompone el mensaje en uno o varios paquetes y los pasa al nivel inferior, nivel de internet.

A3.- El nivel de Internet localiza al receptor del mensaje y **calcula la ruta** que deben seguir los paquetes. (Ver mecanismos de encaminamiento (enrrutamiento) del protocolo IP en el nivel de Internet). A continuación, entrega los paquetes al nivel inferior, nivel de red.

A4.- El nivel de red transmite los paquetes.

B1.- El nivel de red recibe los paquetes en el receptor y los pasa al nivel superior, nivel de internet.

B2.-El nivel de internet verifica que los paquetes han llegado al receptor correcto y los pasa al nivel superior.

B3.- El nivel de transporte (TCP o UDP) agrupa los paquetes para formar el mensaje. El mensaje reconstruido es enviado al nivel superior, nivel de aplicación.

B4.- La aplicación (receptor) recibe el mensaje.

NIVEL DE TRANSPORTE: PROTOCOLO TCP/PROTOCOLO UDP

1.- PROTOCOLO DE TRANSPORTE TCP

Es el más utilizado en la pila IP, tanto que comúnmente la pila IP se denomina **pila TCP/IP.** Se encarga de dividir los mensajes que ha recibido del nivel de aplicación, creando paquetes y enviarlos al nivel de internet / se encarga de combinar los paquetes recibidos del nivel de internet para formar el mensaje que pasa al nivel de aplicación.

Las características del protocolo de transporte TCP son:

1. Garantiza que los datos no se pierden.
2. Garantiza que los mensajes llegarán en orden. Es decir, no sólo todos los mensajes que se envían llegarán, sino que además los mensajes llegarán en el orden en que fueron enviados.
3. **Es un protocolo orientado a conexión**: Un protocolo orientado a conexión se caracteriza porque el canal de comunicación entre dos aplicaciones permanece abierto durante un cierto tiempo, esto supone que se podrán enviar varios mensajes utilizando el mismo canal. Cuando entre dos aplicaciones se transmite información empleado un protocolo de transporte orientado a conexión, se realizan las siguientes operaciones:
    a. **Establecimiento de la conexión**, se realiza siempre al inicio de la comunicación, sirve para crear el canal de comunicación, este **permanecerá abierto hasta que uno de las dos aplicaciones lo cierre.**
    b. **Envío de mensaje,** esto se puede realizar tantas veces como se desee, el canal de comunicación permanece abierto y se reutiliza para los distintos mensajes que se envían.
    c. **Cierre de conexión,** se realiza cuando se desea terminar la comunicación. Una vez cerrado el canal, para poder volver a reaunar la comunicación habrá establecer la conexión de nuevo.

2.- PROTOCOLO DE TRANSPORTE UDP

Las características del protocolo de transporte UDP son:

1.- Es un protocolo **no orientado a conexión**. Por lo que es más rápido que TCP ya que no es necesario establecer conexiones. **(Un protocolo de transporte no orientado a conexión es aquel en el que el canal de comunicación se crea independientemente para cada mensaje, no hay establecimiento de conexión ni cierre de conexión)**

2.- No garantiza que los mensajes lleguen siempre.

3.- No garantiza que los mensajes lleguen en el mismo orden en que fueron enviados.

4.- Los mensajes a enviar serán de 64KB como máximo

5.- Los mensajes en UDP se llaman **DATAGRAMAS.**

El protocolo de transporte UDP es menos fiable que TCP, sólo permite enviar mensajes de hasta 64KB, sin embargo, es más ligero y eficiente que TCP, también se utiliza mucho en computación distribuida.

SOCKETS

Los sockets proporcionan un **mecanismo de abstracción de la pila de protocolos**, de modo que ofrecen una i**nterfaz** de programación sencilla para que las diferentes **aplicaciones de un sistema distribuido** puedan intercambiar mensajes.

Un socket (" enchufe") representa **el extremo** de un canal de comunicación establecido entre un emisor y un receptor.

Ambas aplicaciones deben crear sus respectivos sockets (cada una su extremo del canal de comunicación) y conectarlos entre sí. Una vez conectados, entre ambos sockets se crea una **tubería privada,** a través de la red, que permiten que las aplicaciones en los extremos envíen y reciban mensajes por ella.

Para enviar mensajes las aplicaciones **escriben en su socket** y para recibir mensajes las aplicaciones **leen de su socket.**

DIRECCIONES y PUERTOS

Las aplicaciones tienen que conectarse entre sí, por tanto, deben localizarse dentro de la red de comunicaciones.

Las distintas máquinas conectadas en una red de comunicaciones que usa la pila IP, se distinguen por su dirección IP.

Una dirección IP es un número entero que identifica unívocamente a cada máquina en la red. Existen dos versiones del protocolo IP: IPv4 (IP versión 4) e IPv6 (IP versión 6). IPv4 es la versión que se usa en internet y en la mayoría de redes de área local, IPv6 es una versión posterior pensada para sustituir a IPv4 en un futuro.

Dirección IP fija: Inicialmente a cada máquina conectada a internet se le asignaba una dirección IP fija.

Dirección IP dinámica: Las máquinas reciben una dirección IP distinta cada vez que se conectan, esto permite compartir la misma dirección entre varias máquinas, siempre y cuando no estén conectadas a la vez. Esto soluciona en parte el problema de la cantidad de direcciones IP disponibles, porque la misma dirección IP se reutiliza. Además, La versión IPv6 dispone de un rango de direcciones muchísimo más amplio que IPv4, lo que permite muchas más máquinas conectadas simultáneamente a la red.

Dirección IP en IPv4 está formada secuencias de 32 bits, llamadas palabras. Cada palabra está divida en 4 grupos de 8 bits, llamados octetos. En cada octeto podemos representar números comprendidos entre 0 (00000000 en binario) y 255(11111111 en binario). Escribimos una dirección IP separando cada octeto por un punto y su valor en decimal para que sea más cómodo.

TIPOS DE SOCKETS: SOCCKETS STREAM Y SOCKETS DATAGRAM

SOCKTES STREAM. SOCKETS TCP

Los sockets stream son orientados a conexión, cuando se utilizan sobre la pila de protocolos IP, hacen uso del protocolo de transporte TCP.

Un socket stream se utiliza para comunicarse **siempre con el mismo receptor**, manteniendo el canal de comunicación abierto entre ambas aplicaciones hasta que termine la conexión.

La comunicación entre las aplicaciones se realiza por medio del protocolo TCP, por tanto es una conexión fiable en la que se garantiza la entrega de los paquetes de datos y el orden en que fueron enviados. TCP utiliza un esquema de acuse de recibo de los mensajes, de tal forma que si el emisor no recibe dicho acuse dentro de un tiempo determinado, vuelve a transmitir el mensaje.

Los procesos que se van a comunicar deben establecer antes una conexión mediante un stream. Un stream es una secuencia ordenada de unidades de información(bytes, caracteres,etc.) que pueden fluir en dos direcciones. Están diseñados para acceder a los datos de forma secuencial.

Uno de los elementos o aplicaciones de la comunicación debe ejercer el **papel de proceso servidor** y, la otra aplicación ejercerá el **papel de proceso cliente**.

El proceso servidor crea el socket en primer lugar y se pone a la espera a que el cliente se conecte. Cuando el proceso cliente desea iniciar la comunicación crea su socket y lo conecta al servidor, creando el canal de comunicación. En cualquier momento, cualquiera de los dos procesos puede cerrar su socket, destruyendo el canal de comunicación y terminando la comunicación.

Secuencia de operaciones desde el proceso cliente:

1º. Creación del socket. Se crea un socket y se le asigna un puerto. El número de puerto concreto que usa el socket cliente no es importante, por lo que se suele dejar al sistema operativo que lo asigne automáticamente (el primero libre). Este socket se llama **socket cliente.**

2º. Conexión (operación connect). En este paso se localiza el socket del proceso servidor, y se crea el canal de comunicación que une ambos procesos. Para ello el proceso cliente tiene que conocer la dirección IP del proceso servidor y el puerto por el que está escuchando.

3º. Envío y Recepción de mensajes. Ahora el proceso cliente puede enviar/recibir mensajes, mediante operaciones de escritura /lectura sobre el socket.

4º.- **Cierre de la conexión (operación close)**. Si desea terminar la comunicación el proceso cliente puede cerrar su socket.

Secuencia de operaciones desde el proceso servidor:

1º. Creación del socket. Se crea un socket. A este socket se le llama **socket servidor.**

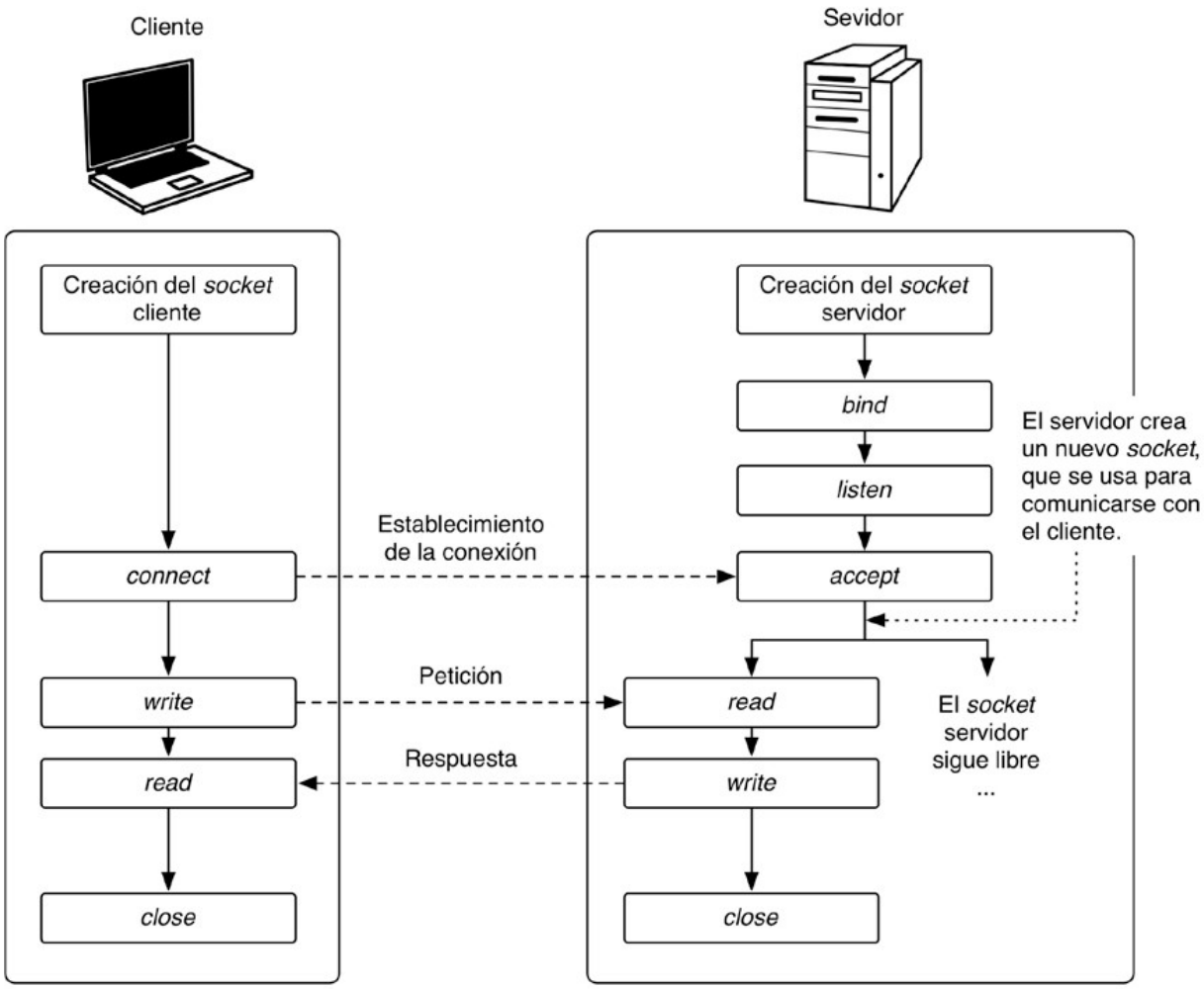
2º. Asignación de dirección y puerto (operación bind). En este caso es importante que la dirección IP y el número de puerto del socket servidor estén claramente especificados, sino el proceso cliente no será capaz de localizar al proceso servidor. La operación bind asigna una dirección IP y un número de puerto concreto al socket, lógicamente la dirección IP del socket debe ser la de la máquina en donde se encuentra el proceso servidor.

3º. Escucha (operación listen). Una vez que se ha creado el socket y se le ha asignado un puerto, se debe **configurar para que escuche por dicho puerto**. La operación listen hace que el socket servidor quede preparado para aceptar conexiones por parte de un proceso cliente.

4º. Acceptación de conexiones (operación accept). Cuando llega una petición de conexión, **se crea un nuevo socket dentro del proceso servidor.** Este **nuevo socket** es el que queda conectado con el socket del proceso cliente, estableciendo un canal de comunicación privado y estable entre ambos. El **socket servidor queda libre**, escuchando a la espera de nuevas conexiones.

5º. **Envío y Recepción de mensajes**. Ahora el proceso servidor puede **enviar/recibir** mensajes mediante operaciones de **escritura/lectura sobre su nuevo socket. No se usa el socket servidor** para realizar esta tarea, puesto que no está conectado.

6º. **Cierre de la conexión (operación close).** Si desea finalizar la conexión el proceso servidor puede **cerrar el nuevo socket**. El socket servidor sigue estando disponible a la escucha de nuevas conexiones.



SOCKETS DATAGRAM

Los sockets datagram no son orientados a conexión, pueden usarse para enviar mensajes (datagramas) a multitud de receptores. **Un mismo socket se puede usar para enviar mensajes a distintos receptores**, simplemente es necesario especificar la dirección y puerto de destino en cada operación send, por lo que no hay un canal privado y permanente entre emisor y receptor. La operación close se realiza solamente cuando ya no se desea seguir usando el socket.

Se usa un canal temporal para cada envío, para cada mensaje. No son fiables ni aseguran que el orden de entrega sea el mismo que el orden de envío. Cuando se usan sobre la pila de protocolos IP **hacen uso del protocolo de transporte UDP**. **No se diferencia claramente el papel de proceso servidor/ proceso cliente.**

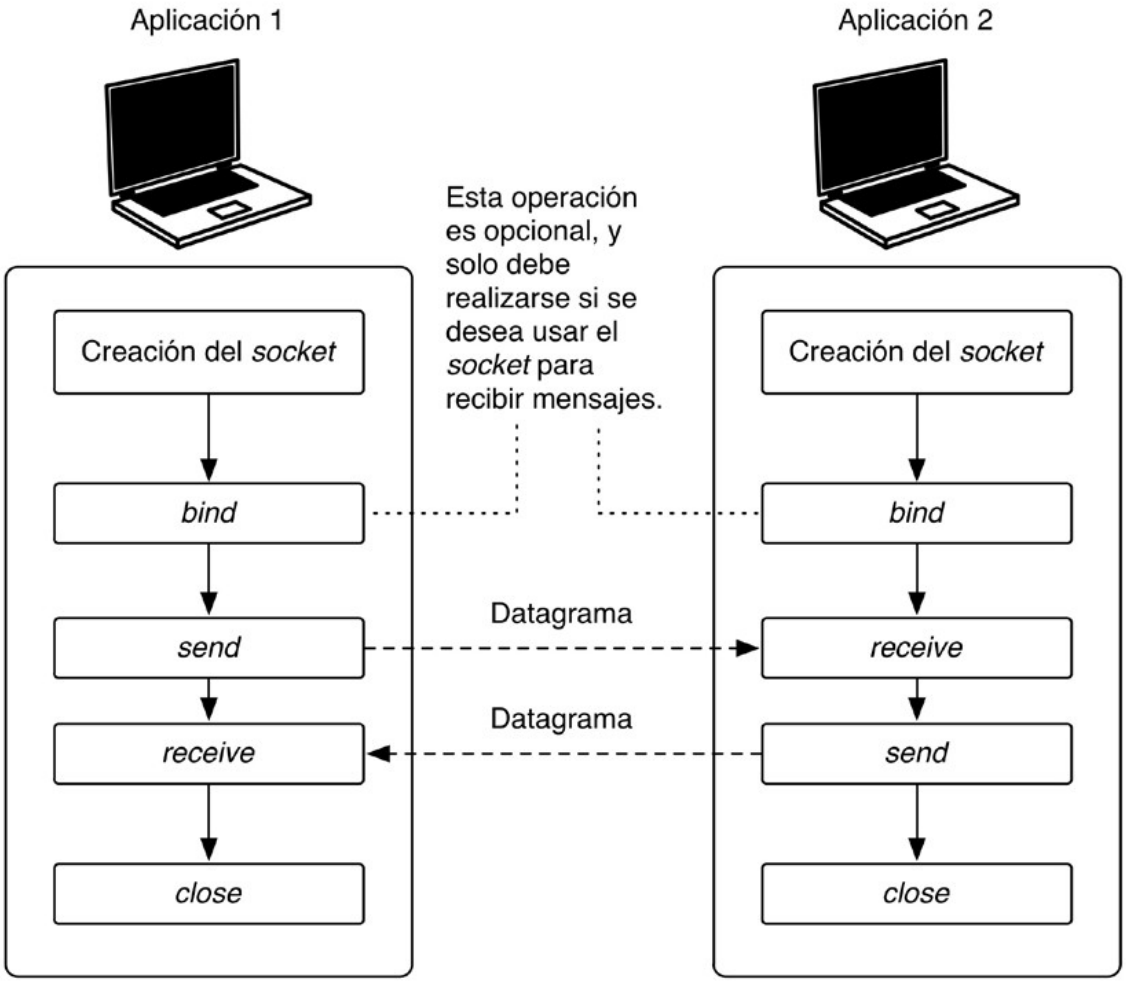
Secuencia de operaciones desde cualquier proceso que use sockets datagram:

1º. Creación del socket.

2º. Asignación de dirección y puerto (operación bind). En el caso de que se desee usar el socket para recibir mensajes es importante que la dirección y puerto estén claramente especificados. Sino los emisores no serán capaces de localizar a los receptores. La operación bind asigna una dirección IP y un número de puerto concreto al socket. La dirección IP asignada debe ser la de la máquina en donde se encuentra la aplicación. **Esta operación es optativa, solo debe realizarse si se desea usar el socket para recibir mensajes**.

3º. Envío y Recepción de mensajes (operaciones send/receive). En el caso de sockets datagram existen dos operaciones especiales: enviar datagrama **send** y recibir datagrama **receive.** La operación send necesita que se le especifique una dirección IP y un puerto, son los datos que se usarán como identificadores del destinatario del datagrama.

4º. **Cierre de la conexión (operación close**). El socket se **puede** cerrar usando la operación close.



PROGRAMACIÓN CON SOCKETS

Las clases principales en Java que permiten la comunicación por sockets:

La clase Socket. java.net.Socket, para la creación de **sockets stream cliente**.

La clase ServerSocket. java.net.ServerSocket, para la creación de **sockets stream servidor.**

La clase DatagramPacket. java.net.DatagramSocket Esta clase da soporte para crear instancias de los datagramas que se van a recibir y de los datagramas que serán enviados.

La clase DatagramSocket. java.net.DatagramSocket, para la creación de **sockets datagram. Esta clase da soporte para el envío y recepción de datagramas UDP.**

La clase Socket, (java.net.Socket) sus métodos más importantes:

MÉTODO	TIPO DE RETORNO	FUNCIÓN
Socket()	Socket	Constructor de la clase. Crea un socket stream cliente. Sin ningún puerto asociado.
Socket(String Host,int port)	Socket	Crea un socket y lo conecta al número de puerto y al nombre de host especificados. Puede lanzar UnKnownHostException, IOException.
Socket(InetAddress address, int port)	Socket	Crea un socket y lo conecta al puerto y dirección IP especificados.
Socket(InetAddress address, int port,InetAddressLocal addr, int localport)	Socket	Crea un socket y lo conecta al puerto y dirección IP especificados. Permite además especificar la IP local y el puerto local a los que se asociará el socket.
connect(SocketAddress adr)	void	Establece la conexión con la dirección y puerto de destino, con la aplicación servidor. Crea un canal de comunicación privado con el proceso servidor.
getInputStream()	InputStream	Obtiene un objeto de clase InputStream, flujo de entrada que se usa para realizar operaciones de lectura de

		bytes para leer desde el socket. Puede lanzar la IOException. El socket debe estar conectado.
getOutputStream()	OutputStream	Obtiene un objeto de clase OutputStream, flujo de salida, que se usa para realizar operaciones de escritura de bytes en el socket.Puede lanzar IOException. El socket debe estar conectado.
getInetAddress()	InetAddress	Devuelve la dirección IP a la que el socket está conectado.Si no está conectado devuelve null.
getLocalPort()	int	Devuelve el puerto local al que está enlazado el socket. Si no está enlazado a ningún puerto -1.
getPort()	int	Devuelve el puerto remoto al que está conectado el socket. 0 si no está conectado a ningún puerto.
close()	void	Cierra el socket

La clase ServerSocket (java.net.ServerSocket), sus métodos más importantes:

MÉTODO	TIPO DE RETORNO	FUNCIÓN
ServerSocket()	ServerSocket	Constructor básico de la clase. **Crea un socket stream servidor.**
ServerSocket(String host, int port)	ServerSocket	Constructor alternativo de la clase. Recibe la dirección IP y el puerto que se desea asignar al socket. Realiza las operaciones de creación del socket y bind directamente.
ServerSocket(int port)	ServerSocket	Crea un socket servidor que enlaza al puerto especificado.
ServerSocket(int port, int maximo)	ServerSocket	Crea un socket servidor y lo enlaza al puerto local especificado, el parámetro máximo indica el número máximo de peticiones de conexión que se pueden mantener en la cola.
ServerSocket(int port, int máximo, InetAddress direc)	ServerSocket	Crea un socket de servidor y lo enlaza al puerto local especificado, especificando un máximo de peticiones de conexión entrantes y la dirección IP local
bind(SocketAddress bindpoint)	void	Asigna al socket creado una dirección y número de puerto determinado.
accept()	Socket	El proceso servidor escucha por el socket servidor, esperando conexiones por parte de clientes. Cuando llega una conexión devuelve un **nuevo objeto de la clase Socket,** crea el canal**,** este nuevo socket queda conectado al cliente.
getInputStream()	InputStream	Obtiene un objeto de clase InputStream, flujo de entrada que se usa para realizar operaciones de lectura de bytes sobre él, para leer del socket.
getOutputStream()	OutputStream	Obtiene un objeto de clase OutputStream, flujo de

		salida, que se usa para realizar operaciones de escritura en él, escribir bytes en el socket.
getLocalPort()	int	Devuelve el puerto local al que está enlazado el ServerSocket
close()	void	Cierra el socket

ENVÍO DE OBJETOS A TRAVÉS DE SOCKETS

Hasta ahora hemos intercambiado cadenas de caracteres entre programas cliente y servidor.

Pero los stream soportan diversos tipos de datos como son los bytes, los tipos de datos primitivos Java y **objetos**.

OBJETOS EN SOCKTES TCP

Las clases **ObjectInputStream** y **ObjectOutputStream** nos permiten enviar objetos a través de los socktes TCP.

Con los métodos **readObject()** y **writeObject()** leeremos un objeto desde el stream y escribiremos un objeto en el stream.

```
//preparando flujo de salida para escribir en él objetos.
ObjectOutputStream outObjeto=
        new ObjectOutputStream(socket.getOutpuStream());
//preparando flujo de entrada para leer de él objetos
ObjectInputStream inObjeto=
    new ObjectInputStream(socket.getInputStream());
```

Las clases a las que pertenecen los objetos a transmitir entre las aplicaciones deben implementar la interfaz Serializable

CONEXIONES DE MÚLTIPLES CLIENTES. HILOS.

La solución para que un programa servidor pueda atender a múltiples clientes está en el **multihilo,** cada cliente será atendido en un hilo**.**

El esquema básico en sotckets TCP será construir un único servidor con la clase ServerSocket e invocar al método accept() para esperar las peticiones de conexión de los clientes. Cuando un cliente se conecta, el método accept() devuelve un objeto Socket, este se usuará para crear un hilo cuya misión es atender a ese cliente. Después se vuelve a invocar al método accept() para esperar a un nuevo cliente, habitualmente la espera de conexiones se hace en un bucle infinito.

CLASES PARA SOCKETS UDP:

. La clase DatagramPacket

. La clase DatagramSocket

En los sockets UDP son más simples y eficientes que los TCP pero no está garantizadas la entrega de paquetes. **Roles emisor/receptor**

No se establece una conexión exclusiva entre cliente y servidor como en el caso de TCP, por ello cada vez que se envíe un datagrama **el emisor** debe indicar explícitamente la dirección IP y el puerto del destino **para cada paquete** y **el receptor** debe extraer la dirección IP y el puerto del emisor del paquete. Los roles cliente-servidor están más difusos, que en el caso de TCP.

Roll emisor:

// **Construimos datagrama a enviar** indicando el host de destino y puerto

CADENA DE BYTES CONTENIDO DEL MENSAJE	LONGITUD DEL MENSAJE	DIRECCIÓN IP DESTINO	N.º DE PUERTO DESTINO

…….

```
DatagramPacket envio= new DatagramPacket(mensaje, mensaje.lenght,destino, port);

DatagramSocket socket= new DatagramSocket(34567);

socket.send(envio);
```

Roll receptor:

// En el otro extremo, para recibir el datagrama

```
DatagramSocket socket= new DatagramSocket(12345);
```

//construimos datagrama a recibir

CADENA DE BYTES A RECIBIR CONTENIDO MENSAJE	LONGITUD DEL MENSAJE

……..

```
DatagramPacket recibo=new DatagramPacket(buffer, buffer.length);

socket.receive(recibo); // recibo datagrama

int bytesrecibidos=recibo.getLength();

String paquete= new String(recibo.getData());

System.out.println("número de bytes recibidos: "+bytesrecibidos);
```

System.out.println (“contenido del paquete: “+paquete.trim());

System.out.println(“Puerto origen del mensaje: “+**recibo.getPort()**);

System.out.println(“IP de origen: “+**recibo.getAddress().getHostAddress()**);

System.out.println(“Puerto destino del mensaje: “+**socket.getLocalPort()**);

La clase DatagramPacket

CONSTRUCTOR	TIPO DE RETORNO	FUNCIÓN
DatagramPacket(byte[] buf,int length)	DatagramPacket	**Constructor para datagramas recibidos,** especificamos cadena de bytes en la que se alojará el mensaje y la longitud del mismo.
DatagramPacket(byte[] buf,int length, InetAddress addrss, int port)	DatagramPacket	**Constructor para el envío de datagramas,** Especificamos la cadena de bytes a enviar, la longitud de la misma, además el host especificado en la dirección addrss y el número de puerto de destino.

Algunos métodos de la clase DatagramPacket

MÉTODOS	TIPO DE RETORNO	FUNCIÓN
getAddress()	InetAddress	Devuelve la dirección IP del host al cual se envía el datagrama o del que se recibió el datagrama.
getData()	byte[]	Obtiene el mensaje contenido en el datagrama tanto el datagrama recibido como enviado
getLength()	int	Obtiene la longitud de los datos a enviar o recibir.
getPort()	int	Obtiene el número de puerto de la máquina remota a la que se le va a enviar el datagrama o de la que se recibió el datagrama.
setAddress(InetAddress addr)		Estable la dirección IP de la máquina a la que se le envía el datagrama. Se usa para construir el paquete datagrama a enviar.
setData(byte[] buf)		Establece el buffer de datos para el paquete datagrama.Se usa para construir el paquete datagrama a enviar.
setLength(int length)		Almacena la longitud de este paquete. Se usa para construir el paquete datagrama a enviar.
setPort(int port)		Almacena el número de puerto del host remoto al que este datagrama se eviará.Se usa para construir el paquete datagrama a enviar.

La clase DatagramSocket

Da soporte a sockets para envío y recepción de datagramas UDP.

CONSTRUCTOR	TIPO DE RETORNO	FUNCIÓN
DatagramSocket()	DatagramSocket	Construye un socket para datagramas, el sistema elige un puerto de los que está libres.
DatagramSocket(int port)	DatagramSocket	Construye un socket para datagramas y lo conecta al puerto local especificado.
DatagramSocket(int port, InetAddress ip)	DatagramaSocket	Permite especificar el puerto local y la dirección local a la que se va asociar el socket

MÉTODOS	TIPO DE RETORNO	FUNCIÓN
receive(DatagramPacket paquete)		Recibe un DatagramPacket del socket y llena paquete con los datos que recibe (mensaje,longitud,origen). Puede lanzar la excepción IOExecption
send(DatagramPacket paquete)		Envía un DatagramPacket a través del socket.El argumento paquete contiene el mensaje y su destino.Puede lanzar la excepción IOException.
close()		Cierra el socket
getLocalPort()	int	Devuelve el número de puerto en el host local al que el socket está enlazado, -1 si el socket está cerrado y 0 sino está enlazado a ningún puerto.
getPort()	int	Devuelve el número de puerto al que está conectado el socket,-1 sino está conectado.
connect(InetAddress address, int port)		Conecta el socket a un puerto remoto y a una dirección IP concretos, el socket sólo podrá enviar y recibir mensajes a/desde esa dirección.
setSoTimeout(int timeout)		Permite establecer un tiempo de espera límite. Entonces el método **receive** se bloquea durante el tiempo fijado. Si no se reciben datos en el tiempo fijado se lanza la excepción InterruptedIOException.