Clase del día - 26/03/2021

La clase de hoy veremos el tema de Comunicación en un grupo confiable.

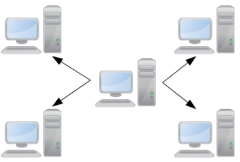
**Tipos de comunicaciones**

Los tipos de comunicaciones entre computadoras son los siguientes:

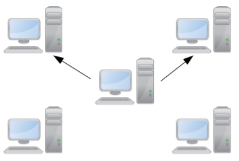
**Unicast**. El unicast es una comunicación punto a punto dónde una computadora envía mensajes a otra computadora.



**Broadcast**. El broadcast es un tipo de multi-transmisión en la cual una computadora envía mensajes a todas las computadoras en una red.



**Multicast**. El multicast también es un tipo de multi-transmisión en la cual  una computadora envía mensajes a una o más computadoras en una red. El broadcast es un caso particular de multicast, cuando la computadora envía mensajes a todas las computadoras de la red.



**Tolerancia a fallas**

Un sistema distribuido es tolerante a las fallas si tiene la capacidad de proveer sus servicios incluso ante la presencia de fallas, es decir, el sistema continua operando con normalidad ante las fallas.

**Fiabilidad de un sistema**

En la medida que un sistema es tolerante a las fallas es un sistema fiable.

La **fiabilidad** de un sistema es un requerimiento no funcional, el cual a su vez se compone de los siguientes sub-requerimientos no funcionales (recordar que en Ingeniería de Software los requerimientos funcionales y no funcionales se pueden dividir en sub-requerimientos):

**Disponibilidad**. La disponibilidad es la capacidad que tiene un sistema de ser utilizado al momento, es decir, la probabilidad de que el sistema funcione correctamente siempre.

**Confiabilidad**. La confiabilidad es la capacidad de un sistema de funcionar continuamente sin fallar. La confiabilidad se define en términos de un intervalo de tiempo de funcionamiento continuo, a diferencia de la disponibilidad la cual se refiere al funcionamiento del sistema en un momento dado.

Por ejemplo, si un sistemas se cae un segundo cada día, se dice que tiene una disponibilidad de 1-1/(24x60x60)=99.998%, sin embargo no es un sistema confiable si consideramos un proceso que puede tardar más de un día y el proceso no puede terminar debido a las caídas del sistema.

**Seguridad**. La seguridad, desde el punto de vista de la tolerancia a fallas, se refiere a la propiedad que tiene el sistema de no causar un evento catastrófico cuándo falla. Por ejemplo, un sistema de conducción autónoma no es seguro si al fallar el automóvil choca y provoca daños a los pasajeros.

**Mantenimiento**. El mantenimiento se refiere a la capacidad que tiene el sistema de ser reparado cuando falla.

**Clasificación de las fallas de un sistema**

Las fallas en un sistema se pueden clasificar en cinco categorías:

**Falla de congelación**. La falla de congelación se presenta cuando el sistema estaba funcionando normalmente y de pronto se detiene.

**Falla de omisión**. Una falla de omisión se presenta cuando el sistema no recibe los mensajes (*omisión de recepción*) o no envía los mensajes (*omisión de envío*).

**Falla de tiempo**. Una falla de tiempo se produce cuando el tiempo de respuesta del sistema es mayor al especificado en los requisitos no funcionales.

**Falla de respuesta**. El sistema presenta una falla de respuesta cuando se produce un valor incorrecto en la respuesta (*falla de valor*) o una respuesta incorrecta debido a una desviación en la ejecución del algoritmo (*falla de transición de estado*).

**Falla arbitraria**. El sistema presenta una falla arbitraria cuandoproduce respuestas arbitrarias, en cualquier momento y con tiempos de repuesta arbitrarios.

En un sistema distribuido pueden fallar los servidores y/o las comunicaciones.

Las fallas que presenta un canal de comunicación pueden ser fallas por congelación, omisión, tiempo y fallas arbitrarias. Veremos más adelante que las fallas que reciben especial atención son las fallas por congelación y omisión.

Un canal de comunicación confiable es aquel que oculta las fallas en la comunicación.

**Socket, dirección IP y puerto**

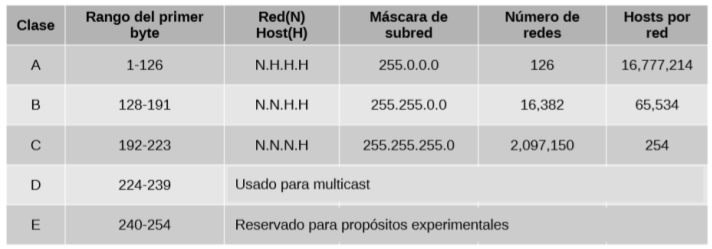
Un socket es un punto final (*endpoint*) de un enlace de dos vías que comunica dos procesos que ejecutan en la red. Un *endpoint*es la combinación de una dirección IP y un puerto.

Una dirección IP versión 4 es un número de 32 bits dividido en cuatro bytes, cada byte puede tener un valor entre 0 y 255. El puerto es un número entre 0 y 65,535

**Clases de dirección IP v4**

Las direcciones IP versión 4 se dividen en cinco clases o rangos, a saber: Clase A, Clase B, Clase C, Clase D y Clase E. Cada clase se define por un rango de valores que puede tomar el primer byte de la dirección IP, así las clases A, B y C son utilizadas para la comunicación unicast, mientras que la clase D es utilizada exclusivamente para la comunicación multicast. La clase E está reservada para propósitos experimentales.

La siguiente tabla muestra los bytes que identifican a las redes y a los hosts en cada clase (Rango del primer byte), así como la máscara de subred, número de redes y número de hosts por red en cada clase.



Las direcciones 127.X.X.X (*loopback address*) son utilizadas para identificar a la computadora local (localhost).

La dirección 255.255.255.255 es usada para broadcast a todos los hosts en la LAN. Las direcciones 224.0.0.1 y 224.0.0.255 están reservadas.

**Protocolos TCP y UDP**

Un protocolo define la estructura de los paquetes de datos. Existen dos tipos de sockets:

**Socket stream**. Socket orientado a conexión:

* Se establece una conexión virtual uno-a-uno mediante *handshaking*.
* Los paquetes de datos son enviados sin interrupciones a través del canal virtual.
* El protocolo TCP (*Transmission Control Protocol*) es el más utilizado para sockets orientados a conexión.

Las características de los sockets conectados son las siguientes:

* Comunicación altamente confiable.
* Un canal dedicado de comunicación punto-a-punto entre dos computadoras.
* Los paquetes son enviados y recibidos en el mismo orden.
* El canal está ocupado aunque no se esté transmitiendo.
* Recuperación tardada de datos perdidos en el camino.
* Cuándo los datos son enviados se espera un acuse de recibo (*acknowledgement*).
* Si los datos no son recibidos correctamente se retransmiten.
* No se utilizan para broadcast ni multicast, ya que los sockets stream establecen solo una conexión entre dos endpoints.
* Se implementan mayormente usando protocolo TCP.

**Socket datagrama**. Socket desconectado:

* Los datos son enviados en un paquete a la vez.
* No se requiere establecer una conexión.
* El protocolo UDP (*User Datagram Protocol*) es el más utilizado para sockets desconectados.

Las características de los sockets desconectados son las siguientes:

* No requieren un canal dedicado de comunicación.
* La comunicación no es 100% confiable.
* Los paquetes son enviados y recibidos en diferente orden.
* Rápida recuperación de datos perdidos en el camino.
* No hay *acknowledgement* ni re-envío.
* Utilizados para broadcast y multicast.
* Utilizados para la transmisión de audio y video en tiempo real.
* Se implementan mayormente usando el protocolo UDP.

**Comunicación unicast confiable**

Para establecer la comunicación unicast confiable se utiliza generalmente el protocolo TCP, el cual implementa la retransmisión de mensajes para ocultar las fallas por omisión.

Las fallas por congelación (cuando se produce la desconexión) no son ocultadas por el protocolo TCP, sin embargo el cliente es informado de la falla de manera que pueda re-conectarse (ver el programa [Cliente2.java](https://m4gm.com/moodle/mod/resource/view.php?id=1297)).

**Comunicación multicast confiable**

Anteriormente hemos hablado de la posibilidad de replicar los datos y los servidores con el propósito de tolerar las fallas en un sistema distribuido. Sin embargo, la replicación implica la multi-transmisión de los mensajes a las réplicas, lo cual requiere contar con una comunicación multicast confiable.

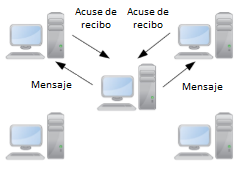
Definimos la comunicación multicast confiable como los mecanismos que garantizan que todos los miembros de un grupo reciben los mensajes transmitidos, sin importar el orden en que reciben los mensajes.

La comunicación punto a punto confiable se implementa con relativa facilidad mediante TCP, sin embargo la comunicación multicast confiable resulta mucho más complicada.

Una primera aproximación para implementar la comunicación multicast confiable es la utilización de múltiples conexiones punto a punto, sin embargo esta solución resulta poco eficiente debido a las características del protocolo TCP.

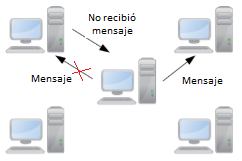
Como vimos anteriormente, la comunicación multicast basada en sockets datagrama no es 100% confiable, debido a que es posible que algunos paquetes se pierdan en el camino, además, los paquetes no son recibidos en el orden en que son enviados.

Una segunda aproximación para la implementación de la comunicación multicast confiable es la utilización de sockets desconectados. En este caso, para garantizar que todos los procesos reciben todos los mensajes, cada proceso receptor deberá enviar un mensaje de acuse de recibo (*acknowledgement*), si el acuse no se recibe en un tiempo determinado, entonces el transmisor deberá retransmitir el mensaje al proceso faltante.



La solución anterior tiene una desventaja, y es que cada mensaje enviado por el transmisor produce una multitud de acuses de recibo, lo cual degrada al transmisor.

Una tercera aproximación es el envío de acuse de recibo "negativo", esto es, si un receptor no recibe un mensaje en un tiempo determinado, entonces envía al transmisor un mensaje indicando que no ha recibido el mensaje. Desde luego, el receptor deberá tener información sobre los mensajes que recibirá, esto se puede lograr agregando al mensaje actual metadatos del siguiente mensaje.



**Multicast atómico**

El multicast atómico se refiere a la garantía de que un mensaje llegue a todos a destinatarios o a ninguno.

La atomicidad en la comunicación multicast es de utilidad para la implementación de los requerimientos no funcionales que tienen que ver con la consistencia de los datos.

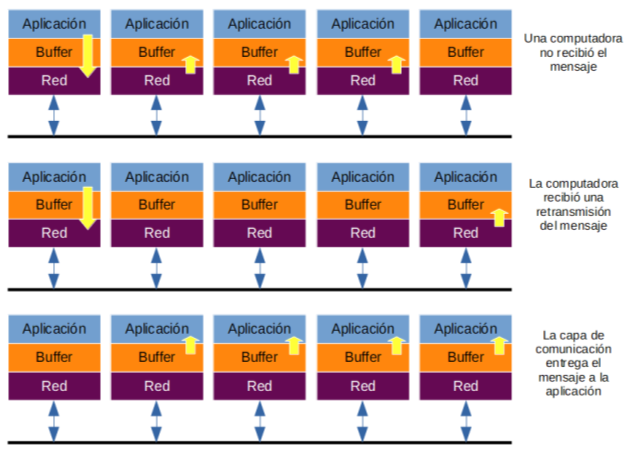
Por ejemplo, si un archivo es replicado en un grupo de computadoras, los cambios que se realizan al archivo deben ser replicados en **todas** las computadoras que forman parte del grupo. Si bien la consistencia del archivo es un requerimiento no funcional del sistema de archivos distribuido, este requerimiento puede ser satisfecho por la capa de comunicaciones si ésta ofrece el multicast atómico.

Existe una variedad de soluciones al multicast atómico, aquí estudiaremos una aproximación basada en la comunicación multicast confiable.

Como vimos anteriormente, la comunicación multicast confiable garantiza que todos los mensajes son recibidos por todos los receptores. Entonces, para contar con el multicast atómico será necesario garantizar que todos los miembros de un grupo reciben los mensajes. Por tanto hay que distinguir entre "recibir el mensaje" y "entregar el mensaje".

Supongamos que una computadora miembro de un grupo envía un mensaje al resto de computadoras en el grupo, sin embargo, por alguna razón, el mensaje no llega a una de las computadoras. Desde luego, el resto de computadoras "recibieron" el mensaje, sin embargo la capa de comunicaciones no puede entregar el mensaje a las aplicaciones antes de confirmar que todos los miembros del grupo en efecto recibieron el mensaje.

Entonces los mensajes entrantes deberán permanecer en un almacén temporal (buffer) en la capa de comunicaciones, y solo en el caso de que todas las computadoras confirmen la recepción del mensaje, entonces y solo entonces el mensaje será entregado a las aplicaciones.



Ahora vamos a ver cómo programar un cliente y un servidor multicast.

En el caso de la comunicación multicast, el servidor es el programa que envía mensajes a los clientes, por esta razón es necesario que los clientes invoquen la función **receive**antes que el servidor ejecute la función **send**.

La comunicación multicast se implementa mediante sockets desconectados, por tanto no se requiere que se establezca una conexión dedicada entre el servidor y el cliente.

Para recibir un mensaje del servidor, los clientes se "unen" a un grupo de manera que el servidor envía mensajes al grupo sin conocer el número de clientes ni sus direcciones IP.

Un grupo multicast se identifica mediante una dirección IP de clase D. Un grupo multicast se crea cuando se une el primer cliente y deja de existir cuando el último cliente abandona el grupo.

[ServidorMulticast.java](https://m4gm.com/moodle/mod/resource/view.php?id=1768)

El programa [ServidorMulticast.java](https://m4gm.com/moodle/mod/resource/view.php?id=1768) es un ejemplo de un servidor que utiliza sockets UDP para enviar mensajes a un grupo de clientes.

Primeramente vamos a implementar el método envia\_mensaje() el cual recibe como parámetros un arreglo de bytes (el mensaje), la dirección IP clase D que identifica el grupo al cual se enviará el mensaje, y el número de puerto.

static void envia\_mensaje(byte[] buffer,String ip,int puerto) throws IOException

{

DatagramSocket socket = new DatagramSocket();

InetAddress grupo = InetAddress.getByName(ip);

DatagramPacket paquete = new DatagramPacket(buffer,buffer.length,grupo,puerto);

socket.send(paquete);

socket.close();

}

Notar que el método envia\_mensaje() puede producir excepciones de tipo IOException.

En este caso declaramos una variable de tipo DatagramSocket la cual va a contener una instancia de la clase DatagramSocket.

Obtenemos el grupo correspondiente a la IP, invocando el método estático getByName() de la clase InetAddress.

Para crear un paquete con el mensaje creamos una instancia de la clase DatagramPacket. Entonces enviamos el paquete utilizando el método send() de la clase DatagramSocket.

Finalmente cerramos el socket invocando el método close().

Ahora vamos a enviar  la cadena de caracteres "hola", en este caso se envía el mensaje al grupo identificado por la IP 230.0.0.0:

envia\_mensaje("hola".getBytes(),"230.0.0.0",50000);

Vamos a enviar cinco números punto flotante de 64 bits.

Primero "empacaremos" los números utilizando un objeto ByteBuffer. Cinco números punto flotante de 64 bits ocupan 5x8 bytes (64 bits=8 bytes). Entonces vamos a crear un objeto de tipo ByteBuffer con una capacidad de 40 bytes:

ByteBuffer b = ByteBuffer.allocate(5\*8);

Utilizamos el método **putDouble**para agregar cinco números al objeto ByteBuffer:

b.putDouble(1.1);

b.putDouble(1.2);

b.putDouble(1.3);

b.putDouble(1.4);

b.putDouble(1.5);

Para enviar el paquete de números, convertimos el objeto BytetBuffer a un arreglo de bytes utilizando el método **array()** de la clase ByteBuffer. Entonces enviamos el arreglo de bytes utilizando el método envia\_mensaje(), en este caso el mensaje se envía al grupo identificado por la dirección IP 230.0.0.0 a través del puerto 50000:

envia\_mensaje(b.array(),"230.0.0.0",50000);

[ClienteMulticast.java](https://m4gm.com/moodle/mod/resource/view.php?id=1771)

Vamos a implementar el método recibe\_mensaje() al cual pasamos como parámetros un socket de tipo MulticastSocket y la longitud del mensaje a recibir (número de bytes).

static byte[] recibe\_mensaje(MulticastSocket socket,int longitud\_mensaje) throws IOException

{

byte[] buffer = new byte[longitud\_mensaje];

DatagramPacket paquete = new DatagramPacket(buffer,buffer.length);

socket.receive(paquete);

return paquete.getData();

}

Notar que el método recibe\_mensaje() puede producir una excepción de tipo IOException.

Creamos un paquete vacío como una instancia de la clase DatagramPacket; pasamos como parámetros un arreglo de bytes vacío y el tamaño del arreglo.

Para recibir el paquete invocamos el método recive() de la clase MulticastSocket. El método recibe\_mensaje() regresa el mensaje recibido.

Ahora vamos a recibir mensajes utilizando el método recibe\_mensaje().

Para obtener el grupo invocamos el método getByName() de la clase InetAddress, en este caso se obtiene el grupo identificado por la IP 230.0.0.0:

InetAddress grupo = InetAddress.getByName("230.0.0.0");

Luego obtenemos un socket asociado al puerto 50000, creando una instancia de la clase MulticastSocket:

MulticastSocket socket = new MulticastSocket(50000);

Para que el cliente pueda recibir los mensajes enviados al grupo 230.0.0.0 unimos el socket al grupo utilizando el método joinGroup() de la clase MulticastSocket:

socket.joinGroup(grupo);

Entonces el cliente puede recibir los mensajes enviados al grupo por el servidor.

Primeramente vamos a recibir una cadena de caracteres:

byte[] a = recibe\_mensaje(socket,4);

System.out.println(new String(a,"UTF-8"));

Ahora vamos a recibir cinco números punto flotante de 64 bits empacados como arreglo de bytes:

byte[] buffer = recibe\_mensaje(socket,5\*8);

ByteBuffer b = ByteBuffer.wrap(buffer);

for (int i = 0; i < 5; i++)  
 System.out.println(b.getDouble());

Finalmente, invocamos el método leaveGroup() para que el socket abandone el grupo y cerramos el socket:

socket.leaveGroup(grupo);

socket.close();

Actividades individuales a realizar

1. Compile los programas [ServidorMulticast.java](https://m4gm.com/moodle/mod/resource/view.php?id=1768) y [ClienteMulticast.java](https://m4gm.com/moodle/mod/resource/view.php?id=1771).
2. Ejecute el programa [ClienteMulticast.java](https://m4gm.com/moodle/mod/resource/view.php?id=1771) en tres ventanas de comandos de Windows (o terminales de Linux) y ejecute el programa [ServidorMulticast.java](https://m4gm.com/moodle/mod/resource/view.php?id=1768) en otra ventana de comandos de Windows (o terminal de Linux). Notar que primero se debe ejecutar los clientes y después se ejecuta el servidor.