**SDCoulouris5ed.pdf**

**Introducción**

Un sistema distribuido es aquel en el que los componentes ubicados en computadoras en red se comunican y coordinan sus acciones solo pasando mensajes. Esta definición lleva a las siguientes características especialmente significativas de los sistemas distribuidos: concurrencia de componentes, falta de un reloj global y fallas independientes de los componentes.

Los desafíos que surgen de las construcciones de sistemas distribuidos son la heterogeneidad de sus componentes, la apertura (que permite agregar o reemplazar componentes), la seguridad, la escalabilidad, el manejo de fallas, la concurrencia de componentes, la transparencia y la calidad de los servicios.

Definimos un sistema distribuido como uno en el que los componentes de hardware y software ubicados en computadoras en red se comunican y coordinan sus acciones solo pasando mensajes. Las computadoras que están conectadas por una red pueden estar separadas espacialmente por cualquier distancia. Nuestra definición de sistemas distribuidos tiene las siguientes consecuencias significativas:

Concurrencia: la capacidad del sistema para manejar recursos compartidos se puede aumentar agregando más recursos a la red. La coordinación de la ejecución simultánea de programas que comparten recursos también es un tema importante y recurrente.

Sin reloj global: cuando los programas necesitan cooperar, coordinan sus acciones mediante el intercambio de mensajes. La coordinación estrecha a menudo depende de una idea compartida del momento en que ocurren las acciones de los programas. Pero resulta que hay límites en la precisión con la cual las computadoras en una red pueden sincronizar sus relojes: no existe una noción global única de la hora correcta. Esta es una consecuencia directa del hecho de que la única comunicación es mediante el envío de mensajes a través de una red.

Fallos independientes: todos los sistemas informáticos pueden fallar, y es responsabilidad de los diseñadores de sistemas planificar las consecuencias de posibles fallas. Los sistemas distribuidos pueden fallar de nuevas maneras. Las fallas en la red provocan el aislamiento de las computadoras que están conectadas a ella, pero eso no significa que dejen de funcionar. De hecho, los programas en ellos pueden no ser capaces de detectar si la red ha fallado o se ha vuelto inusualmente lenta. Del mismo modo, la falla de una computadora, o la terminación inesperada de un programa en algún lugar del sistema (un bloqueo), no se da a conocer de inmediato a los otros componentes con los que se comunica. Cada componente del sistema puede fallar de forma independiente, dejando a los demás aún en ejecución.

La principal motivación para construir y utilizar sistemas distribuidos proviene del deseo de compartir recursos.

**Ejemplos de sistemas distribuidos**

* Comercio y finanzas: Comercio en línea como Amazon y eBay; tecnologías de pagos online como Paypal; la banca electrónica.
* Manejo de la información: La World Wide Web como repositorio de información y conocimiento; buscadores web como Google; librerías digitales en línea con eBooks como Google Books, contenido generado por usuarios, como en Youtube y Wikipedia; redes sociales como Facebook.
* Entretenimiento: Gaming online; música y películas desde el hogar a través de centros de medios en red o en Internet a través de contenido descargable o de streaming; creación de arte.
* Medicina: Registros electrónicos de pacientes en línea y cuestiones relacionadas con la privacidad; telemedicina en el apoyo del diagnóstico remoto o servicios como la cirugía remota; sistemas remotos de vida asistida.
* Educación: E-learning por medio de herramientas web, como los entornos virtuales de aprendizaje; aprendizaje a distancia; soporte para el aprendizaje colaborativo.
* Transporte y logística: Uso de GPS para sistemas de búsqueda de rutas y de información de tráfico; el automóvil moderno como sistema complejo distribuido (aeronaves también); servicios basados en la web de mapas como Google Maps.
* Ciencia: Redes neuronales de computadoras como soporte para almacenamiento, análisis y procesamiento de grandes cantidades de datos científicos; tecnologías para colaboración con grupos científicos de todo el mundo.
* Gestión ambiental: Sensores en red para administrar y monitorear el medio ambiente; advertencia previa de tsunamis, inundaciones o terremotos; coordinar respuestas de emergencia; filtro y análisis de parámetros del ambiente para estudiar fenómenos naturales como el cambio climático.

**SDCoulourisEsp3ed.pdf**

Los desafíos con los que se trabaja al momento de diseñar sistemas distribuidos son variados, aquí se muestran algunos que los diseñadores deben considerar al momento de desarrollar sistemas distribuidos.

**Heterogeneidad**

Se debe permitir la variedad y diferencia en cuestiones de redes, hardware de computadoras, Sistemas Operativos, Lenguajes de programación, Implementaciones de diferentes desarrolladores

**Extensibilidad:**

Es la capacidad de que el sistema pueda tener nuevas funcionalidades sin necesidad de tener cambios muy significativos. La extensibilidad se adquiere al utilizar interfaces para hacer uso de las funciones del sistema.

**Seguridad:**

El sistema debe cumplir con las tres características principales de seguridad: confidencialidad, integridad y disponibilidad. Esto se garantiza a través del uso de cifrados e implementaciones de autenticación en el sistema, la infraestructura del sistema y las buenas prácticas usadas por los programadores.

**Escalabilidad:**

El sistema es capaz de crecer en cuanto recursos y usuarios. Esto quiere decir que si al sistema se le es agregado un recurso como memoria o poder de procesamiento, el sistema seguirá funcionando de una manera eficiente y en el caso de los usuarios la carga es proporcional al uso de recursos.

**Tratamiento de Fallos:**

Un sistema distribuido debe ser capaz de lidiar con los fallos, si uno de sus componentes dejase de funcionar este debe ser transparente al usuario. El correcto funcionamiento del sistema se garantiza a través de diversas técnicas para preservar información, para preservar el software y para el correcto uso de los recursos. Esto lo conocemos como tolerancia ante fallos.

Funciones principales:

* Detección de fallos
* Enmascaramiento de fallos
* Tolerancia de fallos
* Reparación frente a fallos
* Redundancia

**Concurrencia:**

Capacidad del sistema para interactuar con varios usuarios que acceden a distintos recursos sin que esto provoque algún conflicto entre los mismos usuarios. Se hace un aislamiento de los procesos.

**Transparencia:**

Define que tanto puede ver el usuario del sistema. Por ejemplo:

* De acceso
* De Ubicación
* De concurrencia
* De replicación
* De frente a fallos
* De movilidad
* De prestaciones
* De escalado

**SDCoulourisEsp3ed.pdf**

**4 Comunicación entre procesos**

La interfaz de programación de aplicaciones (API) de Java para la comunicación entre procesos en internet proporciona comunicación tanto por datagramas como por streams. Estas proveen bloques constructivos alternativos para los protocolos de comunicación.

**4.1 Introducción**

La interfaz del programa de aplicación para UDPproporciona un abstracción del tipo paso de mensajes, la formas mas simple de comunicacion entre procesos. Esto hace que el proceso emisor pueda transmitir un mensaje simple al proceso receptor. Los paquetes independientes que contienen estos mensajes se llaman datagramas. Tanto en Java como en cada API UNIX, el emisor especifica el destino utilizando un zócalo, conector o socket.

La interfaz del programa de aplicación de TCP proporciona la abstracción de un flujo (Stream) de dos dirección entre pares de procesos. La información intercambiada consiste en un stream de items de datos, sin límites entre mensajes. Los stream son un bloque básico para la construcción de la comunicación productor-consumidor. Un productor y un consumidor forman un par de procesos en los cuales el papel del primero es producir items de datos y del segundo consumirlos.

La multidifusión en grupos es una forma de comunicación entre procesos en el cual un proceso de un grupo de procesos transmite el mismo mensaje a todos los miembros del mismo grupo.

**4.2.1 Las características de la comunicación entre procesos**

El paso de mensajes entre un par de procesos se puede basar en dos operaciones: envía y recibe, definidas en función del destino y del mensaje. Para que un proceso se pueda comunicar con otro, el proceso envía un mensaje a un destino y otro proceso en el destino recibe el mensaje. Esta actividad implica la comunicación de datos desde el proceso emisor al proceso receptor y puede implicar además la sincronización de los dos procesos.

**Comunicacion sincrona y asincrona**. A cada destino de mensajes se asocia una cola. Los procesos emisores producen mensajes que serán añadidos a las colas remotas mientras que los procesos receptores eliminan mensajes de las colas locales. La comunicación entre los procesos emisor y receptor puede ser sincrona o asincrona. En la forma síncrona, los procesos receptor y emisor se sincronizan con cada mensaje, tanto el envia como recibe son operaciones bloqueantes. En la forma de comunicación asíncrona, la utilización de la operación envia es no bloqueante, de modo que el proceso emisor puede continuar tan pronto como el mensaje haya sido copiado en el buffer local, y la transmisión del mensaje se lleva a cabo en paralelo con el proceso emisor.

**Destinos de los mensajes.** Los mensaje son enviados a dirección construidas por pares (direcciones Internet, puerto local). Un puerto local es el destino de un mensaje dentro de un compilador, especificado como un número entero. Un puerto tiene exactamente un receptor pero puede tener muchos emisores. Los procesos pueden utilizar múltiples puertos desde los que recibir mensajes.

Generalmente los servidores hacen públicos sus números de puerto para que sean utilizados por los clientes**.** Si el cliente utiliza una dirección Internet fija para referirse a un servicio, entonces ese servicio debe ejecutarse siempre en el mismo computador para que la dirección se considere válida. Esto se puede evitar utilizando algunas de las siguientes aproximaciones que proporcionan transparencia de ubicación:

Los programas cliente se refiere a los servicios por su nombre y utilizan un servidor de nombres o enlazador para trasladar esos nombres a ubicaciones del servidor en tiempo de ejecución. Esto permite reubicar los servicios en otro lugar, pero no migrarlos.

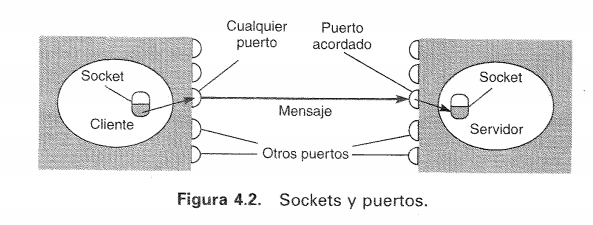
El sistema operativo proporciona identificadores para los destinos de los mensajes independientes de la localización, haciéndoles corresponder con direcciones de más bajo nivel utilizadas para entregar los mensajes en los puertos, permitiendo tanto la migración como la reubicación de los servicios en otro lugar

**Fiabilidad.** Una comunicación fiables se define en términos de validez e integridad. En lo que concierne a la propiedad de validez, se dice que un servicio de mensajes punto a punto es fiable si se garantiza que los mensajes se entrega a pesar de poder dejar caer o perder un número razonable de ellos

**Ordenación** Algunas aplicaciones necesitan que los mensajes sean entregados en el orden de su emisión, esto es, en el orden en el que fueron transmitidos por el emisor. La entrega de mensajes desordenados, por esas aplicaciones, es considerada como un fallo.

**4.2.2 Sockets**

La comunicación mediante los protocolos UDP y TCP se logra gracias a la abstracción de un socket. Su implementación es originaria de UNIX BSD y se encuentran actualmente en todo sistema operativo. Los sockets crean la comunicación entre procesos. En su forma más simple un socket debe tener asociado una dirección de red y un puerto que es por el cual se comunica. Un socket puede tanto recibir como enviar información. El número de puertos disponibles para la comunicación de de 2^16.



Los sockets en Java cuentan con una clase llamada InetAddress la cual abstrae las direcciones de red.

**4.2.3 Comunicación de Datagramas UDP**

El protocolo de comunicación de edatagramas (UDP) es usado cuando se requiere hacer una comunicación entre un cliente y un servidor sin tener que recibir mensajes de acuse por parte del cliente.

Aspectos a contemplar en la comunicación de datagramas:

* **Tamaño de mensaje:** el proceso receptor necesita especificar una cadena de bytes de un tamaño concreto sobre el cuál se alamacenará el mensaje recibido. Si el mensaje es demasiado grande, este será truncado. La capa subyacente de IP permite paquetes de hasta 216 bytes, cualquier aplicación que necesite mensajes mayores a este tamaño deberá fracmentarlos.
* **Bloqueo:** la comunicación de datagramas UDP utiliza operaciones de envío no bloqueantes y recepciones bloqueantes. Esto quiere decir que que la operación de envio delvolverá el control cuando el mensaje se ha dirigido a las capas inferiores de UDP e IP. A la llegada, el mensaje será colocado en una cola conectada que está enlazado con el puerto destino, si no existe ningun proceso ligado al conector destino los mensajes serán descartados.   
     
   El método *recibe* produce un bloque hasta que se reciba un datagrama, a menos que se haya establecido un *tiempo límite* (timeout) asocidado al conector.
* **Tiempo limite de espera:** El método *recibe* con bloqueo indefinido es adecuado para servidores que están esperando recibir peticiones de sus clientes. En algunos casos no resulta adecuado poner tiempos infenidos por posibles fallas del emisor, por lo cual se establecen tiempos límites de espera (*timeouts)* en los conectores. El tiempo límite de espera debe ser lo suficientemente grande en comparación con el tiempo adecuado de transmición de mensaje.
* **Recibe de cualquiera:** El método *recibe* no especifica el origen de los mensajes. En su lugar, al invocar *recibe* aceptamos mensajes dirigidos a su conector desde cualquier conector. El método *recibe* devuelve la dirección de Internet y el puerto emisor, permitiendo al receptor comprobar de donde viene el mensaje.

**Modelo de fallo:** El término modelo de fallo define las propiedades de una comunicación fiable en términos de dos propiedades: integridad y validez. La propiedad de integridad requiere que los mensajes no se corrompan ni se dupliquen. Este modelo de fallo puede utilizarse para propioner un modelo de fallo para los datagramas UDP, que padece de la siguientes debilidades:

* *Fallos de omisión:* Los mensajes pueden desecharse ocasionalmente, ya sea porque se ha producido un error detectado por la suma de comprobación o porque no queda espacio en el búfer ya sea en el origen o en el destino.
* *Ordenación: Algunas veces, los mensajes se entregan en desorden con respecto a su orden de emisión.*

***Las aplicaciones que utilizan datagramas UDP dependen de sus propias comprobaciones para conseguir la calidad que necesitan respecto a la fiabilidad de la comunicación.***

**Utilización de UDP:** Un ejemplo de la utilización de de un servicio que sea susceptible a sufrir fallos de omisión ocasionales es el Servicio de Nombres de Dominio en Internet (Domain Name Service, DNS). Los datagramas UDP no padecen las sobrecargas asociadas a la entrega de mensajes garantizada. Existen tres fuentes principales para esa sobrecarga:

1. La necesidad de almacenar información.
2. La transmisión de mensajes extra.
3. La latencia para el emisor.

**4.2.4 Comunicación de streams TCP**

API para el protocolo TCP originaria de UNIX BSD 4.x.

* Tamaño de los mensajes: Decide cuántos datos recoge antes de transmitirlos como paquetes IP. En el destino, los datos son proporcionados según se vayan solicitando.
* Mensajes perdidos: Si el emisor no recibe dicho acuse dentro de un plazo de tiempo, volverá a transmitir el mensaje.
* Control de flujo: Intenta ajustar las velocidades de los procesos que leen y escriben en un stream. Bloqueo de escritura hasta que la lectura finalice.
* Duplicación y ordenación de los mensajes: A cada paquete IP se le asocia un identificador; el receptor detecta y rechaza mensajes duplicados, y reordena mensajes que lleguen desordenados.
* Destinos de los mensajes: Un par de procesos establecen una comunicación, simplemente leen y escriben sin preocuparse por las direcciones de Internet ni de los números de puerto.

El API para la comunicación supone una conexión cliente - servidor, aunque después se comuniquen igual a igual. El cliente crea un stream conector sobre un puerto y envía una petición de conexión con el servidor en su puerto de servicio. El servidor crea un conector de escucha ligado al puerto de servicio y la espera de clientes que soliciten conexiones, manteniendo una cola de peticiones. Cuando un servidor acepta una conexión, crea un nuevo conector para mantener la comunicación con el cliente, mientras reserva el conector del puerto de servicio para escuchar peticiones de conexión de otros clientes.

Ambos conectores se conectan por un par de streams, uno en cada dirección, de entrada y de salida. Un proceso del par cliente-servidor puede enviar información al otro escribiendo en su stream de salida, y el otro puede obtener la información leyendo desde su stream de entrada.

Cuando un proceso finaliza o falla, todos sus conectores se cierran y cualquier proceso que se intente comunicar con él descubrirá que la conexión se ha roto.

* Concordancia de ítems de datos: Los dos procesos que se comunican deben estar de acuerdo con el tipo de datos que se transmiten por el stream
* Bloqueo: Cuando un proceso intenta leer datos de un canal de entrada, o bien se extraen los datos de la cola o se bloqueará hasta que existan datos disponibles.
* Hilos: Cuando un servidor acepta una conexión, generalmente crea un nuevo hilo con el que comunicarse en el nuevo cliente. El servidor se bloquea a la espera de entradas de un cliente sin afectar a los otros.

**Modelo de fallo**

Los streams TCP utilizan una suma de comprobación para detectar y rechazar paquetes corruptos, y utilizan un número de secuencia para paquetes duplicados. Utilizan timeouts y retransmisión de los paquetes perdidos. Los mensajes tienen garantizada su entregada cuando alguno de sus paquetes se haya perdido. No se garantiza la entrega de mensajes en presencia de algún tipo de problema.

**Utilización de TCP**

Servicios que se ejecutan sobre conexiones TCP con números de puerto reservados: HTTP, FTP, Telnet, SMTP.

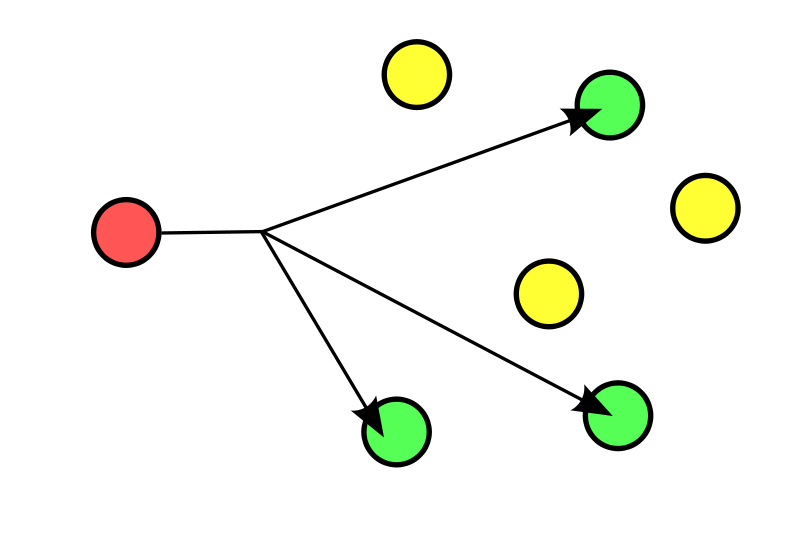
**API de Java para los streams TCP**

* Clase ServerSocket: Utilizada por un servidor para crear un conector en el puerto de servidor que escucha peticiones de conexión de los clientes. El accept toma una petición connect, la encola, y da como resultado una instancia de Socket, un conector que da acceso a los streams al cliente de entrada y salida.
* Clase Socket: El cliente utiliza un constructor para crear un conector, especificando el host y el puerto del servidor. Excepciones: UnknownHostException, IOException (entrada y salida).

Métodos getInputStream() y getOutputStream() para flujos lectura y escritura, devuelven objetos de tipo InputStream y OutputStream (clases abstractas para leer y escribir bytes). Objetos de tipo DataInputStream y DataOutputStream permiten utilizar representaciones binarias de los tipos de datos primitivos para ser leídos y escritos de forma independiente de la máquina.

Métodos writeUTF() y readUTF() para escribir y leer cadenas de caracteres en sus respectivos streams. Cuando un proceso ha cerrado su conector, no le será posible utilizar sus streams de entrada y salida.

**4.4 Comunicación multicast**

****

La comunicación multicast funciona mandando un mensaje de un proceso a otro a otro en un grupo de procesos. Por sí solo este protocolo no garantiza la entrega de mensajes o el orden en que estos llegan. El protocolo es útil en infraestructuras distribuidas las cuales necesitan de ciertas características tales como:

* Tolerancia a fallos en servicios replicados: Es decir que aunque la comunicación entre un cliente y el servidor falle, el servidor aun podrán servir a los demás clientes.
* Encontrar servicios en la red: Gracias al uso de mensajes multicast se es posible que los clientes y los servidores encuentren otros clientes o servidores para el uso de estos.
* Mejor rendimiento con datos replicados: En cada cambio en la información es posible que esta pueda ser replicada en cada proceso de comunicación.
* Propagación de notificaciones de eventos: Se le puede comunicar a un grupo entero de multicast un mismo mensaje.

**4.4.1 IP multicast**

IP multicast está construido en la parte superior del protocolo IP. Los paquetes IP están dirigidos a computadoras. Permite al emisor transmitir un único paquete IP a un conjunto de computadoras que forman un grupo multicast. El emisor no conoce las identidades de cada equipo ni el tamaño del grupo.

Permite a las computadoras unirse o abandonar en cualquier momento o a unirse a un número arbitrario de grupos. Es posible enviar datagramas a un grupo multicast sin ser miembro.

IP multicast solo esta disponible via UDP. Una aplicación multicast envía datagramas UDP con direcciones multicast y un número de puertos. Esta se puede unir a un grupo multicast haciendo que su socket se una al grupo y lo habilita para recibir mensajes del mismo. A nivel IP, una computadora pertenece a un grupo multicast cuando uno o más de sus procesos tienen sockets pertenecientes a este. Cuando un mensaje multicast es recibido, varias copias son reenviadas a todos los sockets locales que se han unido a la dirección multicast especificada y que están enlazados al número de puerto especificado.

* Routers multicast: Reenvían datagramas únicos hacia routers en otras redes, en donde son de nuevo retransmitidos a los miembros locales. El TTL (time to live) es el número de routers que el datagrama multicast tiene permitido saltar.

Asignación de direcciones de multicast:

Las direcciones de Clase D (es decir, las direcciones en el rango 224.0.0.0 a 239.255.255.255) están reservadas para el tráfico de multicast y administradas globalmente por la Autoridad de Números Asignados de Internet (IANA).

Las direcciones de multicast pueden ser permanentes o temporales. Los grupos permanentes existen incluso cuando no hay miembros: sus direcciones son asignadas por la IANA y abarcan los diversos bloques mencionados anteriormente. Por ejemplo, 224.0.1.1 en el bloque de Internet está reservado para el Protocolo de tiempo de red (NTP) y el rango 224.0.6.000 a 224.0.6.127 en el bloque ad hoc está reservado para el proyecto IS. El resto de las direcciones de multicast están disponibles para su uso por grupos temporales, que deben crearse antes de su uso y dejar de existir cuando todos los miembros se hayan ido. Cuando se crea un grupo temporal, se requiere una dirección de multicast gratuita para evitar la participación accidental en un grupo existente. El protocolo de multicast IP no aborda directamente este problema. Si se usa localmente, son posibles soluciones relativamente simples, por ejemplo, establecer el TTL en un valor pequeño, haciendo improbables las colisiones con otros grupos. Sin embargo, los programas que usan multicast IP en Internet requieren una solución más sofisticada para este problema. RFC 2908 [Thaler y col. 2000] describe una arquitectura de asignación de direcciones de multicast (MALLOC) para aplicaciones de Internet, que asigna direcciones únicas para un período de tiempo determinado y en un ámbito determinado. Como tal, la propuesta está intrínsecamente ligada a los mecanismos de alcance mencionados anteriormente. Se adopta una solución cliente-servidor mediante la cual los clientes solicitan una dirección de multicast de un servidor de asignación de direcciones de multicast (MAAS), que luego debe comunicarse entre dominios para garantizar que las asignaciones sean únicas para la vida útil y el alcance dados.

Modelo de falla  para datagramas multicast

Los datagramas multicast sobre IP multicast tienen las mismas características de falla que los datagramas UDP; es decir, sufren fallas por omisión. El efecto en una multicast es que no se garantiza que los mensajes se entreguen a ningún miembro del grupo en particular ante una sola falla de omisión. Es decir, algunos pero no todos los miembros del grupo pueden recibirlo. Esto se puede llamar multicast no confiable, porque no garantiza que se envíe un mensaje a ningún miembro de un grupo.

La API de Java proporciona una interfaz de datagrama para multicast IP a través de la clase MulticastSocket, que es una subclase de DatagramSocket con la capacidad adicional de poder unirse a grupos multicast. La clase MulticastSocket proporciona dos constructores alternativos, lo que permite crear sockets para usar un puerto local especificado o cualquier puerto local libre. Un proceso puede unirse a un grupo de multicast con una dirección de multicast dada invocando el método joinGroup de su socket de multicast. Efectivamente, el socket se une a un grupo de multicast en un puerto determinado y recibirá datagramas enviados por procesos en otras computadoras a ese grupo en ese puerto. Un proceso puede abandonar un grupo específico invocando el método leaveGroup de su socket multicast.

En la figura los argumentos del método main especifican que un mensaje sea multicast y la dirección de multicast de un grupo (por ejemplo, "228.5.6.7"). Después de unirse a ese grupo multicast, el proceso crea una instancia de DatagramPacket que contiene el mensaje y lo envía a través de su socket  multicast a la dirección del grupo multicast en el puerto 6789. Después de eso, intenta recibir tres mensajes de multicast de sus pares a través de su socket, que también pertenece al grupo en el mismo puerto. Cuando varias instancias de este programa se ejecutan simultáneamente en diferentes computadoras, todas ellas se unen al mismo grupo, y cada una de ellas debe recibir su propio mensaje y los mensajes de aquellos que se unieron después.

La API de Java permite configurar el TTL para un socket de multicast mediante el método setTimeToLive. El valor predeterminado es 1, lo que permite que la multicast se propague solo en la red local.

Una aplicación implementada sobre IP multicast puede usar más de un puerto. Por ejemplo, la aplicación MultiTalk [mbone], que permite a grupos de usuarios mantener conversaciones basadas en texto, tiene un puerto para enviar y recibir datos y otro para intercambiar datos de control.

**4.4.2 Fiabilidad y ordenación de multidifusión**

Esta sección habla de los posibles fallos en modelo IP para multidifusión. Se deben considerar los factores que pueden hacer que cualquier proceso fallé, por ejemplo, si un enrutador (router) multidifusión falla, el grupo de miembros más allá del enrutador no recibieron el mensaje de multidifusión. Aunque los miembros locales puedan hacerlo.

Por otra parte el ordenamiento también es otro problema. Lo paquetes IP mandamos por una internetwork no necesariamente llegan en orden en los que fueron enviados, con la posible efecto de que algunos miembros del grupo reciban datagramas desde un emisor único en diferente orden de otro grupo de miembros. También, los mensajes enviados de dos diferentes procesos no necesariamente llegarán en el mismo orden a todos los miembros del grupo.

1. *Tolerancia a fallos basado en servicios replicados:* Si se considera un servicio replicado que consiste en los miembros de un grupo de servidores que empiezan en el mismo estado inicial y siempre realizan las mismas operaciones en el mismo orden, de tal manera que se mantengan consistentes él unos con otros. Esta aplicación de multidifusión requiere que ya sea todas las réplicas o ninguna de ellas reciba cada petición, se volverán inconsistes con los otros. En la mayoría de los casos, el servicio requerirá que todos los miembros reciban mensajes de petición en el mismo orden uno como del otro.
2. *Descubriendo servicios en redes espontáneas:* Una forma en la cual un proceso descubre servicios de redes espontáneas es hacer una petición multidifusión en intervalos periódicos, y para los servicios disponibles se debe escuchar y responder estas peticiones.
3. *Mejor rendimiento mediante datos replicados:* Si se considera el caso donde los datos replicados en si, en lugar de operaciones en datos, son distribuidas por medio de mensajes de multidifusión. El efecto de los mensajes perdidos y le orden inconsistente dependerá en el método de replicación y la importancia de las réplicas estando todas actualizadas.

*Propagación de notificaciones de evento*: La aplicación en particular determina las cualidades requeridas de multidifusión.

**4.3 Representación externa de datos y empaquetado.**

La información que almacena un programa en ejecución es almacenada a través de estructuras de datos, sin embargo para comunicarse a través de un mensaje es necesario que estas sean convertidas a información pura, en otras palabras en bits para su reestructuración al llegar el mensaje a otro proceso. Debido a las diversas implementaciones de las estructuras de datos y los mismos datos, se deben utilizar técnicas para hacer que la información de un proceso a otro sea realmente la misma (Pueda ser utilizada de la misma forma). Estas son:

1. Se hace un acuerdo de utilizar un tipo de formato externo para transmisión y otro interno para recepción, si los sistemas fueran del mismo tipo, deberían mandar la información de forma nativa (Sin usar un formato externo).
2. El emisor avisa el tipo de formato que usará para que el receptor lo convierta a uno que pueda recibir solo de ser necesario.

Representación externa de datos: Estándar acordado para la representación de estructuras de datos y valores primitivos.

Empaquetado: Tomar una coleccion de items y ensamblarlos para ser enviados a través de un mensaje.

Desempaquetado: Uso de los datos del mensaje para ser convertidos en su forma original de estructura y valores primitivos.

Existen dos alternativas para la representación externa de datos y el empaquetado:

1. Representación mediante CORBA: Utilizando métodos en CORBA.
2. Serialización de objetos: Comúnmente usado en Java pero también existe para otros lenguajes de programación.

Ambas opciones son implementadas a través de un middleware sin participación del programador de la aplicación. El rendimiento y la fiabilidad son tomados en cuenta al hacer el empaquetado y desempaquetado. También se pueden hacer implementacion tan básicas como la comunicación usando ASCII.

**Empaquetado en CORBA**

Las operaciones de empaquetado se pueden generar automáticamente a partir de las especificaciones de los tipos de datos de los elementos que tienen que ser transmitidos. Los tipos de las estructuras de datos y de los elementos de datos básicos están descritos en CORBA IDL, que proporciona una notación para describir los tipos de los argumentos y los resultados de los métodos RMI.

**4.3.2 Serialización de Objetos en Java**

Declarar que una clase implementa la interfaz Serializable tiene el efecto de permitir que sus instancias sean serializables.

Serialización se refiere a la actividad de aplanar un objeto o un conjunto de objetos para obtener una forma lineal adecuada para ser almacenada en disco o para ser transmitida en un mensaje. La deserialización consiste en restablecer el estado de un objeto desde su estado lineal. Debe incluirse en la forma lineal alguna información sobre la clase de cada objeto, que se compone del nombre de la clase y un número de versión.

El número de versión está pensado para cambiar y reflejar cambios y modificaciones importantes en la clase. La serialización comprueba que se tiene la versión correcta de la clase.

Cuando un objeto es serializado, todos los objetos a los que referencia son serializados con él para asegurarse de que cuando el objeto sea reconstruido, todas sus referencias pueden ser rellenadas en el destino. Las referencias se serializan como apuntadores. Debe existir una correspondencia 1 - 1 entre las referencias y los apuntadores.

Para serializar un objeto, se escribe la información de su clase, los tipos y los nombres de los campos. Si los campos pertenecen a otra clase, se repite el proceso recursivamente. Cada clase recibe un apuntador, y ninguna clase es escribe más de una vez en el flujo de bytes.

La serialización y deserialización de los argumentos y los resultados de una invocación remota son llevados a cabo generalmente de forma automática por el middleware, sin ninguna participación del programador. Aquellas variables que no deberían ser serializadas se escriben como transient, como referencias a recursos locales (archivos o sockets).

**El uso de la reflexión**

La reflexión es la habilidad de preguntar sobre las propiedades de una clase. Hace posible que se creen las clases a partir de su nombre, y crea un constructor para una clase dada con unos argumentos dados. La serialización y deserialización es genérica: no hay necesidad de generar funciones de empaquetado especiales para cada tipo de objeto.

Con la reflexión, la serialización de objetos encuentra el nombre de la clase del objeto, los nombres, tipos y valores de sus variables de instancia. Para la deserialización se utiliza el nombre de la clase en la forma serializada para crear un nuevo constructor con los tipos de argumentos requeridos, con el cual se crea una nueva instancia del objeto con aquellos valores leídos de la forma serializada.

**4.3.3 Referencias a objetos remotos**

Una referencia a un objeto remoto es un identificador para un objeto remoto que es válida a lo largo y ancho del sistema distribuido. En el mensaje de invocación se incluya una referencia a un objeto remoto que especifica cual es el objeto invocado.

Las referencias a objetos remotos deben ser únicas entre todos los procesos en las computadoras de un sistema distribuido. Cualquier intento de invocar objetos borrados debería producir un error en lugar de permitir el acceso a un objeto diferente.

Para que una referencia a un objeto remoto sea única, se puede concatenar la dirección IP del computador y el número de puerto, junto con el instante de tiempo de su creación y un número de objeto local.

En RMI, los objetos remotos viven en el proceso que los crea y sobreviven mientras el proceso continúe en ejecución. Los mensajes de invocación se envían a la dirección IP de la referencia remota, y dentro del computador al proceso identificado por el número de puerto.

**4.4 Comunicación Cliente - Servidor**

La comunicación petición - respuesta es síncrona, ya que el proceso cliente se bloquea hasta que llega la respuesta del servidor. La respuesta del servidor es un acuse de recibo para el cliente. La comunicación cliente - servidor asíncrona es una alternativa que puede ser útil cuando los clientes puedan recuperar las respuestas más tarde.

El protocolo petición . respuesta, construido sobre datagramas, evita las sobrecargas asociadas con el protocolo TCP.

**El protocolo de petición-respuesta.** El siguiente protocolo está basado en un trio de primitivas de comunicación: hazOperacion, damePeticion, enviaRespuesta. Este protocolo de petición-respuesta, diseñado especialmente, hace corresponder a cada petición una respuesta. Podría estar diseñado para obtener ciertas garantías de entrega. Si se utilizan datagramas UDP, las garantías de entrega deben venir dadas por el protocolo petición-respuesta, donde el mensaje de respuesta del servidor sirve como acuse de recibo del mensaje de petición del cliente. La primitiva hazOperacion se utiliza en los clientes para invocar operaciones remotas. Su resultado es una respuesta RMI. El primer argumento de hazOperacion es una instancia de la clase RemoteObjectRef, que representa las referencias de los objetos remotos. Esta primitiva envía un mensaje de petición al servidor cuya dirección y puerto se especifican en la referencia de objeto remoto dada como argumento, después de hazOperacion se invoca el método recibe  para conseguir la respuesta y lo devuelve a su invocador.

La primitiva damePeticion se usa en el servidor para hacerse con las peticiones de servicio, Cuando el servidor ha invocado el método sobre el objeto especificado, utiliza el método enviarRespuesta para mandar la respuesta al cliente.

La información transmitida en un mensaje de petición se acomoda de la siguiente manera:

El primer campo indica si es petición o respuesta, el segundo tiene el identificador de mensaje, el tercero es la referencia al objeto remoto empaquetada, el cuarto campo es un identificador del método a invocar.

Identificadores de mensaje: lo utiliza cualquier esquema de gestión de mensajes para proporcionar propiedades adicionales.  Un identificador de mensajes tiene 2 partes:

1.      Un identificador de petición

2.      Un identificador para el proceso emisor

La primera parte hace que el identificador sea único para el emisor y la segunda lo hace único para el sistema distribuido

**Modelo de fallos del protocolo petición-respuesta**. Si las tres primitivas previamente mencionadas se implementan en datagramas UDP, adolecerán de los mismos fallos de comunicación que cualquier otro ejemplo de aplicación UDP. Esto es:

·        Sufrirán fallos de omisión

·        No se garantiza que los mensajes lleguen en el orden de emisión

**Tiempos de espera límite:** cuando hazOperacion supera un timeout, la opción más simple es devolver inmediatamente el control indicando al cliente que fracasó. Para compensar la posibilidad de la perdida de los mensajes hazOperacion manda el mensaje de petición de forma repetida hasta que obtiene una respuesta o está seguro que el retraso se debe a una falla del servidor.

Eliminación de mensajes de petición duplicados: cuando un mensaje de petición es retransmitido puede llevar al servidor a ejecutar una operación más de una vez, para evitarlo se diseñó al protocolo para reconocer mensajes sucesivos (del mismo cliente) con el mismo identificador de petición y eliminarlos.

**Pérdida de mensajes respuesta:** Si el servidor ya envió la respuesta y se recibe duplicada la petición, necesita volver a ejecutar la petición, a menos que haya almacenado el resultado original de la ejecución. Una operación idempotente es una operación que puede ser llevada a cabo repetidamente con el mismo efecto que si hubiera sido ejecutada una sola vez. Un servidor cuyas operaciones sean todas idempotentes no tendrán que tomar medidas especiales para evitar que se ejecuten más de una vez.

**Historial:** permite almacenar las respuestas enviadas a cada cliente sin tener que ejecutar las operaciones, contiene la respuesta, el identificador del mensaje y el identificador del cliente, el problema de esto es su costo de almacenamiento, el servidor debe decir cuando ya no es necesario almacenar un mensaje

**Protocolo de intercambio de RPC.** En la implementación de los distintos tipos de RPC se utilizan 3 protocolos:

·        El protocolo petición

·        El protocolo petición-respuesta

·        El protocolo petición-respuesta-confirmación de la respuesta  (RRA)

El  protocolo petición puede utilizarse cuando el cliente no necesita el resultado ni confirmación de la respuesta. El protocolo petición-respuesta, no se utilizan mensaje especiales de acuse ya que respuesta del servidor sirve como confirmación.

El protocolo  RRA está basado  en el intercambio de estos 3 mensajes: el mensaje de reconocimiento de la respuesta contiene el id Petición del mensaje de respuesta que reconoce, esto hace que el servidor pueda descartar las entradas de su historial. Aunque el intercambio implica un mensaje adicional, no necesita bloquear al cliente, ya que el reconocimiento puede transmitirse después de que la respuesta haya sido entregada al cliente.

**Utilización de streams TCP para implementar el protocolo petición-respuesta:** Una de las razones por la cual elegir la implementación de los protocolos petición-respuesta sobre streams TCP es el deseo de evitar la implementación de protocolos multipaquete, permitiendo la transmisión de argumentos y resultados de cualquier tamaño. Si se utiliza el protocolo TCP, se está asegurando que los mensajes de petición y de respuesta serán entregados de manera fiable, de modo que no será necesario un protocolo petición-respuesta para tratar los mensajes de retransmisión y filtrar los duplicados, y los historiales.

**HTTP (HyperText Transfer Protocol), un ejemplo de protocolo petición-respuesta**

Las peticiones de los clientes especifican un URL que incluye el nombre DNS del host del servidor web y un número de puerto *opcional,* además del identificador de un recurso en el servicio. Soporta un conjunto fijo de métodos (GET, PUT, POST, etc.) que son aplicables a todos los recursos. En este protocolo, cada objeto tiene sus propios métodos.

* *Negociación del contenido:* las peticiones de los clientes pueden incluir información sobre qué tipo de representación de datos pueden aceptar (por ejemplo lenguaje o tipo de medio), haciendo posible que el servidor pueda elegir la representación más apropiada para el usuario.
* *Autenticación:* se utilizan credenciales y desafíos para conseguir una autenticación del estilo *clave de acceso.*

Los recursos considerados como datos se proporcionan en forma de estructuras de tipo MIME tanto en los argumentos como en los resultados. ***Multipurpose Internet Mail Extension (Extensiones de Correo Electrónico Multipropósito, MIME)*** es un estándar para enviar mensajes de correo electrónico compuestos por varias partes conteniendo a la vez, por ejemplo, texto, imágenes y sonido.

**Métodos HTTP**

Cada petición de un cliente especifica el nombre de un método que habrá de ser aplicado al recurso en el servidor y el URL de dicho recurso. El mensaje de respuesta indica el estado de la petición. Los métodos considerados son los siguientes:

* ***GET:*** Pide el recurso cuyo URL se da como argumento. Se pueden añadir argumentos al URL; por ejemplo, un GET se puede utilizar para enviar el contenido de un formulario a un programa ***cgi*** que los tomará como entrada.
* ***HEAD:*** Esta petición es idéntica a GET, sólo que no devuelve datos. Sin embargo, devuelve toda la información sobre los datos, como el tiempo de la última modificación, su tipo o tamaño.
* ***POST:*** Especifica el URL de un recurso (por ejemplo un programa) que puede tratar los datos aportados con la petición. Este método está diseñado para:   
  + Proporcionar un bloque de datos (por ejemplo los obtenidos en un formulario) a un proceso de gestión de datos como un servlet o un programa *cgi.*
  + Enviar un mensaje a un tablón de anuncios, lista de correo o grupo de noticias.
  + Modificar una base de datos con una operación de añadir registro.
* ***PUT:*** Indica que los datos aportados en la petición deben ser almacenados con la URL aportada como su identificador, ya sea como una modificación de datos existentes o como la creación de un recurso nuevo
* ***DELETE:*** El servidor borrará el recurso identificado por el URL El servidor no siempre permitirá esta función, en cuyo caso se devolverá una indicación de fallo.
* ***OPTIONS:*** El servidor proporciona al cliente una lista de métodos aplicables a un URL (por ejemplo, GET,HEAD,PUT) y sus requisitos especiales.
* ***TRACE:*** El servidor envía de vuelta el mensaje de petición. Se utiliza en procesos de depuración.

**Contenido del mensaje:** Cada mensaje HTTP *request* especifica el nombre de un método, el URL de un recurso, la versión del protocolo, algunas cabeceras y un cuerpo de mensaje opcional. Los campos de la cabecera contienen modificadores de la petición e información sobre el cliente, como las condiciones sobre la última fecha de modificación del recurso o los tipos de contenido aceptables.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Metodo* | URL | version HTTP | Cabeceras | Cuerpo del mensaje |
| GET | //www.dcs.qmw.ac.uk/index.html | HTTP/1.1 |  |  |
| Mensaje HTTP Request | | | | |

Un mensaje *Reply* especifica la versión del protocolo, un código de estado y su razón, algunas cabeceras y un cuerpo de mesaje opcional. El código de estado y la razón proporcionan un informe sobre el éxito o cualquier otra situación asociada al cumplimiento de la petición.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Versión HTTP* | Código de estado | razón | Cabeceras | Cuerpo del mensaje |
| GET | 200 | OK |  |  |
| Mensaje HTTP Reply | | | | |

El cuerpo del mensaje en los mensajes de petición o en los de respuesta contienen los datos asociados con la URL especificada en la petición. El cuerpo del mensaje tiene sus propias cabeceras especificando información sobre los datos, como la longitud, el tipo MIME, el conjunto de caracteres, la codificación del contenido y la último fecha de modificación.