



Final

▼ Unidad 1: Caracterización de los Sistemas Distribuidos

▼ Sistema Distribuido

Un **sistema distribuido** es una colección de computadoras independientes que aparecen ante los usuarios del sistema como una única computadora. (Tanenbaum).

Un sistema distribuido es aquel en el que los componentes localizados en computadores, conectados en red, comunican y coordinan sus acciones únicamente mediante el paso de mensajes. (Coulouris)

▼ Recursos

Compartir recursos es uno de los motivos principales para construir sistemas distribuidos.

Los recursos son administrados por servidores y accedidos por clientes. O encapsulados como objetos y accedidos por otros objetos clientes.

Recursos pueden ser componentes hardware como discos, impresoras, o entidades software como archivos, bases de datos y objetos.

▼ Características de los Sistemas Distribuidos

- **Concurrencia:** en una red de computadoras, la ejecución de programas concurrentes es la norma. La coordinación de programas que comparten recursos y se ejecutan de forma concurrente es un tema importante y recurrente.
- **Inexistencia de reloj global:** la cooperación entre programas se realiza mediante el intercambio de mensajes. La coordinación depende de una idea compartida del instante en el que ocurren las “acciones” de los

programas. Existen límites de precisión en los computadores de una red para sincronizar sus relojes. No existe una única noción global del tiempo correcto.

- **Fallos independientes:** todos los sistemas pueden fallar. Los sistemas distribuidos pueden fallar de formas diferentes:
 - Fallos en la red. Producen aislamiento de los computadores conectados a él, peor eso no significa que detengan su ejecución. Pueden no ser capaces de detectar cuando la red ha fallado o está excesivamente lenta.
 - Parada de un computador o terminación inesperada de un programa (crash). No se da a conocer inmediatamente a los demás componentes con los que se comunica.

▼ Ventajas de los Sistemas Distribuidos

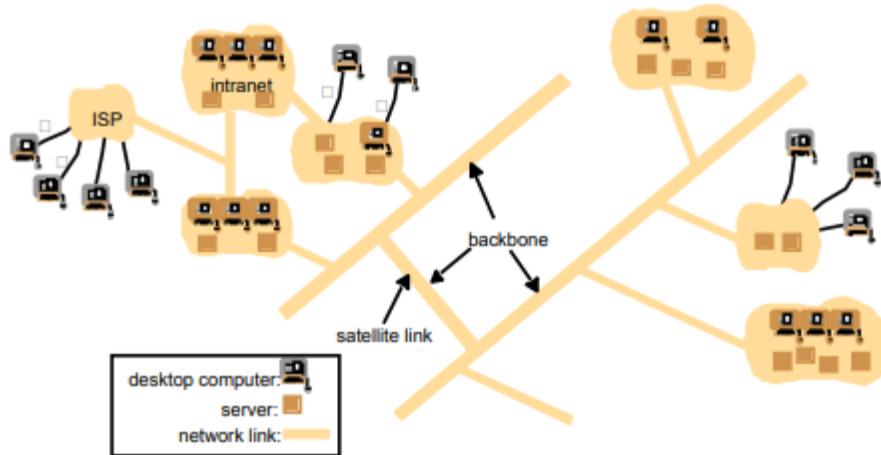
- **Economía.** Los microprocesadores ofrecen mejor proporción precio/rendimiento que los mainframes.
- **Velocidad.** Un sistema distribuido puede tener mayor poder de cómputo que un mainframe.
- **Distribución inherente.** Algunas aplicaciones utilizan máquinas que están separadas a cierta distancia.
- **Confiabilidad.** Si una máquina se descompone, el sistema puede sobrevivir como un todo.
- **Crecimiento por incrementos.** Se puede añadir poder de cómputo en pequeños incrementos.

▼ Ejemplos de Sistemas Distribuidos

▼ Internet

El conjunto de servicios es abierto, puede ser extendido por la adición de servidores y nuevos tipos de servicios.

Una topología típica de internet.



La figura nos muestra una colección de intranets, subredes gestionadas por compañías y otras organizaciones. Los proveedores de servicios de Internet (ISP's) son empresas que proporcionan enlaces de módem y otros tipos de conexión a los usuarios, permitiéndoles el acceso a servicios como correo electrónico y páginas web. Las intranets están enlazadas conjuntamente por conexiones troncales (backbones). Una conexión o red troncal es un enlace de red con una gran capacidad de transmisión, que puede emplear conexiones de satélite, cables de fibra óptica y otros circuitos de gran ancho de banda.

▼ Buscadores Web

La tarea de un motor de búsqueda web es indexar el contenido completo de la World Wide Web, esta es una tarea muy compleja, ya que la Web consiste en más de **63 mil millones de páginas y un trillón de direcciones web únicas**.

Google, líder de mercado en tecnología de búsqueda web, ha puesto un esfuerzo significativo en el diseño de una sofisticada infraestructura de sistemas distribuidos para apoyar la búsqueda. Esto representa una de las mayores y más complejas instalaciones de sistemas distribuidos de la historia de la informática.

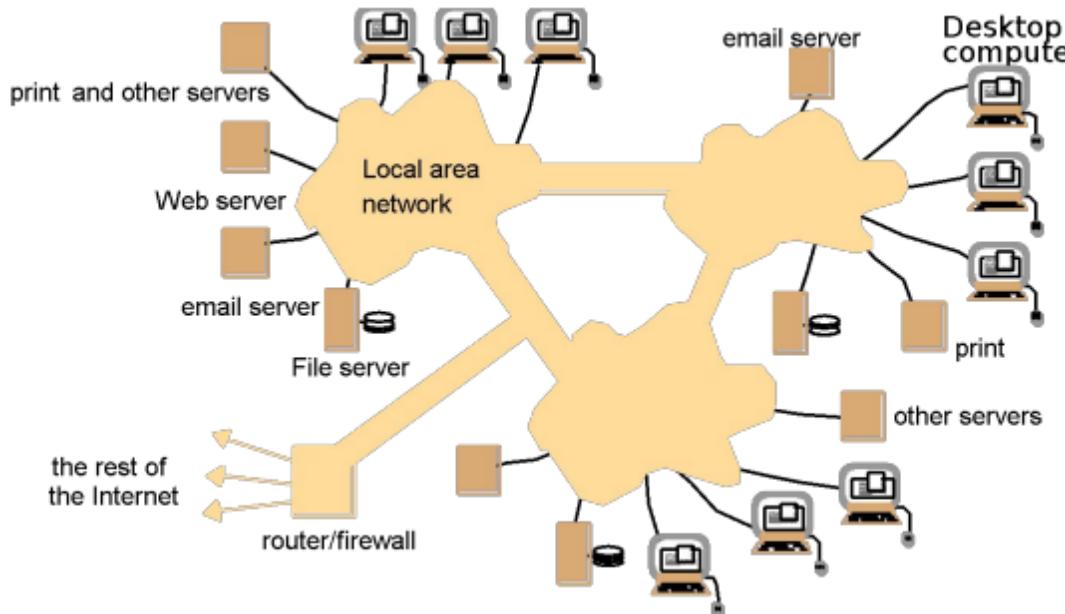
Los puntos destacados de esta infraestructura incluyen:

- **Infraestructura física subyacente** que consta de un gran número de ordenadores en red ubicados en centros de datos de todo el mundo.
- **Sistema de archivos distribuido** diseñado para soportar archivos muy grandes y muy optimizado para el estilo de uso requerido.
- **Sistema de almacenamiento distribuido** que ofrece acceso rápido a conjuntos de datos grandes.
- **Servicio de bloqueo** que ofrece funciones de sistema tales como bloqueo distribuido y acuerdo.
- **Modelo de programación** que soporta la gestión de cálculos paralelos y distribuidos muy grandes a través de la infraestructura física subyacente.

▼ Intranet

Una intranet es una porción de Internet que es, administrada separadamente y que tiene un límite que puede ser configurado.

Una intranet típica.

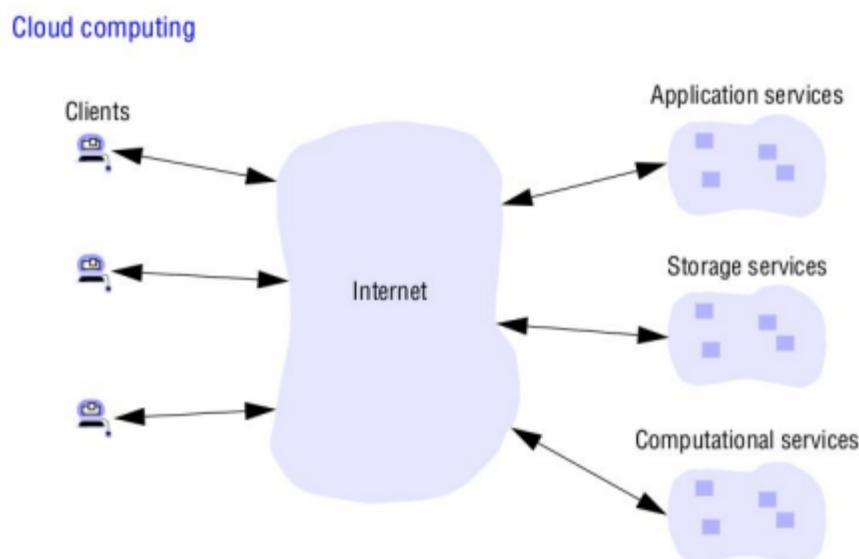


Esta compuesta de varias LANS enlazadas por conexiones backbone. La configuración de red de una intranet es responsabilidad de la organización que la administra. Una intranet está conectada a Internet por medio de un

encaminador (router), lo que permite a los usuarios hacer uso de servicios de otro sitio como la Web o el correo electrónico.

▼ Tendencias

- **Cloud Computing.** El término se utiliza para capturar esta visión de la **computación como una utilidad**. Una nube se define como un conjunto de servicios de aplicaciones y almacenamiento suficientes para soportar la mayoría de las necesidades de los usuarios. El término también promueve una visión de todo como un servicio, desde la infraestructura física o virtual hasta el software. La computación en la nube reduce los requisitos de los dispositivos de los usuarios.



- **Computación móvil y ubicua.** Los avances tecnológicos en la miniaturización de dispositivos y redes han llevado cada vez más a la integración de dispositivos pequeños y portátiles en los sistemas distribuidos.
 - **Computación móvil:** se denomina a la realización de tareas de cómputo mientras el usuario está en movimiento. Los usuarios que están fuera de su hogar Intranet disponen de la posibilidad de acceder a los recursos mediante los dispositivos que llevan con ellos.

- **Computación ubicua:** la integración de la computación en el entorno de la persona, de forma que los ordenadores no se perciban como objetos diferenciados. El término ubicuo está pensado para sugerir que los pequeños dispositivos llegarán a estar tan extendidos en los objetos de cada día que apenas nos daremos cuenta de ellos.

▼ Desafíos de los Sistemas Distribuidos

▼ Heterogeneidad

Internet permite que los usuarios accedan a servicios y ejecuten aplicaciones sobre un conjunto heterogéneo (variado y diferenciado) de redes y computadores. Esto se aplica a las redes, hardware, sistemas operativos, lenguajes de programación, implementaciones.

- **Middleware:** software que provee una abstracción de programación, así como un enmascaramiento de la heterogeneidad. Generalmente se implementa sobre protocolos de Internet, enmascarando éstos la diversidad de redes existentes.
- **Código móvil:** código que puede ser enviado desde un computador a otro y ejecutarse en este. Ejemplo: applets de Java. El conjunto de instrucciones de un computador depende del hardware. Ejemplo, ejecutables para PC no pueden ser ejecutados en Mac.

▼ Extensibilidad

Es la característica que determina si el sistema puede ser extendido y reimplementado en diferentes aspectos. La extensibilidad de los sistemas distribuidos se determina en primer lugar por el grado en el cual se pueden añadir nuevos servicios y ponerlos a disposición para el uso.

Los sistemas abiertos se caracterizan porque sus interfaces están publicadas. Los sistemas distribuidos abiertos se basan en la provisión de un mecanismo de comunicación uniforme e interfaces públicas para acceder a recursos compartidos. Pueden construirse con hardware y software heterogéneo.

▼ Seguridad

Entre los recursos de información que se ofrecen y se mantienen en los sistemas operativos, muchos tienen un alto valor para sus usuarios, por esto su seguridad es de considerable importancia.

La seguridad de los recursos de información tienen tres componentes:

Confidencialidad: protección contra el descubrimiento por individuos no autorizados.

Integridad: protección contra la alteración o corrupción.

Disponibilidad: protección contra interferencia con los procedimientos de acceso.

▼ Escalabilidad

Los sistemas distribuidos operan efectiva y eficientemente en muchas escalas diferentes. **Se dice que un sistema es escalable si conserva su efectividad cuando ocurre un incremento significativo en el número de recursos y usuarios.**

El diseño de los sistemas distribuidos escalables presenta los siguientes retos:

- **Control del costo de los recursos físicos:** según crece la demanda de un recurso, debiera ser posible extender el sistema, a un coste razonable, para satisfacerla.
- **Control de las pérdidas de prestaciones:** administración de un conjuntos de datos cuyo tamaño es proporcional al número de usuarios o recursos del sistema.
- **Prevención de desbordamiento de recursos software:** un ejemplo de pérdida de escalabilidad se muestra en el tipo de número usado para las direcciones Internet.
- **Evitar cuellos de botella de prestaciones:** para evitar cuellos de botella de prestaciones, los algoritmos deberían ser descentralizados.

▼ Tratamiento de Fallos

- **Detección de fallos:** algunos fallos son detectables, por ejemplo, se pueden utilizar checksums para detectar datos corruptos en un

mensaje o un archivo.

- **Enmascaramiento de fallos:** algunos fallos que han sido detectados pueden ocultarse o atenuarse.
- **Tolerancia de fallos:** La mayoría de los servicios tienen fallos, es posible que no sea práctico detectar y ocultar todos los fallos que pudieran aparecer. Sus clientes pueden diseñarse para tolerar ciertos fallos, lo que implica que los usuarios también tendrán que tolerarlos.
- **Recuperación frente a fallos:** la recuperación implica que, tras una caída del servidor, el estado de los datos pueda reponerse o retractarse a una situación anterior.
- **Redundancia:** puede lograrse que los servicios toleren fallos mediante el empleo redundante de componentes.

▼ **Concurrencia**

Tanto los servicios como las aplicaciones proporcionan recursos que pueden compartirse entre los clientes en un sistema distribuido. Existe por lo tanto la posibilidad de que varios clientes intenten acceder a un recurso compartido a la vez.

El proceso que administra un recurso compartido puede atender las peticiones una por una en cada momento, pero esta aproximación limita el throughput. Por esto, los servicios y aplicaciones permiten procesar concurrentemente múltiples peticiones de los clientes. Normalmente mediante el uso de hilos.

▼ **Transparencia**

- **Transparencia de acceso** que permite acceder a los recursos locales y remotos empleando operaciones idénticas.
- **Transparencia de ubicación** que permite acceder a los recursos sin conocer su localización.
- **Transparencia de concurrencia** que permite que varios procesos operen concurrentemente sobre recursos compartidos sin interferencia mutua.

- **Transparencia de replicación** que permite utilizar múltiples ejemplares de cada recurso para aumentar la fiabilidad y las prestaciones.
- **Transparencia frente a fallos** que permite ocultar fallos, dejando que los usuarios y programas completen su tarea a pesar de los fallos del hardware o del software.
- **Transparencia de movilidad** que permite la reubicación de recursos y clientes en un sistema sin afectar la operación de los usuarios y los programas.
- **Transparencia de prestaciones** que permite reconfigurar el sistema para mejorar las prestaciones según varía su carga.
- **Transparencia al escalado** que permite al sistema y a las aplicaciones expandirse en tamaño sin cambiar la estructura del sistema.

▼ Unidad 2: Modelos de Sistemas

▼ Introducción

Modelos Físicos: es la forma más explícita de describir un sistema, incluye la composición de hardware de un sistema en términos de los ordenadores y sus redes de interconexión.

Modelos Arquitectónicos: describen un sistema en términos de las tareas de cálculo y de comunicación realizadas por sus elementos computacionales.

Modelos Fundamentales: perspectivas abstractas con el fin de examinar los aspectos individuales de un sistema distribuido.

Estas cuestiones se consideran en tres modelos:

- **El modelo de interacción** trata de las prestaciones y de la dificultad de poner límites temporales en un sistema distribuido, por ejemplo para la entrega de mensajes.

- **El modelo de fallos** intenta dar una especificación precisa de los fallos que se pueden producir en los procesos y en los canales de comunicación. Define comunicación fiable y procesos correctos.
- **El modelo de seguridad** discute sobre las posibles amenazas para los procesos y los canales de comunicación. Introduce el concepto de canal seguro.

▼ Modelos Físicos

Un **Modelo Físico** es una representación de los elementos de hardware de un sistema distribuido que abstrae los detalles específicos de las tecnologías informáticas y las redes empleadas.

Tres generaciones de sistemas distribuidos:

- **Primeros Sistemas Distribuidos:** emergieron entre los 70' y 80' en respuesta a la emergente tecnología LAN, específicamente Ethernet. Típicamente entre 10 a 100 nodos conectados por medio de una LAN.
- **Sistemas Distribuidos en la escala de Internet:** surgieron en la década de los 90' en respuesta al impresionante crecimiento de Internet. Consiste en un conjunto de nodos interconectados por una red de redes.
- **Sistemas Distribuidos Contemporáneos:** aparición de la computación móvil en la cual los nodos comprenden computadores personales, asistentes digitales, teléfonos móviles. La aparición de la computación en Nube, computación ubicua y tecnologías variadas de red, así como infinidad de aplicaciones hacen la cantidad de expanda increíblemente.

▼ Modelos Arquitectónicos

La arquitectura de un sistema es su estructura en términos de componentes especificados por separado. El objetivo global es asegurar que la estructura satisfará las demandas presentes y previsibles sobre él.

Un modelo arquitectónico de un sistema distribuido **simplifica y abstrae** las funciones de los componentes individuales de dicho sistema y posteriormente considera, la ubicación de los componentes en una red, y las interrelaciones entre los componentes.

▼ Arquitectura de Software

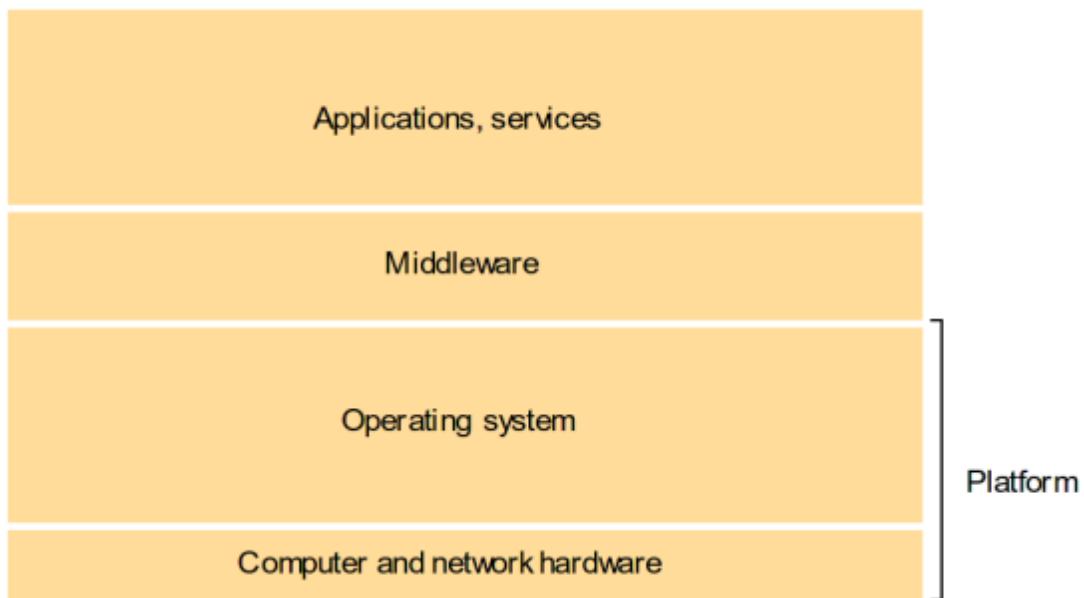
El término “arquitectura de software” se refería inicialmente a la estructura del software como capas o módulos en un único computador. Más recientemente en términos de los servicios ofrecidos y solicitados entre procesos.

Esta vista orientada a proceso y a servicio puede expresarse en términos de capas de servicio.

Un servidor es un proceso que acepta peticiones de otros procesos.

Un servicio distribuido puede proveerse desde uno o más procesos servidor, interactuando entre cada uno de ellos, y con los procesos clientes, para mantener una visión de los recursos del servicio consistente con el sistema.

▼ Patrones de Arquitectura



- **Plataforma.** Al nivel de hardware y las capas más bajas de software se denominan “plataforma” para sistemas distribuidos y aplicaciones. Estas capas más bajas proporcionan servicios a las que están por encima de ellas, y que son implementadas independientemente en cada computador, proporcionando una interfaz de programación del sistema.

- **Middleware.** Es una capa de software cuyo propósito es enmascarar la heterogeneidad y proporcionar un modelo de programación conveniente para los programadores de aplicaciones. Se representa mediante procesos u objetos en un conjunto de computadores que interactúan entre sí para implementar mecanismos de comunicación y de recursos compartidos para aplicaciones distribuidas.

El middleware se ocupa de proporcionar bloques útiles para la construcción de componentes software que puedan trabajar con otros en un sistema distribuido.

Mejora el nivel de las actividades de comunicación soportando abstracciones como: procedimiento de invocación remota, comunicación entre un grupo de procesos, notificación de eventos, replicación de datos compartidos y transmisión de datos multimedia en tiempo real.

▼ Roles y Responsabilidades

El aspecto importante en el diseño de sistemas distribuidos es la división de responsabilidades entre los componentes del sistema.

Consideraciones en la arquitectura:

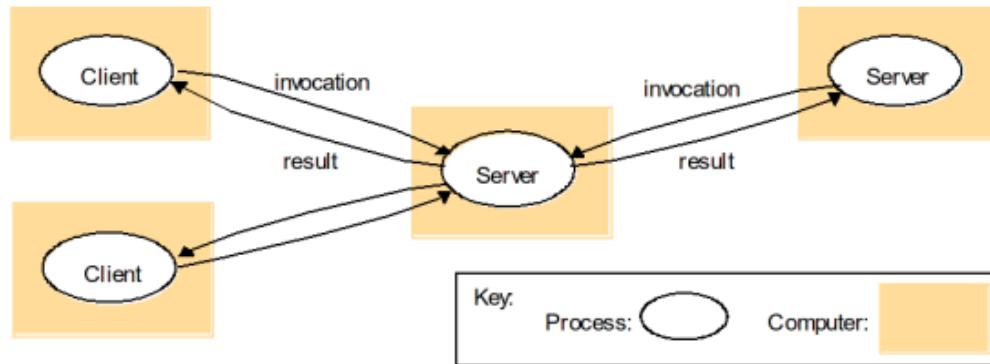
- La partición de datos o la replicación en servidores cooperativos.
- El uso de caché para los datos en clientes y servidores proxy.
- El uso de código y agentes móviles.
- Los requisitos para añadir o eliminar dispositivos móviles de forma conveniente.

No hay un tiempo global en un sistema distribuido. Los relojes de los computadores no tienen necesariamente el mismo tiempo. Toda comunicación entre procesos se realiza por medio de mensajes. La comunicación en una red puede verse afectada por: retrasos, puede sufrir variedad de fallos y es vulnerable a los ataques de seguridad.

▼ Modelo cliente-servidor

Históricamente la arquitectura cliente-servidor es la más importante, y continúa siendo la más ampliamente utilizada.

Clients que invocan a servidores individuales.



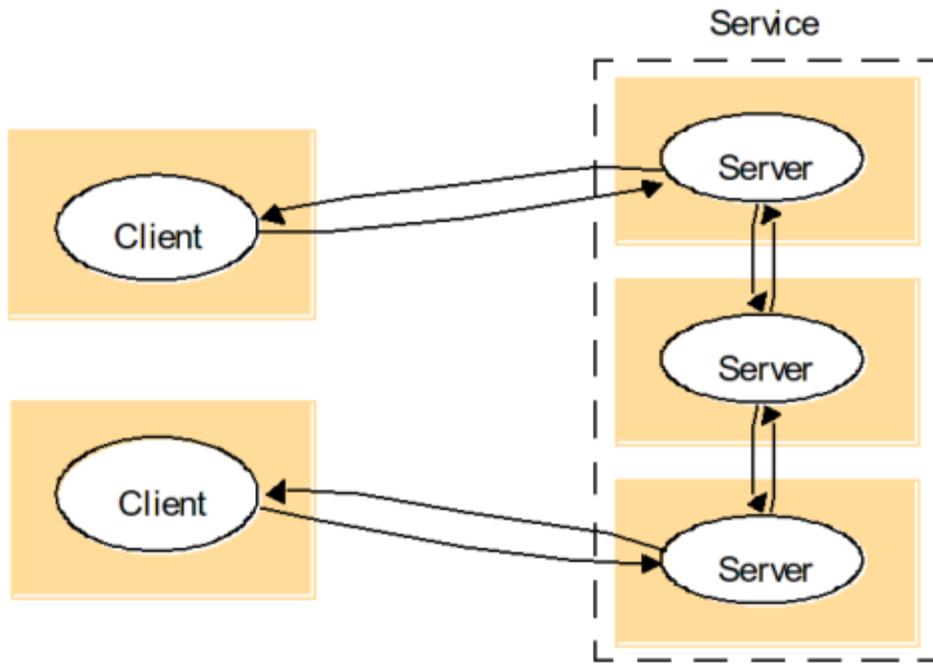
Sencilla estructura sobre la que interaccionan los procesos clientes con los procesos servidores individuales, en computadores separados, con el fin de acceder a los recursos compartidos que ellos gestionan.

Los servidores pueden, a su vez, ser clientes de otros servidores. Por ejemplo, un servidor web es a veces un cliente de un servidor de archivos local.

Los servidores web y la mayoría del resto de los servicios de internet son clientes del servicio DNS.

Una máquina de búsqueda es tanto un cliente como un servidor: responde a las consultas del navegador de los clientes y ejecuta web crawlers que actúan como clientes de otros servidores web.

▼ Servicios proporcionados por múltiples servidores



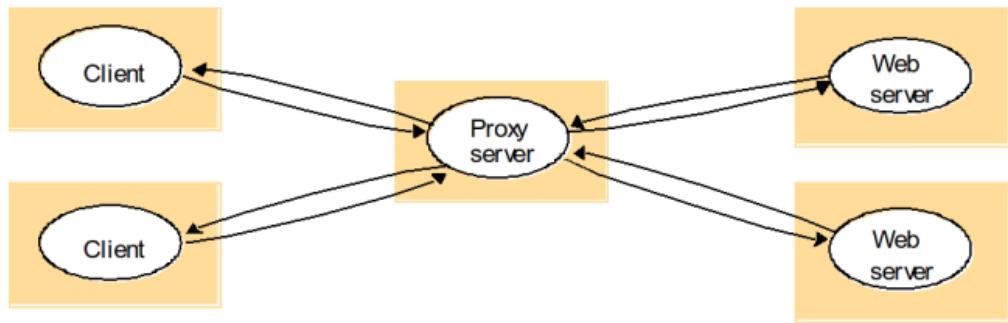
Los servicios pueden implementarse como distintos procesos de servidor en computadores separados interactuando para proporcionar un servicio a los procesos clientes.

Los servidores pueden dividir el conjunto de objetos en los que está basado el servicio y distribuirselos entre ellos mismos, o pueden mantener copias replicadas de ellos en varias máquinas.

Partición: La web es un ejemplo típico de partición de datos. Cada servidor web administra su propio conjunto de recursos. Un usuario puede emplear un navegador para acceder al recurso que está en cualquiera de los servidores.

Replicación: se utiliza para aumentar las prestaciones y disponibilidad y para mejorar la tolerancia de fallos. Proporciona múltiples copias consistentes de datos que se ejecutan en diferentes computadoras.

▼ Servidores proxy y cachés



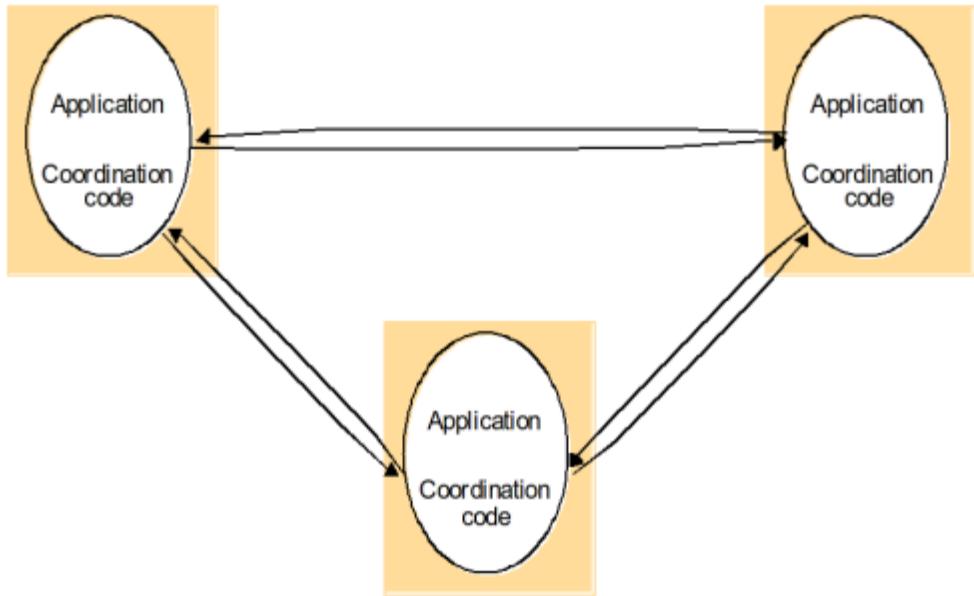
Un caché es un almacén de objetos de datos utilizados recientemente, y que se encuentra más próximo que los objetos en sí. Al recibir un objeto nuevo en un computador se añade al almacén del caché, reemplazando si fuera necesario algunos objetos existentes.

Cuando se necesita un objeto en un proceso cliente, el servicio caché comprueba inicialmente la caché y le proporciona el objeto de una copia actualizada. Si no, se buscará una copia actualizada. Los cachés pueden estar ubicados en cada cliente o en un servidor proxy que puede compartirse desde varios clientes.

Los servidores proxy para la web proporcionan un caché compartido de recursos web a las máquinas clientes de uno o más sitios.

El propósito de los servidores proxy es incrementar la **disponibilidad** y **prestaciones** del servicio.

▼ Procesos de igual a igual. Peer to Peer



En esta arquitectura todos los procesos desempeñan tareas semejantes, interactuando cooperativamente como iguales para realizar una actividad distribuida o cómputo sin distinción entre clientes y servidores.

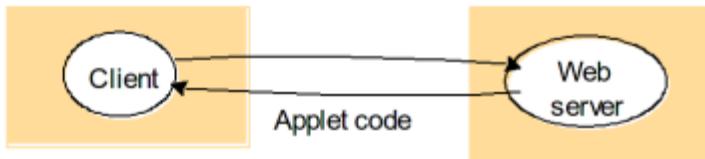
En este modelo, el código en los procesos parejos o iguales mantiene la consistencia de los recursos y sincroniza las acciones cuando es necesario.

La eliminación del proceso servidor reduce los retardos de comunicación entre procesos al acceder a objetos locales.

Una aplicación de pizarra distribuida, que permite que usuarios en varios computadores vean y modifiquen interactivamente un dibujo, puede implementarse con un proceso de aplicación en cada sitio, que se basa en capas de middleware para realizar **notificación de eventos y comunicación a grupos** para indicar a todos los procesos de los cambios en el dibujo.

▼ Variaciones en el modelo cliente-servidor

a) client request results in the downloading of applet code



b) client interacts with the applet



▼ Código móvil

Los applets son el conjunto más conocido de código móvil; **un usuario que ejecute un navegador y seleccione un enlace con un applet, descargará el código en el navegador y se ejecutará allí.**

Una ventaja de esto es que puede proporcionar una buena respuesta interactiva puesto que no sufre ni de los retardos ni de la variabilidad de ancho de banda.

Consideremos una aplicación donde se necesite que los usuarios dispongan de información actualizada según se hagan cambios en el origen de los datos en el servidor. Esto no se puede conseguir mediante las interacciones normales con el servidor web, que siempre se inician por parte del cliente. La solución es utilizar software adicional que trabaja de una forma a la que se le refiere como **modelo push**, donde el servidor, en lugar del cliente, es el que inicia la interacción. Por ejemplo, para un servicio de notificación de cambios en bolsa, los usuarios deberían descargar un applet especial, que recibe las actualizaciones del servidor.

▼ Agentes móviles

Es un programa en ejecución que se traslada de un computador a otro en la red realizando una tarea para alguien; por ejemplo,

recolectando información, y retornando eventualmente con los resultados.

Los agentes móviles son una amenaza potencial de seguridad para los recursos de los computadores que visitan. El entorno debe decidir que recursos locales estará permitido el acceso. Los agentes móviles pueden ser vulnerables, y pueden no ser capaces de finalizar su tarea si se les niega el acceso a la información que necesitan.

Un ejemplo de Agente Móvil es el programa gusano worm desarrollado con el fin de utilizar computadores desocupados para realizar cálculos intensivos. Se pueden utilizar agentes móviles para instalar y mantener software en los computadores de una organización.

▼ Clientes ligeros

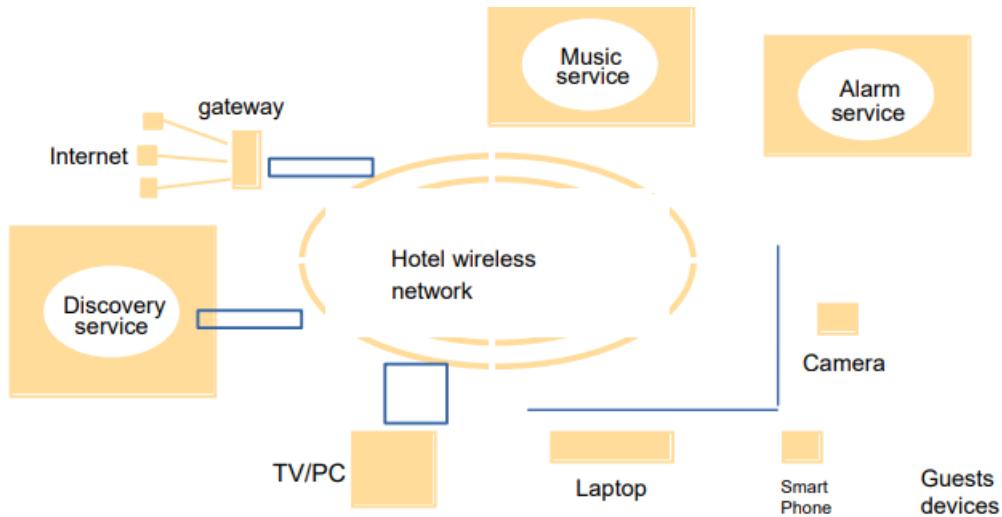
El término cliente ligero se refiere a una capa de aplicación que soporta una interfaz de usuario basada en ventanas sobre un computador local del usuario mientras se ejecutan programas de aplicación en un computador remoto.

Las aplicaciones se ejecutan en un servidor de cómputo, un potente computador que tiene capacidad para ejecutar gran número de aplicaciones simultáneamente.

El principal inconveniente de una arquitectura de cliente ligero se encuentra en las actividades gráficas fuertemente interactivas, en la que los retrasos experimentados por los usuarios se ven aumentados por la necesidad de transferir imágenes e información vectorial entre el cliente ligero y los procesos de aplicación, padeciendo latencias tanto de red como de sistema operativo.

Implementaciones de clientes ligeros: WinFrame de Citrix es una implementación comercial del concepto de cliente ligero. Puede ejecutarse en una gran variedad de plataformas y soporta un entorno (desktop) que proporciona acceso interactivo a las aplicaciones corriendo en máquinas Windows NT.

▼ Dispositivos móviles y enlace espontáneo



La figura muestra los dispositivos que el usuario ha llevado a la suite del hotel un computador portátil, una cámara digital y una agenda portátil (PDA), cuyas posibilidades a través de interfaces inalámbricas le permiten actuar como un controlador universal.

Las características esenciales de la conexión a red espontánea son:

- **Conexión fácil a la red local:** los enlaces sin cable eliminan la necesidad de cableado preinstalado y eliminan el inconveniente y las cuestiones de fiabilidad que rodean a los conectores y enchufes. Un dispositivo nuevo se reconfigura de modo transparente; el usuario no tiene que introducir los nombres o direcciones de los servicios locales para obtenerlo.
- **Integración fácil con servicios locales:** los dispositivos que se encuentran insertados y moviéndose entre redes existentes de dispositivos descubren automáticamente qué servicios se proporcionan, sin acciones especiales de configuración por el usuario.

Para usuarios móviles se presentan otras cuestiones:

- **Conectividad limitada:** los usuarios no siempre están conectados cuando se desplazan. Por ejemplo, al viajar en un tren por un túnel pueden desconectarse eventualmente de su red inalámbrica. También pueden desconectarse durante largos períodos de tiempo en zonas sin cobertura.
- **Seguridad y privacidad:** en un escenario como el del huésped de un hotel se presentan muchas cuestiones de seguridad y privacidad personal. El hotel y sus huéspedes son vulnerables a los ataques de seguridad.

▼ Requisitos de diseño para arquitecturas distribuidas

- **Prestaciones.** Los retos que surgen de la distribución de recursos aumentan la necesidad de gestión de actualizaciones concurrentes. Los temas de prestaciones se consideran bajo:
 - **Capacidad de respuesta (Responsiveness):** los usuarios de aplicaciones interactivas necesitan rapidez y consistencia en las interfaces. A menudo, los programas cliente necesitan acceder a recursos compartidos.
 - **Productividad (Throughput):** Una medida tradicional de prestaciones para computadores es la productividad, la rapidez a la que se realiza el trabajo computacional.
 - **Balance de cargas computacionales:** uno de los propósitos de los sistemas distribuidos es permitir que las aplicaciones y los procesos de servicio evolucionen concurrentemente sin competir por los mismos recursos y explotando los recursos computacionales disponibles.
- **Calidad de servicio:** las principales propiedades no funcionales de los sistemas, que afectan a la calidad del servicio experimentado por los clientes y usuarios, son: fiabilidad, seguridad y prestaciones.
- **Aspecto de fiabilidad:**
 - **Tolerancia frente a fallos:** Las aplicaciones estables deben seguir funcionando aún en presencia de fallos de hardware, software y

redes.

- **Seguridad:** Ubicar y proteger datos/recursos sensibles contra ataques.

▼ Modelos Fundamentales

- **Interacción:** el computo ocurre en los procesos; los procesos interaccionan por paso de mensajes, lo que resulta en comunicación y coordinación entre procesos.
- **Fallo:** la correcta operación de un sistema distribuido se ve amenazada donde aparezca un fallo en cualquier computador sobre el que se ejecuta o en la red que los conecta.
- **Seguridad:** la naturaleza modular de los sistemas distribuidos y su extensibilidad los expone a ataques tanto de agentes externos como internos.

▼ Modelo de Interacción

▼ Factores que afectan a los procesos de un SD

- **Prestaciones de los canales de comunicación:** las características de prestaciones de la comunicación sobre una red de computadores incluyen **la latencia, el ancho de banda y las fluctuaciones:**
 - El retardo entre el envío de un mensaje por un proceso y su recepción por otro se denomina **latencia**. La latencia incluye:
 - El tiempo que se toma en llegar a su destino el primero de una trama de bits.
 - El retardo en acceder a la red.
 - El tiempo empleado por los servicios de comunicación del sistema operativo tanto en el proceso que envía como en el que recibe.
 - **El ancho de banda** en una red de computadores es la cantidad total de información que puede transmitirse en un intervalo dado. Cuando una red está siendo utilizada por un número grande de canales de comunicación, habrá que compartir el ancho de banda disponible.

- La **fluctuación (jitter)** es la variación en el tiempo invertido en completar la entrega de una serie de mensajes. Es importante para los datos multimedia.
- **Reloj de computadores y eventos de temporización:** cada computador de un sistema distribuido tiene su propio reloj interno; los procesos locales obtienen de él, el valor del tiempo actual.

Esta es la forma en la que dos procesos sobre dos computadores diferentes asocian marcas temporales a sus eventos. Sin embargo, siempre sus relojes locales proporcionarán valores de tiempo diferentes. Esto ocurre porque los relojes presentan derivas con respecto al tiempo perfecto y, más importante, sus tasas de deriva difieren de una a otra.
- **Tasa de deriva del reloj (clock drift rate):** proporción en el que el reloj de un computador difiere del reloj de referencia absoluto. Incluso, si los relojes de todos los computadores de un sistema distribuido se ponen en hora a la vez, a partir de un tiempo sus relojes variarán en una magnitud significativa a menos que se apliquen correcciones.

▼ Variantes

- **Dos variantes del modelo de interacción.** en un sd es difícil establecer cotas sobre el tiempo que debe tomar la ejecución de un proceso, el reparto de un mensaje o la deriva de un reloj. Hay dos modelos que parten de posiciones opuestas, el primero tiene en cuenta una fuerte restricción sobre el tiempo, el segundo no se hace ninguna presuposición.
- **Sistemas distribuidos síncronos:** se define un sistema distribuido síncrono como aquel en el que se establecen los siguientes límites:
 - El tiempo de ejecución de cada etapa de un proceso tiene ciertos límites inferior y superior conocidos.
 - Cada mensaje transmitido sobre un canal se recibe en un tiempo limitado conocido.

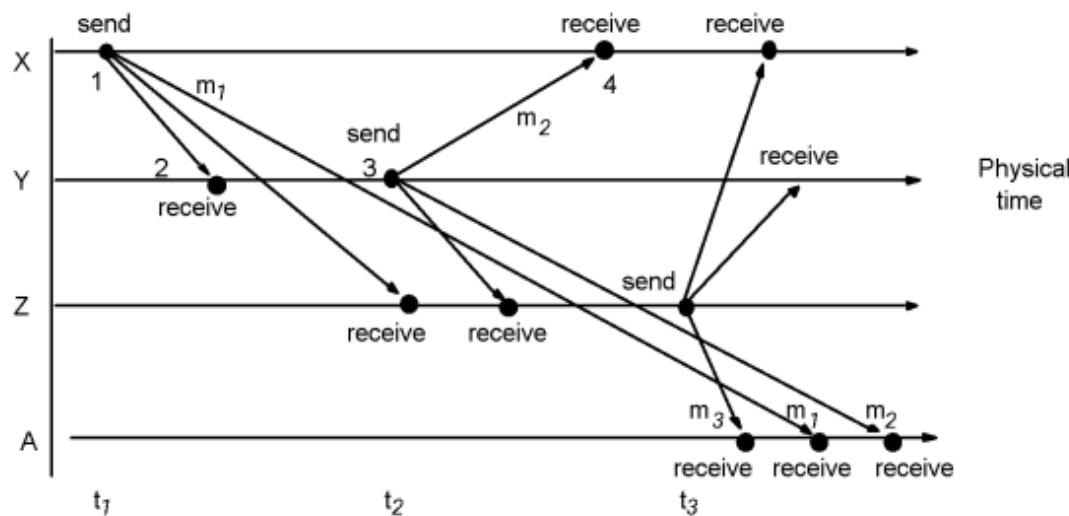
- Cada proceso tiene un reloj local cuya tasa de deriva sobre el tiempo real tiene un límite conocido.
- **Sistemas distribuidos asíncronos:** muchos sistemas distribuidos, como internet, son de gran utilidad aun sin ser sistemas síncronos. Un sistema asíncrono es aquel en que no existen limitaciones sobre:
 - La velocidad de procesamiento. Ejemplo, un paso de un proceso puede requerir tan sólo un pico segundo y otro un siglo.
 - Los retardos de transmisión de mensaje. Ejemplo, al repartir un mensaje de un proceso A a otro B puede tardar un tiempo cero o bien varios años.
 - Las tasas de deriva de reloj. (la tasa de deriva de reloj es arbitraria).

El modelo asíncrono no presupone nada sobre los intervalos de tiempo involucrados en cualquier ejecución. Éste es exactamente el modelo de Internet, en él no hay límite intrínseco en la caga del servidor o de la red. A veces un mensaje de correo puede tomarse varios días en llegar

▼ Ordenamiento de Eventos

En muchos casos, nos interesa saber si un evento en un proceso ocurrió antes, después o concurrentemente con otro evento de algún proceso. La ejecución de un sistema puede describirse en términos de los eventos y su ordenación aún careciendo de relojes precisos.

Considere el siguiente conjunto de intercambios de mensajes, el usuario X envía un mensaje, los usuarios Y y Z responden, A mira



b. Inbox de A

Item	De	Asunto
1	Z	Re: Reunión
2	X	Reunión
3	Y	Re: Reunión

Si los relojes de los computadores de X, Y y Z pudieran sincronizarse, podría utilizarse el tiempo asociado localmente a cada mensaje enviado. Por ejemplo, los mensajes m₁, m₂ y m₃ llevarían la tiempos t₁, t₂ y t₃ donde t₁ < t₂ < t₃. Los mensajes recibidos se mostrarían a los usuarios ordenados correctamente.

Si los relojes están sincronizados aproximadamente, la temporización ocurrirá en el orden correcto. Como los relojes de un sd no pueden sincronizarse de manera perfecta, Lamport propuso un modelo de tiempo lógico que pudiera utilizarse para ordenar los eventos de procesos que se ejecutan en máquinas diferentes.

El tiempo lógico permite inferir el orden en que se presentan los mensajes sin recurrir a relojes.

▼ Ordenamiento de Eventos - Lamport

Es obvio que los mensajes se reciben después de su envío, así podemos admitir un ordenamiento lógico para los pares de sucesos que se

muestran en la Figura; por ejemplo, considerando sólo los eventos relativos a X e Y.

También sabemos que las respuestas se reciben después de que se reciben los mensajes, de modo que tenemos el siguiente ordenamiento lógico para Y:

- X envía m1, antes de que Y reciba m1.
- Y envía m2 antes de que X reciba m2.
- Y recibe m1 antes de enviar m2.

El tiempo lógico lleva esta idea más lejos asignando un número a cada evento, que se corresponde con su ordenamiento lógico, de modo que los últimos eventos tendrán números mayores que los primeros. Como ejemplo, en la Figura se consignan los números 1 a 4 en los eventos en X e Y.

▼ **Modelo de Fallo**

Define las formas en que puede ocurrir un Fallo

▼ **Fallos por Omisión**

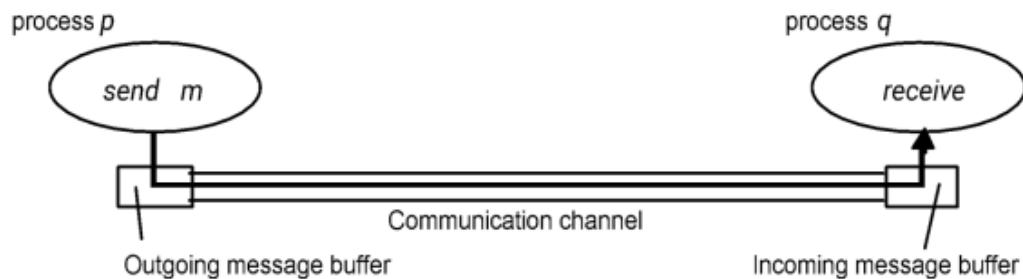
Las faltas clasificadas como fallos por omisión se refieren a casos en los que los procesos o los canales de comunicación no consiguen realizar acciones que se suponen que pueden hacer.

- **De procesos:** el principal fallo por omisión de un proceso es el fracaso o ruptura accidentada del procesamiento (crash). Cuando decimos que un proceso se rompe queremos decir que ha parado y no ejecutará ningún paso más.

Otros procesos pueden ser capaces de detectar tales fracasos por el hecho de que los procesos rotos fallan repetidamente en responder a los mensajes de invocación. Desgraciadamente, este método de detección de rupturas descansa en el uso de timeouts; es decir, un método en el cual un proceso otorga un período fijo de tiempo para que algo ocurra. En un sistema asíncrono un timeout puede indicar tan sólo que un proceso no responde; puede que se haya roto o sea lento, o que los mensajes no han llegado.

La rotura de un proceso se denomina fallo-parada (fail-stop) si los otros procesos pueden detectar con certeza que el proceso ha fracasado.

- **De comunicaciones:** el canal de comunicación produce un fallo de omisión si no transporta un mensaje desde el búfer de mensajes salientes de **p** al búfer de mensajes entrantes de **q**. A esto se denomina “perder mensajes” y su causa suele ser la falta de espacio en el búfer de recepción, o por un error de red, detectable por una suma de chequeo.



- **Fallos por omisión de envío:** pérdida de mensajes entre el proceso emisor y el búfer de mensajes de salida.
- **Fallos por omisión de recepción:** la pérdida de mensajes entre el búfer de mensajes de entrada y el proceso receptor.
- **Fallos por omisión del canal:** la pérdida de mensajes entre los búferes.

▼ Fallos arbitrarios

El término arbitrario, o fallo bizantino, se emplea para describir la peor semántica de fallo posible, en la que puede ocurrir cualquier tipo de error.

Un fallo arbitrario en un proceso es aquel en el que se omiten pasos deseables para el procesamiento o se realizan pasos no intencionados de procesamiento. En consecuencia los fallos arbitrarios en los procesos no pueden detectarse observando si el proceso responde a las invocaciones, dado que podría omitirlos arbitrariamente.

Ejemplo de fallos bizantinos en canales: contenidos corruptos de mensajes, mensajes no existentes, mensajes duplicados.

Clase de fallo	Afecta	Descripción
Fallo-parada (fail-stop)	Proceso	El proceso se detiene y permanece detenido. Otros procesos pueden detectar este estado.
fracaso o ruptura accidentada del procesamiento (crash)	Proceso	El proceso se detiene y permanece detenido. Otros procesos pueden no ser capaces de detectar este estado.
Omisión de envío	Proceso	Un proceso completa una operación de envío, pero el mensaje no se coloca en su búfer de mensajes salientes.
Omisión de recepción	Proceso	Un mensaje se coloca en el búfer de mensajes entrantes de un proceso, pero ese proceso no lo recibe.
Omisión de Canal	Canal	Un mensaje insertado en un búfer de mensajes salientes nunca llega al búfer de mensajes entrantes en el otro extremo.
Arbitrario (bizantino)	Proceso o canal	El proceso o el canal exhiben un comportamiento arbitrario: puede enviar/transmitir mensajes arbitrarios en momentos arbitrarios o cometer omisiones; un proceso puede detenerse o realizar una acción incorrecta.

▼ Fallos de temporización y Fiabilidad

- **Fallos de temporización.** Los fallos de temporización se aplican en los sistemas distribuidos síncronos donde se establecen límites en el tiempo de ejecución de un proceso, en el tiempo de reparto de un mensaje y en la tasa de deriva de reloj.

Clase de fallo	Afecta	Descripción
Reloj	Proceso	El reloj local del proceso excede el límite de su tasa de deriva sobre el tiempo real.
Prestaciones	Proceso	El proceso excede un límite sobre el intervalo entre dos pasos.
Prestaciones	Canal	La transmisión de un mensaje toma más tiempo que el límite permitido.

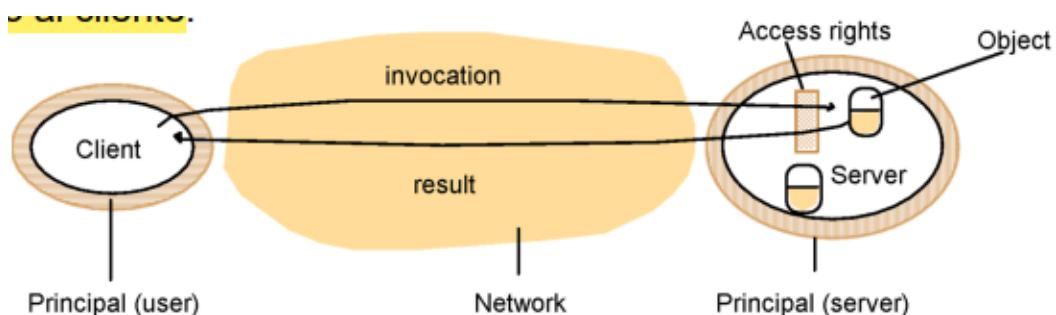
- **Fiabilidad y comunicación uno a uno.** El término comunicación fiable se define en términos de validez e integridad, definidos como sigue:
 - **Validez:** Cualquier mensaje en el búfer de mensajes salientes eventualmente llegará al búfer de mensajes entrantes.
 - **Integridad:** El mensaje recibido es idéntico al enviado, y no se entregan mensajes por duplicado.

▼ Modelo de Seguridad

La seguridad de un sistema distribuido puede lograrse asegurando los procesos y los canales empleados y protegiendo los objetos que se encapsulan contra el acceso no autorizado.

▼ Protección de Objeto

- **Protección de objetos.** Los usuarios pueden lanzar programas clientes que envían invocaciones al servidor para realizar operaciones sobre los objetos. El servidor realiza la operación indicada en cada invocación y envía el resultado al cliente.

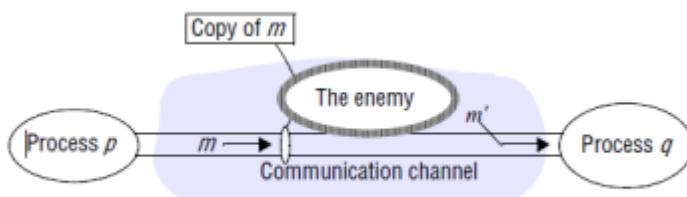


▼ Ataques

- **Asegurar procesos y sus interacciones.** Los procesos interactúan enviando mensajes. Los mensajes están expuestos a un ataque dado que la red y el servicio de comunicación que usan es abierto. Los servidores y procesos parejos exponen sus interfaces, permitiendo ser destino de los mensajes de otros procesos.

▼ El enemigo

- **El enemigo.** Es un adversario capaz de enviar cualquier mensaje a cualquier proceso y también de leer o copiar cualquier mensaje entre un par de procesos. Tales ataques pueden cometerse simplemente utilizando un computador conectado a una red para lanzar un programa que lea los mensajes de la red.



Amenazas del enemigo:

- **A procesos:** cualquier proceso diseñado para admitir peticiones puede recibir mensajes de cualquier otro proceso del sistema distribuido, y bien pudiera no ser capaz de determinar la identidad del emisor.
- **A servidores:** dado que un servidor puede recibir invocaciones de muchos clientes diferentes, no necesariamente puede determinar la identidad del principal que se halla tras cualquier invocación particular.
- **A clientes:** cuando un cliente recibe un resultado de una invocación de un servidor, no necesariamente puede decir si la fuente del mensaje resultado es del servidor o de un enemigo (spoofing).

- **A los canales de comunicación:** un enemigo puede copiar, alterar o insertar mensajes según viajan a través de la red y sus pasarelas.

Todas estas amenazas pueden vencerse empleando canales seguros, que se basan en la criptografía y la autenticación.

▼ Canales Seguros

La encriptación y la autenticación se emplean para construir canales seguros en forma de capa de servicio sobre los servicios de comunicación existentes. Un canal seguro es un canal de comunicación que conecta un par de procesos, cada uno de los cuales actúan en representación de un principal.

Un canal seguro presenta las siguientes propiedades:

- Cada proceso conoce bien la identidad del principal en cuya representación se ejecuta otro proceso. Si un cliente y un servidor se comunican vía un canal seguro, el servidor conoce la identidad del principal tras las invocaciones y puede comprobar sus derechos de acceso antes de realizar una operación. Esto permite que el servidor proteja sus objetos correctamente y permite al cliente estar seguro de estar recibiendo resultados de un servidor que actúa en buena fe.
- Un canal seguro asegura la privacidad y la integridad (protección contra la manipulación) de los datos transmitidos por él.
- Cada mensaje incluye un sello de carácter temporal, de tipo físico o lógico, para prevenir el reenvío o la reordenación de los mensajes.

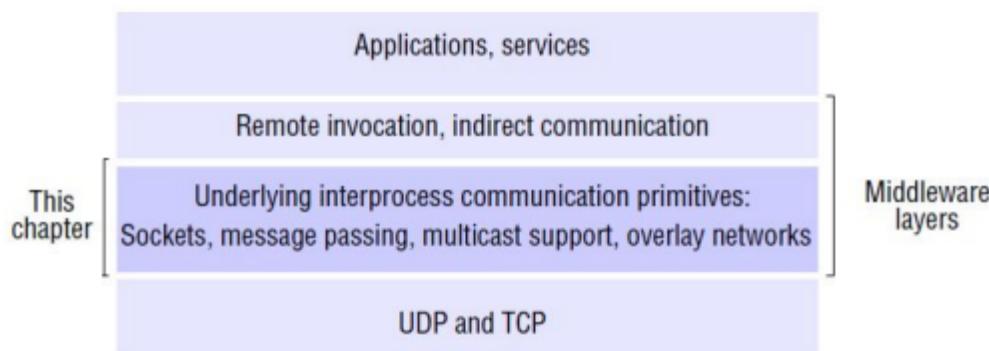
▼ Otras posibles amenazas

- **Denegación de servicio.** ésta es una forma de ataque en la que el enemigo realiza un número excesivo de invocaciones sin sentido sobre servicios o la red, lo que resulta en una sobrecarga de los recursos físicos (ancho de banda, capacidad de procesamiento del servidor).
- **Código Móvil.** un código descargado del servidor y ejecutado en el cliente podría actuar como caballo de troya.

▼ Unidad 3: Comunicación entre Procesos

▼ Capas de Middleware

Middleware layers



▼ Características de la Comunicación entre Procesos

El paso de mensajes entre un par de procesos se basa 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 (una secuencia de bytes) 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.

A cada destino de mensajes se asocia una cola. Procesos emisores producen mensajes que serán añadidos a las colas remotas. Procesos receptores eliminarán mensajes de las colas locales.

La comunicación entre los procesos emisor y receptor puede ser síncrona o asíncrona.

- **Comunicación síncrona:** los procesos receptor y emisor se sincronizan con cada mensaje. En este caso, tanto **envía** como **recibe** son operaciones bloqueantes.

P. emisor, envía - bloquea hasta que se produce el correspondiente recibe.

P. receptor, invoca un *recibe*, se bloquea hasta que llega un mensaje.

- **Comunicación asíncrona:** la operación **envía** es NO BLOQUEANTE, el proceso emisor puede continuar tan pronto como el mensaje haya sido copiado al búfer local, y la transmisión del mensaje se lleva a cabo en paralelo con el proceso emisor. **Recibe** puede tener variantes bloqueantes y no bloqueantes.
 - En la variante no bloqueante, el proceso receptor sigue con su programa después de invocar a la operación **recibe**, la cual proporciona un búfer que será llenado en un segundo plano, pero el proceso debe ser informado por separado de que su búfer ha sido llenado, ya sea por método de encuesta o mediante una interrupción.
 - En un entorno que soporte múltiples hilos en un único proceso, el **recibe** bloqueante tiene pocas desventajas, ya que puede ser invocado por un hilo mientras que el resto de hilos permanecen activos, y la simplicidad de sincronizar los hilos receptores con el mensaje entrante, es una ventaja substancial.

La comunicación no bloqueante parece ser más eficiente, pero implica una complejidad extra en el proceso receptor asociada con la necesidad de capturar mensajes entrantes fuera de su flujo de control. Por estas razones, los sistemas actuales no proporcionan la forma no bloqueante de **recibe**.

Destinos de los mensajes. En los protocolos Internet, los mensajes son enviados a direcciones construidas. (Direccion_Internet, Puerto_local).

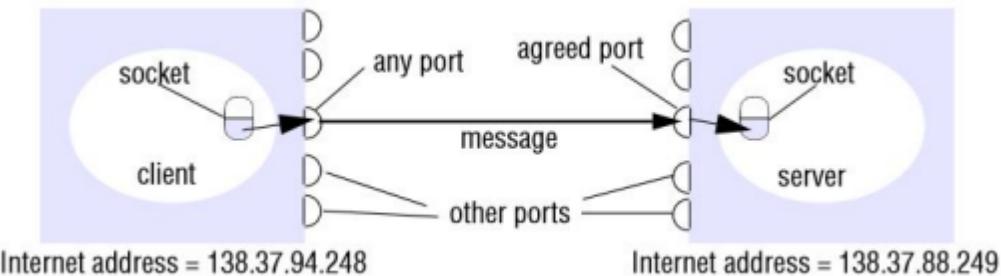
Un puerto local es el destino de un mensaje dentro de un computador. Tiene exactamente un receptor pero puede tener muchos emisores. Los procesos pueden utilizar múltiples puertos desde los que recibir mensajes. Cualquier proceso que conozca un número de puerto apropiado puede enviarle un mensaje.

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.

▼ Sockets

Ambas formas de comunicación (UDP y TCP) utilizan la abstracción de sockets (conectores), que proporciona los puntos extremos de la comunicación entre procesos.

La comunicación entre procesos consiste en la transmisión de un mensaje entre un conector de un proceso y un conector de otro proceso.



Los procesos pueden usar el mismo socket para enviar y recibir mensajes. Cada computadora tiene 2^{16} puertos posibles para ser usados por los procesos locales para recibir mensajes.

Cualquier proceso puede hacer uso de múltiples puertos para recibir mensajes, pero un proceso no puede compartir puertos con otros procesos en la misma computadora. Los procesos que usan IP multicast son una excepción. Sin embargo, cualquier número de procesos puede enviar mensajes al mismo puerto.

Cada socket está asociado con un protocolo particular, ya sea UDP o TCP.

- **Receptor:** en procesos receptores de mensajes, su conector debe estar asociado a un puerto local y a una de las direcciones del computador donde se ejecuta.

Los mensajes enviados a una dirección de Internet y a un número de puerto concretos, sólo pueden ser recibidos por el proceso cuyo conector esté asociado con esa dirección y ese puerto.

API de Java para las direcciones Internet. Como los paquetes IP que subyacen a TCP y UDP se envían a direcciones Internet, Java proporciona una clase, *InetAddress*, que representa las direcciones Internet. Los usuarios de esta clase se refieren a los computadores por sus nombres de host en el Servicio de Nombres de Dominio (Domain Name Service, DNS).

Por ejemplo, se pueden crear instancias de *InetAddress* que contienen direcciones de Internet invocando a un método estático de *InetAddress* con un nombre DNS de host como argumento. El método utiliza DNS para conseguir la correspondiente dirección Internet.

Por ejemplo, para conseguir un objeto que represente la dirección Internet de un host cuyo nombre DNS es bruno.dcs.qmw.ac.uk, utilice:

InetAddress

```
unComputador=InetAddress.getByName(bruno.dcs.qmw.ac.uk)
```

▼ Comunicación de Datagramas UDP

Un datagrama enviado por UDP se transmite desde un proceso emisor a un proceso receptor **“sin acuse de recibo”** ni **“reintentos”**. Si algo falla, el mensaje no puede llegar a su destino.

Se transmite un datagrama, entre procesos, cuando uno lo envía, y el otro lo recibe.

Cualquier proceso que necesite enviar o recibir mensajes, debe crear, primero, un **conector asociado a una dirección Internet y a un puerto local**. Un servidor, enlazará su conector a un puerto de servidor. Un cliente, ligará su conector a cualquier puerto local libre.

El método **recibe** devolverá, M: mensaje; D: dirección emisor; P: puerto emisor, permitiendo al receptor enviar la correspondiente respuesta.

Aspectos referentes a la comunicación de datagramas:

- **Tamaño del mensaje:** el receptor debe especificar una cadena de bytes de un tamaño concreto para el mensaje recibido. Si el mensaje es

demasiado grande para dicha cadena, será truncado a la llegada. La capa subyacente IP permite paquetes de hasta 2^{16} bytes, sin embargo, la mayoría de los entornos imponen una restricción a su talla de 8 kilobytes.

- **Bloqueo:** comunicación de datagramas UDP utiliza operaciones de envío, **envía**, no bloqueantes y recepciones, **recibe**, bloqueantes.
- **P. origen:** la operación **envía** devuelve el control cuando ha dirigido el mensaje a las capas inferiores UDP e IP, que son las responsables de su entrega en el destino.
- **P. destino:** a la llegada, el mensaje será colocado en una cola del conector, que está enlazado con el puerto de destino. El mensaje podrá obtenerse de la cola de recepción mediante una invocación del método **recibe** sobre este conector. Si no existe ningún proceso ligado al conector destino, los mensajes serán descartados.

EL método **recibe** produce un bloqueo hasta que se reciba un datagrama, a menos que se haya establecido un tiempo límite (time out) asociado al conector.

- **Tiempo límite de espera:** **recibe** con bloqueo indefinido, adecuado para servidores que están esperando recibir peticiones desde sus clientes.
En algunos programas no resulta apropiado la espera indefinida, sobre todo en aquellas situaciones en las que el emisor puede haber caído o se hay podido perder el mensaje. Para esto, se pueden fijar tiempos límites de espera (timeouts) en los conectores.
- **Recibe de cualquiera:** el método **recibe** no especifica el origen de los mensajes. El método **recibe** devuelve la dirección Internet y el puerto del emisor, permitiendo al receptor ver de donde viene el mensaje. Es posible vincular un socket y una dirección Internet remotas particulares, en cuyo caso el conector sólo podrá recibir y enviar mensajes con esa dirección.
- **Modelo de fallo:** el modelo de fallo puede utilizarse para proponer un modelo de fallo para los datagramas UDP, que padece de las siguientes debilidades:

- **Fallos por omisión:** los mensajes pueden deshacerse ocasionalmente, por un error detectado por comprobación o porque no queda espacio en el búfer origen o 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. Puede construirse un servicio de entrega fiable a partir de uno que adolece de fallos de omisión mediante la utilización de acuses de recibo.

Utilización de UDP. para algunas aplicaciones, resulta aceptable utilizar un servicio que sea susceptible de sufrir fallos de omisión ocasionales. **Por ejemplo el Servicio de Nombres de Dominio (DNS) está implementado sobre UDP.**

Los datagramas UDP son, en algunas ocasiones, una elección atractiva porque no padecen de las sobrecargas asociadas a la entrega de mensajes garantizada.

Existen tres fuentes principales para esta sobrecarga.

- La necesidad de almacenar información de estado en el origen y en el destino.
- La transmisión de mensajes extra.
- La latencia para el emisor.

API Java para datagramas UDP. La API Java proporciona una comunicación de datagramas por medio de dos clases: *DatagramPacket* y *DatagramSocket*.

- **DatagramPacket:** esta clase proporciona un constructor que crea una instancia compuesta por: una cadena de bytes que almacena el mensaje, la longitud del mensaje, la dirección Internet, el número de puerto local.

Las instancias de *DatagramPacket* podrán ser transmitidas entre procesos cuando uno las **envía**, y el otro las **recibe**.

- **DatagramSocket:** proporciona varios métodos que incluyen los siguientes:
 - **send y receive:** estos métodos sirven para transmitir datagramas entre un par de conectores.
 - send (una instancia de *DatagramPacket*, conteniendo el mensaje y su destino).
 - receive (*DatagramPacket* vacío en el que colocar el mensaje, su longitud y su origen).

Ambos pueden lanzar una excepción *IOException*.

- **setSoTimeout:** permite establecer un tiempo de espera límite. Cuando se fija un límite, el método receive se bloquea durante el tiempo fijado y después lanza una excepción *InterruptedException*.
- **connect:** utilizado para conectarse a un puerto remoto y dirección Internet concretos, en cuyo caso el conector sólo podrá enviar y recibir mensajes de esa dirección.

▼ Comunicación de Streams TCP

La API para el protocolo TCP. proporciona la abstracción de un flujo de bytes (stream) en el que pueden escribirse y desde el que pueden leerse datos. La abstracción de stream oculta las siguientes características de la red:

- Tamaño de los mensajes: la aplicación puede elegir la cantidad de datos que quiere escribir o leer del stream.
- El conjunto de datos puede ser muy pequeño o muy grande. La implementación del flujo TCP subyacente decide cuántos datos recoge antes de transmitirlos como uno o más paquetes IP.
- En el destino, los datos son proporcionados a la aplicación según los va solicitando. Las aplicaciones pueden forzar, si es necesario, que los datos sean enviados de forma inmediata.

Mensajes perdidos: el protocolo TCP utiliza un esquema de acuse de recibo de los mensajes.

Como un ejemplo de un esquema simple (no utilizado por TCP), el extremo emisor almacena un registro de cada paquete IP enviado y el extremo receptor acusa el recibo de todos los paquetes IP que le llegan. Si el emisor no recibe dicho acuse de recibo dentro de un plazo de tiempo fijado, volverá a transmitir el mensajes.

Control del flujo: el protocolo TCP intenta ajustar las velocidades de los procesos que leen y escriben en un stream. Si el escritor es demasiado rápido para el lector, entonces será bloqueado hasta que el lector haya consumido una cantidad suficiente de datos.

Duplicación y ordenación de los mensajes: a cada paquete IP se le asocia un identificador, que hace posible que el receptor pueda detectar y rechazar mensajes duplicados, o que pueda reordenar los mensajes que lleguen desordenados.

Destinos de mensajes: un par de procesos en comunicación establecen una conexión antes de que puedan comunicarse mediante un stream. Una vez establecida la comunicación, los procesos simplemente leen o escriben en el stream sin tener que preocuparse de las direcciones Internet ni de los números de puerto. La conexión implica una petición de conexión, *connect*, desde el cliente al servidor, seguida de una aceptación, *accept*, desde el servidor al cliente antes de que cualquier comunicación pueda tener lugar. Esto puede suponer una sobrecarga considerable para una única petición y una única respuesta.

El API para la comunicación por streams supone que en el momento de establecer una conexión uno de ellos es cliente y el otro es servidor, aunque después se comuniquen de igual a igual.

El rol de cliente implica la creación de un conector, de tipo stream, sobre cualquier puerto y la posterior petición de conexión con el servidor en su puerto de servicio.

Concordancia de ítems de datos: los dos procesos que se comunican necesitan estar de acuerdo en el tipo de datos transmitidos por el stream. Por ejemplo, si un proceso escribe un int seguido de un double, el proceso receptor debe interpretarlo como un int seguido de un double. Cuando un par de procesos no coopera correctamente en el uso del stream, el proceso lector puede encontrarse con problemas cuando interprete los datos o

puede bloquearse debido a que se encuentre una cantidad insuficiente de datos en el stream.

Bloqueo: los datos escritos en un stream se almacenan en un búfer en el conector destino. Cuando un proceso intenta leer datos en un canal de entrada, se extrae los datos de la cola o se bloquea hasta que existan datos disponibles.

El proceso que escribe los datos en el stream resultará bloqueado por el mecanismo de control de stream de TCP si el conector del otro lado intenta almacenar en la cola de entrada más información de la permitida.

Hilos: cuando un servidor acepta una conexión, generalmente crea un nuevo hilo con el que comunicarse con el nuevo cliente. La ventaja de utilizar un hilo separado para cada cliente es que el servidor puede bloquearse a la espera de entradas sin afectar a los otros clientes. En un entorno en el cual no se disponga de hilos, una alternativa es comprobar si existen datos accesibles en el stream antes de intentar leerlos.

Modelo de fallo. los streams TCP utilizan:

- **suma de comprobación** para detectar y rechazar paquetes corruptos.
- **número de secuencia** para detectar y eliminar los paquetes duplicados.

Con respecto a la propiedad de validez, los streams TCP utilizan timeouts y retransmisión de los paquetes perdidos.

Se tiene garantizada la entrega incluso cuando alguno de los paquetes subyacentes se haya perdido.

Conexión Rota: si la pérdida de paquetes sobrepasa un cierto límite, o la red que conecta un par de procesos está severamente congestionada, el software TCP responsable de enviar los mensajes no recibirá acuses de recibo de los paquetes enviados y después de un tiempo declarará rota la conexión.

Cuando una conexión está rota, se notificará al proceso que la utiliza siempre que intente leer o escribir. Esto tiene los siguientes efectos:

- Los procesos que utilizan la conexión no distinguen entre un fallo en la red y un fallo en el proceso que está en el otro extremo de la conexión.

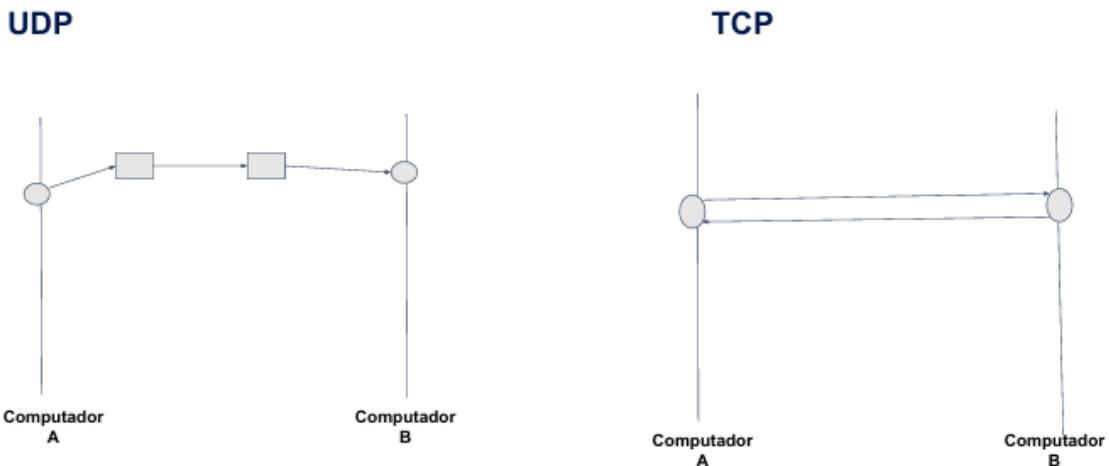
- Los procesos comunicantes no pueden saber si sus mensajes recientes han sido recibidos o no.

Utilización de TCP. Muchos de los servicios utilizados se ejecutan sobre conexiones TCP, con números de puertos reservados. Entre ellos se encuentran los siguientes:

- **HTTP / HTTPS:** El protocolo de transferencia de hipertexto se utiliza en comunicación entre un navegador y un servidor web.
- **FTP:** El protocolo de transferencia de archivos permite leer los directorios de un computador remoto y transferir archivos entre los computadores de una conexión.
- **Telnet:** La herramienta Telnet proporciona acceso a un terminal en un computador remoto.
- **SMTP:** El protocolo simple de transferencia de correo se utiliza para mandar correos electrónicos entre computadores.

El API Java para los streams TCP. La interfaz Java para los streams TCP está constituida por las clases *ServerSocket* y *Socket*.

- **ServerSocket:** está diseñada para ser utilizada por un servidor para crear un conector en el puerto de servidor que escucha las peticiones de conexión de los clientes. Su método *accept* toma una petición *connect* de la cola, o si la cola está vacía, se bloquea hasta que llega una petición. El resultado de ejecutar *accept* es una instancia de *Socket*, un conector que da acceso a streams para comunicarse con el cliente.
- **Socket:** es utilizada por un par de procesos de una conexión. El cliente utiliza un constructor para crear un conector, especificando el nombre DNS de host y el puerto del servidor. Este constructor no sólo crea el conector asociado con el puerto local, sino que también se conecta con el computador remoto especificado en el puerto indicado.



▼ Representación Externa de Datos y Empaquetado

La información almacenada en los programas en ejecución se representa como estructuras de datos (por ejemplo, conjunto de objetos interconectados) mientras que la información en los mensajes consiste en secuencias de bytes.

Independientemente de la forma de comunicación utilizada, las estructuras de datos deben ser aplanadas (convertido a una secuencia de bytes) antes de la transmisión y reconstruido en destino.

Los datos primitivos en la transmisión pueden tener valores de diferentes tipos, y no todos los computadores almacenan valores, como enteros, en el mismo orden. La representación de números en coma flotante también difiere entre las arquitecturas.

Dos métodos se pueden utilizar para realizar el intercambio de valores binarios de datos entre dos ordenadores:

- Los valores se convierten a un formato convenido externo antes de la transmisión y convertido a la forma local en recepción. Si los dos ordenadores conocen el mismo tipo, la conversión a formato externo puede omitirse.
- Los valores se transmiten en formato del remitente, junto con una indicación del formato utilizado, y el receptor convierte los valores si es necesario.

Alternativas para la representación externa de datos y empaquetado:

- **Representación común de datos CORBA:** una representación externa de los tipos estructurados y primitivos, que se pueden pasar como argumentos y resultados de invocaciones de métodos remotos en CORBA. Puede ser utilizado por una gran variedad de lenguajes de programación.
- **Serialización de objetos Java:** representación "aplanada" y externa de datos de cualquier objeto o árbol de objetos que pueden necesitar ser transmitidos en un mensaje o almacenado en un disco. Es para uso exclusivo de Java.
- **XML (eXtensible Markup Language):** define un formato para la representación de datos estructurados. Fue pensado originalmente para los documentos que contienen datos estructurados auto-descritos. Documentos accesibles en la Web. Representar datos enviados y recibidos en mensajes intercambiados por los clientes y servidores de los Servicios Web.
- **Protocol Buffers (Google)**
- **JSON**

▼ Comunicación en Grupo

Multidifusión IP. La multidifusión IP se construye sobre el protocolo Internet, IP. Los paquetes IP se dirigen a los computadores; mientras que los puertos pertenecen a niveles TCP y UDP.

La multidifusión IP permite que el emisor transmita un único paquete IP a un conjunto de computadores que forman un grupo de multidifusión.

El emisor no tiene que estar al tanto de las entidades de los receptores individuales y del tamaño del grupo. Los grupos de multidifusión se especifican utilizando las direcciones Internet de la clase D, esto es, una dirección cuyos primeros cuatro bits son 1110 en IPv4.

El convertirse en miembro de un grupo de multidifusión, permite al computador recibir los paquetes IP enviados al grupo. La pertenencia a los grupos de multidifusión es dinámica, permitiéndose que los computadores

se apunten o se borren a un número arbitrario de grupos en cualquier instante.

Los siguientes detalles son específicos de IPv4:

- **Routers multidifusión:** los paquetes IP pueden multidifundirse tanto en la red local como en toda Internet. La multidifusión local utiliza la capacidad de multidifusión de la red local, por ejemplo una Ethernet. La multidifusión dirigida a Internet hace uso de las posibilidades de multidifusión de los routers, los cuales reenvían los datagramas únicamente a otros routers con miembros de ese grupo, donde serán multidifundidos a los miembros locales. Para limitar la distancia de propagación de un datagrama de multidifusión, el emisor puede especificar el número de routers que puede cruzar; llamado tiempo de vida (*time to TTL*).
- **Reserva de direcciones de multidifusión:** las direcciones de multidifusión se pueden reservar de forma temporal o permanente. Existen grupos permanentes incluso cuando no existe ningún miembro.

▼ Virtualización de red

Se ocupa de la construcción de redes virtuales diferentes sobre una red existente tal como el Internet. Cada red virtual puede ser diseñada para soportar una aplicación distribuida en particular. Por ejemplo, puede soportar streaming multimedia, como en ..., y coexistir con otro que admite un juego en línea multijugador, ambos corriendo sobre la misma red subyacente.

Esto sugiere una respuesta al dilema: "una red virtual específica de aplicación puede ser construida sobre una red existente y optimizada para esa aplicación particular, sin cambiar las características de la Red subyacente."

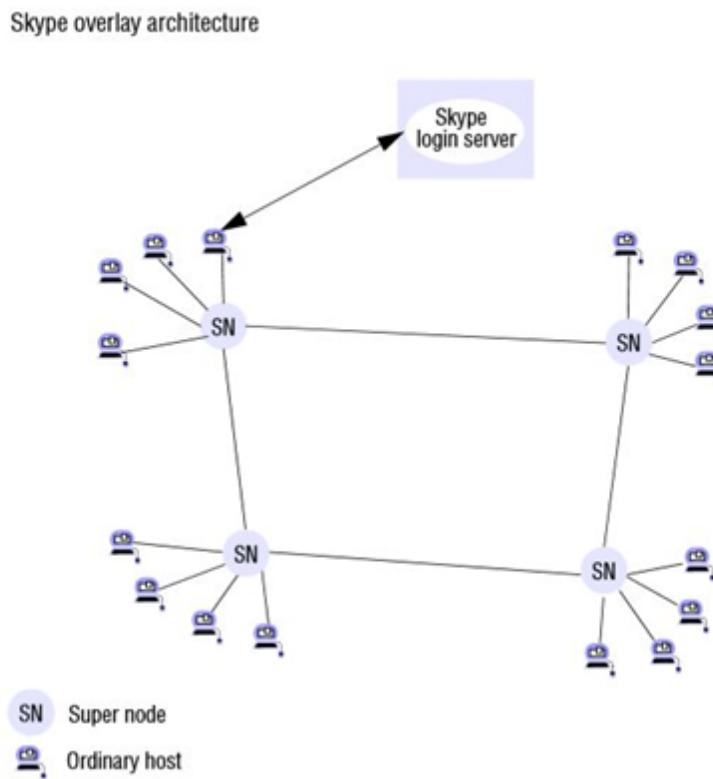
▼ Overlay networks

Una red de superposición (overlay networks) es una red virtual que consta de nodos y enlaces virtuales, que se encuentran encima de una red subyacente (como una red IP) y ofrece algo que no se proporciona de otra manera:

- un servicio adaptado a las necesidades de una clase de aplicación o de un servicio particular de alto nivel, por ejemplo, distribución de contenido multimedia.
- funcionamiento más eficiente en un entorno de red dado, por ejemplo, encaminamiento en una red ad hoc.
- una característica adicional, por ejemplo, comunicación multicast o segura.

De manera similar, cada red virtual tiene su propio esquema de direccionamiento particular, protocolos y algoritmos de enruteamiento, pero redefinido para satisfacer las necesidades particulares de las diferentes clases de aplicaciones.

Skype: Un ejemplo de overlay network



▼ Unidad 4: Objetos distribuidos e Invocación Remota

▼ Introducción

- **Aplicación Distribuida:** Aplicaciones que se componen de programas cooperantes corriendo en procesos distintos. Es necesario invocar operaciones en otros procesos, generalmente en computadores diferentes.
- **Middleware:** Software que proporciona un modelo de programación sobre bloques básicos arquitectónicos, esto es: procesos y paso de mensajes.

Emplea protocolos basados en mensajes entre procesos para proporcionar abstracciones de un nivel mayor, tales como "invocaciones remotas y eventos".

Se puede tener implementaciones en lenguajes diferentes.

Un middleware proporciona:

- **Transparencia frente a ubicación:** En RPC, el cliente que llama a un procedimiento no puede discernir si el procedimiento se ejecuta en el mismo proceso o en un proceso diferente.
- **Protocolos de comunicación:** Los protocolos del middleware no dependen de los protocolos de transporte subyacentes.
- **Hardware de los computadores:** Se emplean en el empaquetado y desempaquetado de mensajes.
- **Sistemas Operativos:** Las abstracciones de mayor nivel que provee la capa de middleware son independientes de los sistemas operativos subyacentes.
- **Utilización de diversos lenguajes de programación:** Diversos middleware se diseñan para permitir que las aplicaciones distribuidas sean escritas en más de un lenguaje de programación

Los conceptos de middleware a alto nivel se centran en cómo los procesos se comunican en un sistema distribuido.

Hay tres paradigmas de invocación remota:

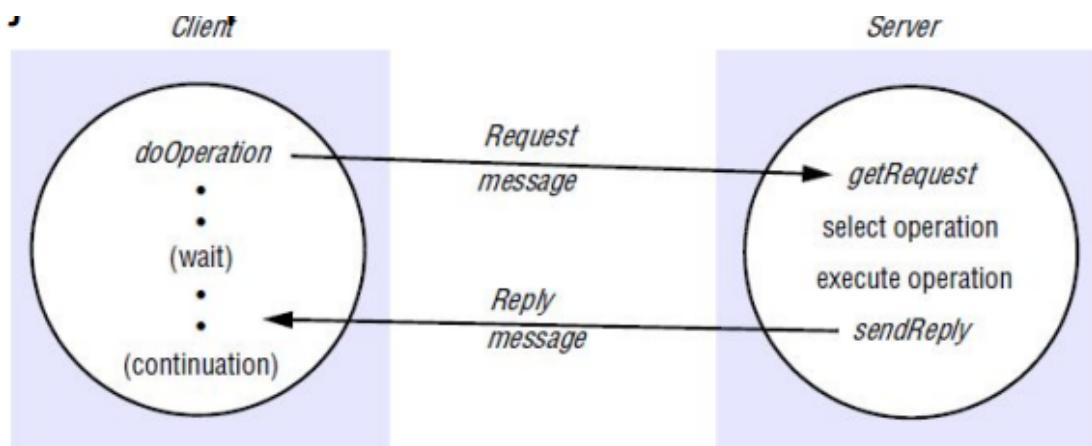
- **Protocolos de Petición-Respuesta**, un modo relativamente a bajo nivel para ejecutar una operación remota. Sienta las bases para RPC y RMI
- **Remote Procedure Call (RPC)**, es el primer modelo (y el más conocido) para facilitar las llamadas a servidores remotos.
- **Remote Method Invocation (RMI)**, extensión de RPC para la llamada a métodos de objetos en nodos remotos.

▼ Protocolo Petición-Respuesta

- Comunicación cliente-servidor, está orientada a soportar los roles y el intercambio de mensajes de las interacciones típicas cliente-servidor.
- En el caso normal, la comunicación petición-respuesta es síncrona, ya que el proceso cliente se bloquea hasta que llega la respuesta del servidor.
- También puede ser **fiable**, ya que la respuesta del servidor actúa como **acuse de recibo para el cliente**.
- La comunicación cliente-servidor asíncrona es útil en situaciones donde los clientes pueden recuperar las respuestas más tarde.

▼ Primitivas del Protocolo Petición-Respuesta

1. **doOperation(s, args)**: Invoca una operación remota en s, con los argumentos indicados en args (parámetros y tipo de operación).
2. **getRequest()**: Espera/adquiere una petición en el servidor.
3. **sendReply(r, c)**: Manda el mensaje de respuesta r al cliente c.



public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)

Sends a request message to the remote server and returns the reply. The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[] getRequest ()

Acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

Sends the reply message reply to the client at its Internet address and port.

Información transmitida en un mensaje de solicitud o respuesta:

messageType	<i>int (0=Request, 1=Reply)</i>
requestId	<i>int</i>
remoteReference	<i>RemoteRef</i>
operationId	<i>int or Operation</i>
arguments	<i>// array of bytes</i>

▼ Modelo de Fallos

Si las tres primitivas **doOperation**, **getRequest**, **sendReply** se implementan con datagramas UDP, **adolecerán de los mismos fallos de comunicación** que cualquier otra aplicación de UDP. Esto es:

- **Fallos por omisión:** el servidor no responde o el mensaje se pierde.
- **Bizantino:** Los mensajes pueden llegar duplicados y desordenados.

Además, el protocolo puede padecer el fallo de los procesos.

Modelo de Fallo por Omisión: Estrategia de **timeout** en **doOperation()**.

- Opción 1: la operación **doOperation** simplemente retorna un indicador de fallo.

- Opción 2: reintentar `doOperation` de forma repetida hasta que se obtenga una respuesta.

Modelo de Fallo Bizantino:

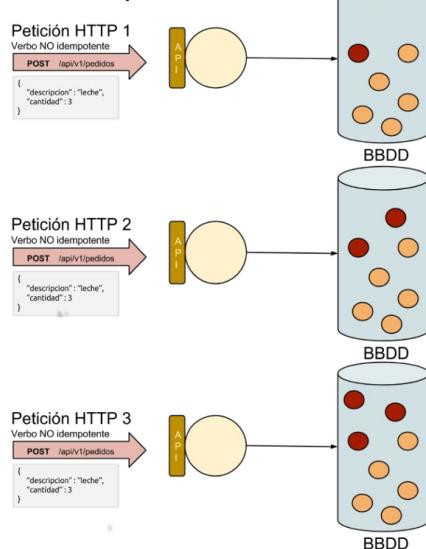
- Opción 1: Puede implementar un mecanismo para descartarlas si todavía está realizando la operación. Esto es posible con el identificador de petición:

`id_cliente # id_peticion`

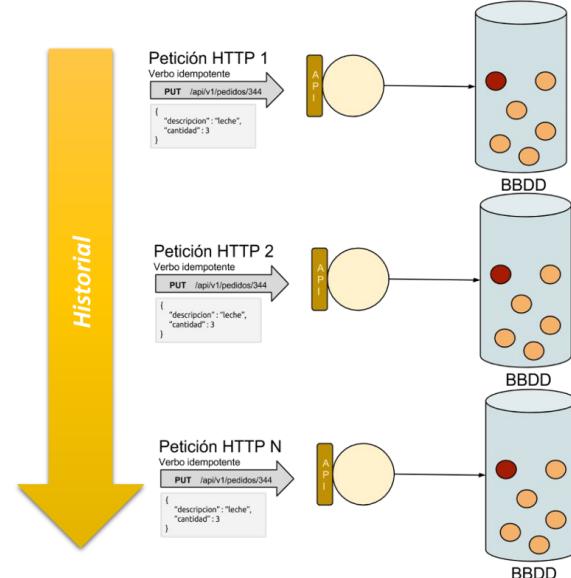
- Opción 2: Si ya la ha realizado y la operación es idempotente, la realiza de nuevo.
Si no, puede implementar un histórico con los resultados de las operaciones realizadas.

▼ Idempotencia

- Operaciones no idempotentes



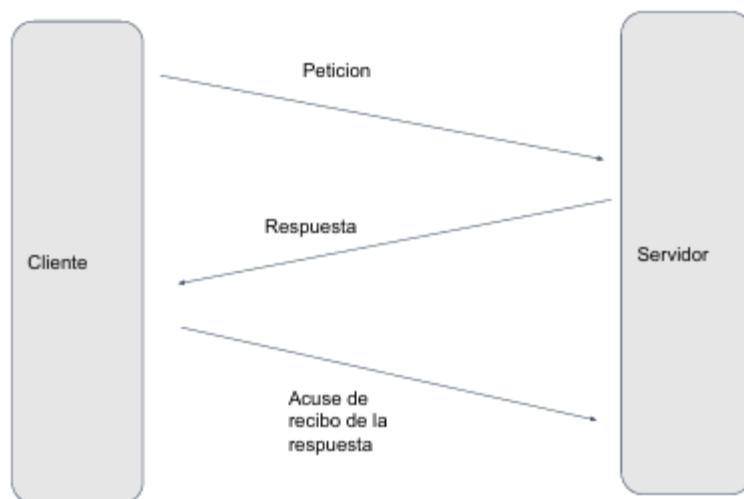
- Operaciones idempotentes



▼ Estilos de Protocolos RPC

En la implementación de los distintos tipos de RPC se utilizan tres protocolos con diferentes semánticas en presencia de fallos de comunicación

- **El protocolo petición (R).** cuando el procedimiento no devuelve ningún valor y el cliente no necesita confirmación de que ha sido ejecutado.
- **El protocolo petición-respuesta (RR).** los mensajes de respuesta del servidor sirven como confirmación de las peticiones cliente.
- **El protocolo petición-respuesta-confirmación de la respuesta (RRA).** está basado en el intercambio de tres mensajes: petición-respuesta-confirmación de la respuesta.



<i>Name</i>	<i>Messages sent by</i>		
	<i>Client</i>	<i>Server</i>	<i>Client</i>
R	<i>Request</i>		
RR	<i>Request</i>	<i>Reply</i>	
RRA	<i>Request</i>	<i>Reply</i>	<i>Acknowledge reply</i>

▼ Protocolo petición-respuesta

Http: basado en protocolo petición-respuesta.

- Cliente: Navegador.
- Servidor: WebServer.

HTTP especifica los mensajes que involucran métodos, los argumentos, resultados y reglas para ensamblarlos en los mensajes.

Soporta un conjunto fijo de métodos (GET, PUT, POST, etc) que son aplicables a todos los recursos. Al contrario de otros protocolos, cada objeto tiene sus propios métodos. Además permite:

- **Negociación del Contenido:** las peticiones de los clientes pueden incluir información (por ejemplo lenguaje o tipo de medio).
- **Autenticación:** se utilizan credenciales y desafíos para conseguir una autenticación del estilo clave de acceso.

HTTP se implementa sobre TCP. En la versión original del protocolo, cada interacción cliente-servidor se componía de los siguientes pasos:

- El cliente solicita una conexión al puerto del servidor por defecto o a otro especificado en la petición aceptada por el servidor.
- El cliente envía un mensaje de petición al servidor.
- El servidor envía un mensaje de respuesta al cliente.
- Se cierra la conexión.

▼ 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.

Los métodos considerados son los siguientes:

- **GET:** pide el recurso cuyo URL se da como argumento.
 - Si el URL se refiere a datos, entonces el servidor responderá enviando de vuelta los datos indicados por el URL.
 - Si el URL se refiere a un programa, entonces el servidor web ejecutará el programa y devolverá su salida al cliente.
 - Admite argumentos al URL.

- **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 proporcionados en el cuerpo de 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.

▼ Mensaje HTTP

Mensaje HTTP request

<i>method</i>	<i>URL or pathname</i>	<i>HTTP version</i>	<i>headers</i>	<i>message body</i>
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Mensaje HTTP reply

<i>HTTP version</i>	<i>status code</i>	<i>reason</i>	<i>headers</i>	<i>message body</i>
HTTP/1.1	200	OK		resource data

▼ Códigos HTTP

Códigos de estado de respuesta HTTP indican el estado (exitoso o no) de una petición HTTP. Las respuestas se agrupan en cinco clases:

1. Respuestas informativas (100-199),
2. Respuestas satisfactorias (200-299),
3. Redirecciones (300-399),
4. Errores de los clientes (400-499),
5. Errores de los servidores (500-599).

Code	Reason-Phrase	Defined in...
100	Continue	Section 6.2.1
101	Switching Protocols	Section 6.2.2
200	OK	Section 6.3.1
201	Created	Section 6.3.2
202	Accepted	Section 6.3.3
203	Non-Authoritative Information	Section 6.3.4
204	No Content	Section 6.3.5
205	Reset Content	Section 6.3.6
206	Partial Content	Section 4.1 of [RFC7233]
300	Multiple Choices	Section 6.4.1
301	Moved Permanently	Section 6.4.2
302	Found	Section 6.4.3
303	See Other	Section 6.4.4
304	Not Modified	Section 4.1 of [RFC7232]
305	Use Proxy	Section 6.4.5
307	Temporary Redirect	Section 6.4.7
400	Bad Request	Section 6.5.1
401	Unauthorized	Section 3.1 of [RFC7235]
402	Payment Required	Section 6.5.2
403	Forbidden	Section 6.5.3
404	Not Found	Section 6.5.4
405	Method Not Allowed	Section 6.5.5
406	Not Acceptable	Section 6.5.6
407	Proxy Authentication Required	Section 3.2 of [RFC7235]
408	Request Timeout	Section 6.5.7
409	Conflict	Section 6.5.8
410	Gone	Section 6.5.9
411	Length Required	Section 6.5.10
412	Precondition Failed	Section 4.2 of [RFC7232]
413	Payload Too Large	Section 6.5.11
414	URI Too Long	Section 6.5.12
415	Unsupported Media Type	Section 6.5.13
416	Range Not Satisfiable	Section 4.4 of [RFC7233]
417	Expectation Failed	Section 6.5.14
426	Upgrade Required	Section 6.5.15
500	Internal Server Error	Section 6.6.1
501	Not Implemented	Section 6.6.2
502	Bad Gateway	Section 6.6.3
503	Service Unavailable	Section 6.6.4
504	Gateway Timeout	Section 6.6.5
505	HTTP Version Not Supported	Section 6.6.6

▼ Remote Procedure Call (RPC)

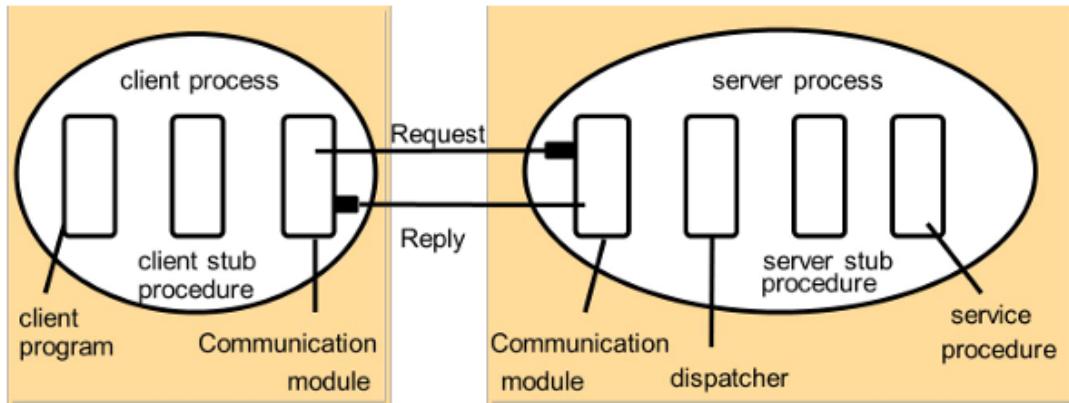
▼ RPC

- La llamada a procedimiento remoto (RPC) representa un adelanto en computación distribuida, con el objetivo de hacer que la programación de sistemas distribuidos parezca similar, si no idéntica, a la programación convencional.
- Logra un alto nivel de transparencia de distribución. Esta unificación se logra de una manera muy simple, extendiendo la abstracción de una llamada de procedimiento a entornos distribuidos.
- Una llamada a un procedimiento remoto es muy similar a una invocación a un método remoto, en la que un programa cliente llama a un procedimiento de otro programa en ejecución en un servidor.
- Los servidores pueden ser clientes de otros servidores para permitir cadenas de RPC.

- Un proceso servidor define en su **interfaz de servicio** los procedimientos disponibles para ser llamados remotamente. RPC, como RMI, puede implementarse para ofrecer alguna de las semánticas de invocación discutidas, al menos una vez o como máximo una vez.
- RPC se implementa usualmente sobre un protocolo petición-respuesta.
- Los contenidos de los mensajes de petición y respuesta son los mismos que los que mostraron para RMI, excepto que se omite el campo **ReferenciaObjeto**.
- El software que soporta RPC es similar al mostrado para RMI **excepto que no se requieren módulos de referencias remotas**, dado que las llamadas a procedimientos no tienen que ver con objetos y referencias a objetos.
- El cliente que accede a un servicio incluye un **procedimiento de resguardo** para cada procedimiento en la interfaz de servicio. Procedimiento de resguardo es similar al de un proxy.
- Se comporta como un procedimiento local del cliente, pero en lugar de ejecutar la llamada, empaqueta el identificador del procedimiento y los argumentos en un mensaje de petición, que se envía vía su módulo de comunicación al servidor.
- Cuando llega el mensaje de respuesta, desempaquetá los resultados. **El proceso servidor contiene un distribuidor junto a un procedimiento de resguardo de servidor y un procedimiento de servicio para cada procedimiento de la interfaz de servicio.**
- El distribuidor selecciona uno de los procedimientos de resguardo según el identificador de procedimiento del mensaje de petición.
- **Un procedimiento de resguardo de servidor** es como un método de esqueleto en el que se desempaquetan los argumentos en el mensaje de petición, se llama al procedimiento de servicio correspondiente y se empaquetan los datos con el resultado para el mensaje de respuesta.

- Los procedimientos de servicio implementan los procedimientos en la interfaz de servicio.

▼ Papel de los Procedimientos de Resguardo de Cliente y Servidor en RPC



▼ Interfaces

- **Las interfaces en los sistemas distribuidos.** En un programa distribuido, los módulos pueden lanzarse en procesos separados.
- Para definir módulos, la mayoría de los LP proporcionan medios para que varios módulos puedan comunicarse entre sí: las **interfaces**.
- No es posible para un módulo que se ejecuta en un proceso acceder a las variables de un módulo que está en otro proceso.
- Asimismo, la interfaz de un módulo escrita para RPC o RMI no puede especificar el acceso directo a variables.
- Las interfaces en IDL de **CORBA** pueden especificar atributos, lo que parece violar esta regla. Sin embargo, los atributos no son accedidos directamente sino mediante ciertos procedimientos de escritura y lectura que se añaden automáticamente a la interfaz.

Parámetros de entrada:

- Se pasan al módulo remoto mediante el envío de los valores de los argumentos en el mensaje de petición.

- Posteriormente se proporcionan como argumentos a la operación que se ejecutará en el servidor.

Los parámetros de salida:

- Se devuelven en el mensaje de respuesta y se sitúan como la respuesta de la llamada o reemplazando valores del argumento en el entorno.
- Los punteros en un proceso dejan de ser válidos en el remoto. En consecuencia, no pueden pasarse punteros como argumentos o como valores retornados.
- **Interfaces de servicio:** En el modelo cliente-servidor, cada servidor proporciona procedimientos disponibles para clientes. El término se emplea para referirse a la especificación de los procedimientos que ofrece un servidor. Por ejemplo, un servidor de archivos proporcionará procedimientos para leer y escribir archivos. **Define los tipos de los argumentos de entrada y salida.**
- **Interfaces remotas:** En el modelo de objetos distribuidos, una interfaz remota especifica los métodos de un objeto que están disponibles para su invocación por objetos de otros procesos, y define los tipos de los argumentos de entrada y de salida. Sin embargo, la gran diferencia es que los métodos en las interfaces remotas pueden pasar objetos como argumentos y como resultados de los métodos.
- **Lenguajes de definición de interfaces.**
 - Mecanismo RMI con un lenguaje de programación concreto.
 - Incluye una notación apropiada para definir interfaces.
 - Debe permitir relacionar los parámetros de entrada y de salida con el uso habitual de los parámetros en ese lenguaje.
- JavaRMI es un ejemplo en el que se ha añadido un mecanismo RMI a un LP.
- Esta aproximación es útil cuando se puede escribir cada parte de una aplicación distribuida en el mismo lenguaje.

- Es conveniente, permite al programador emplear un solo lenguaje para las invocaciones locales y remotas.
- **Los lenguajes de definición de interfaces (IDL)** permiten que los objetos implementados en lenguajes diferentes se invoquen unos a otros.
- **Un IDL proporciona:** notación para definir interfaces en la cual cada uno de los parámetros de un método se podrá describir como de entrada o de salida además de su propia especificación de tipo.

▼ Semánticas de Invocación

Medidas de tolerancia a fallos			Semánticas De Invocación
Retransmisión de mensajes	Filtrado De duplicados	Reejecución del procedimiento O retransmisión de la respuesta	
No	No procede	No procede	Pudiera ser
Si	No	Reejecutar proced.	Al menos una vez
Si	Si	Retransmitir resp.	Como máximo una vez

- Medidas de tolerancia a fallo:
 - **Reintento del mensaje de petición:** se retransmite el mensaje de petición hasta que, o bien se recibe una respuesta o se asume que el servidor ha fallado.
 - **Filtrado de duplicados:** cuando se emplean retransmisiones, se descartan las peticiones duplicadas en el servidor.
 - **Retransmisión de resultados:** se mantiene una historia de los mensajes de resultados para permitir retransmitir los resultados perdidos sin reejecutar las operaciones en el servidor.
- **Semántica de invocación “pudiera ser”:** el que invoca no puede decir si un método se ha ejecutado una vez, o ninguna en absoluto.
 - La semántica pudiera ser aparece cuando no se aplica ninguna medida de tolerancia ante fallos. Esta puede padecer de los

siguientes tipos de fallo:

- Fallos de omisión si se pierde la invocación o el mensaje con el resultado.
- Fallos por caída cuando el servidor que contiene el objeto remoto falla.
- **Incógnita!** Si el mensaje con el resultado no se recibe tras un timeout y no hay reintentos, no hay certeza si se ha ejecutado el método. Si se pierde el mensaje de invocación, entonces el método no se habrá ejecutado. Por otro lado, puede haberse ejecutado el método y haberse perdido el mensaje con el resultado.
- **Semántica de invocación “al menos una vez”:** el invocante recibe un resultado, en cuyo caso el invocante sabe que el método se evaluó al menos una vez, a menos que se reciba una excepción informando que no se recibe ningún resultado.
 - La semántica de invocación al menos una vez puede alcanzarse mediante la **retransmisión de los mensajes de petición**, que enmascara los fallos por omisión de los mensajes de invocación del resultado. La semántica al menos una vez puede padecer los siguientes tipos de fallos:
 - Fallos por caída cuando el servidor que contiene el objeto remoto falla.
 - Fallos arbitrarios. En casos donde el mensaje de invocación se retransmite, el objeto remoto puede recibirla y ejecutar el método más de una vez, provocando que se almacenen o devuelvan valores posiblemente erróneos.
- **Semántica como máximo una vez:** el invocante recibe bien un resultado, en cuyo caso el invocante sabe que el método se ejecutó exactamente una vez, o una excepción que le informa de que no se recibió el resultado, de modo que el método se habrá ejecutado o una vez o ninguna en absoluto.

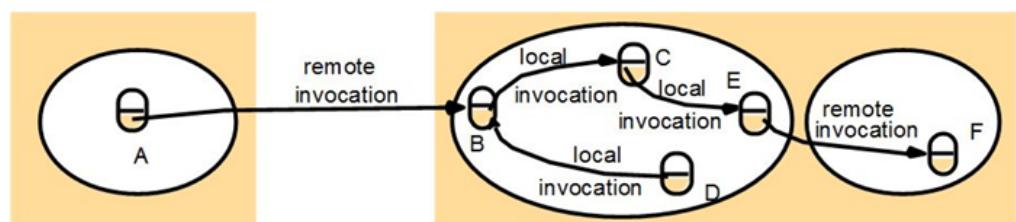
- La semántica de invocación como **máximo una vez** puede obtenerse utilizando todas las medidas de tolerancia frente a fallos.
- Tanto Java RMI como CORBA observan la semántica de invocación como **máximo una vez**, pero CORBA permite emplear la semántica **pudiera ser** para los métodos que no devuelven resultados.
- Sun RPC proporciona la semántica de llamadas **al menos una vez**.

▼ RMI

- Invocación a Método Remoto
- Remote Method Invocation

▼ El modelo de objetos distribuido

- RMI está estrechamente relacionado con RPC, pero extiende al mundo de los objetos distribuidos. En RMI, un objeto llamante puede invocar un método en un objeto potencialmente remoto. Al igual que con RPC, los detalles subyacentes generalmente están ocultos para el usuario.
- **Invocaciones de métodos remotas:** Las invocaciones de métodos entre objetos en diferentes procesos.
- **Invocaciones de métodos locales:** Las invocaciones de métodos entre objetos del mismo proceso.
 - Ejemplo: AppMovil solicita sonidos guardados por otro proceso.

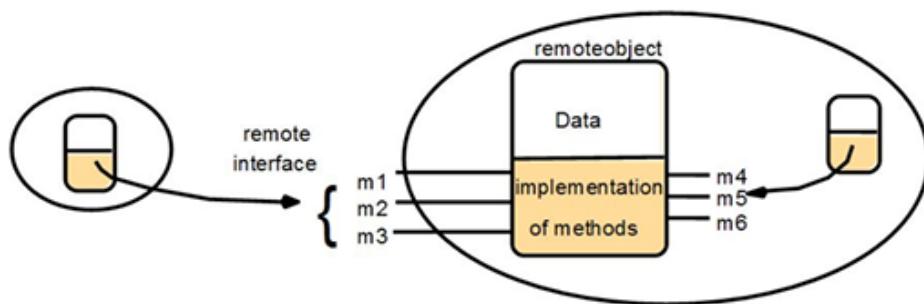


- **Objetos Remotos:** Objetos que pueden recibir invocaciones remotas.

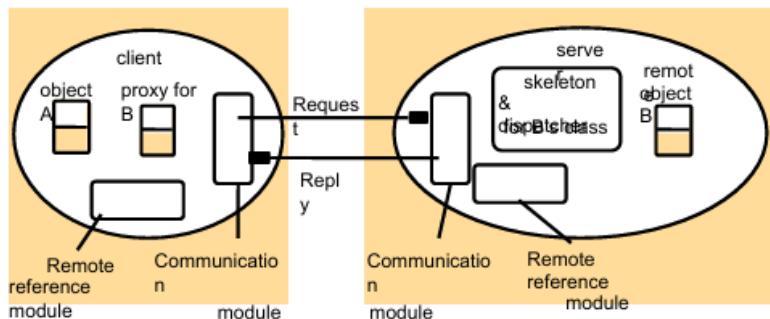
- B y F son objetos remotos. Todos los objetos pueden recibir invocaciones locales, a pesar de que sólo puedan recibirlas de otros objetos que posean referencias a ellos. Por ejemplo, el objeto C puede tener una referencia al objeto E de modo que puede invocar uno de sus métodos.
- Los 2 conceptos fundamentales siguientes son el corazón del modelo de objetos distribuidos:
 - **Referencia de objeto remoto:** otros objetos pueden invocar los métodos de un objeto remoto si tienen acceso a su “referencia de objeto remoto”.
 - **Interfaz remota:** cada objeto remoto tiene una interfaz remota que especifica cuáles de sus métodos pueden invocarse remotamente.
- **Referencias a objetos remotos:** permite que cualquier objeto que pueda recibir un RMI tenga una referencia a objeto remoto.
- Una referencia a objeto remoto “**es un identificador que puede usarse a lo largo de todo un sistema distribuido para referirse a un objeto remoto particular único.**”
- Las referencias a objetos remotos son análogas a las locales en cuanto a que:
 - El objeto remoto donde se recibe la invocación de método remoto se especifica mediante una referencia a objeto remoto.
 - Las referencias a objetos remotos pueden pasarse como argumentos y resultados de las invocaciones de métodos remotos.
- **Interfaces remotas:** La clase de un objeto remoto implementa los métodos de su interfaz remota. Los objetos en otros procesos pueden invocar solamente los métodos que pertenezcan a su interfaz remota.
- Los objetos locales pueden invocar los métodos en la interfaz remota así como otros métodos implementados por un objeto remoto.

- **El sistema CORBA** proporciona IDL, que permite definir interfaces remotas.
- **En Java RMI**, las interfaces remotas se definen de la misma forma que cualquier interfaz Java.
- Adquieren su capacidad de ser interfaces remotas al extender una interfaz denominada **Remote**.
- **Acciones en un sistema de objetos distribuido:** Como en el caso no distribuido, una acción se inicia mediante la invocación de un método, que pudiera resultar en consiguientes invocaciones sobre métodos de otros objetos. Pero en el caso distribuido, los objetos involucrados en una cadena de invocaciones relacionadas pueden estar ubicados en procesos o en computadores diferentes.
 - Cuando una invocación cruza los límites de un proceso o un computador, se emplea una RMI, y la referencia remota al objeto se hace disponible para hacer posible la RMI.
 - En la Figura 5.3, el objeto A necesita poseer una referencia a objeto remoto para el objeto B. Las referencias a un objeto remoto pueden obtenerse como resultado de una invocación a un objeto remoto. Por ejemplo, el objeto A podría obtener una referencia remota al objeto F desde el objeto E.
- **Excepciones:** Cualquier invocación remota puede fallar por razones relativas a que el objeto invocado está en un proceso o computador diferente del objeto que lo invoca. Por ejemplo, el proceso que contiene el objeto remoto pudiera malograrse o estar demasiado ocupado para responder, o pudiera perderse el mensaje resultante de la invocación.
- Es así, que una invocación a un método remoto debiera ser capaz de lanzar excepciones tales como timeouts debidos a la distribución así como aquellos lanzados durante la ejecución del método invocado. Ejemplos de esto último son: un intento de lectura pasado el fin de un archivo, o un acceso a un archivo sin los permisos adecuados.

- CORBA IDL proporciona una notación para las excepciones específicas del nivel de aplicación, y el sistema subyacente genera excepciones estándar cuando ocurren errores debidos a la distribución. Los programas clientes CORBA deben ser capaces de gestionar las excepciones. Por ejemplo, un programa cliente C++ empleará los mecanismos de excepciones de C++.
- Figura 5.4. Un objeto remoto y su interfaz remota



- Figura 5.6. El papel de un proxy y un esqueleto en la invocación de métodos remotos



▼ Implementación de RMI

- **Módulo de comunicación.** Los dos módulos cooperantes realizan el protocolo de petición-respuesta, que retransmite los mensajes de petición y respuesta entre el cliente y el servidor.
- El módulo de comunicación emplea sólo los tres primeros elementos, que especifican: el tipo de mensaje, su **idPeticion** y la referencia remota del objeto que se invoca. El **idMetodo** y todo el empaquetado y desempaquetado es cuestión del software RMI.

- Los módulos de comunicación son responsables conjuntamente de proporcionar una semántica de invocación, por ejemplo, como **máximo una vez**.
- El módulo de comunicación en el servidor selecciona el distribuidor para la clase del objeto que se invoca, pasando su referencia local, que se obtiene del módulo de referencia remota en respuesta al identificador de objeto remoto en el mensaje petición.
- **Módulo de referencia remota.**
 - Traduce las referencias entre objetos locales y remotos.
 - Crea referencias a objetos remotos.
- En cada proceso, el módulo tiene una **tabla de objetos remotos** que almacena la correspondencia entre referencias a objetos locales en ese proceso y las referencias a objetos remotos (cuyo ámbito es todo el sistema).
- **La tabla incluye:**
 - Una entrada para todo objeto remoto implementado por el proceso. Por ejemplo, en la Figura 5.6, el objeto remoto B estará registrado en la tabla del servidor.
 - Una entrada para cada proxy local. Por ejemplo, en la Figura 5.6 el proxy para B estará registrado en la tabla de ese cliente.

- Las acciones del módulo de referencia remota ocurren como sigue:
 - Cuando se pasa un objeto remoto por primera vez, como argumento o resultado, se le pide al módulo de referencia remota que cree una referencia a un objeto remoto, que se añade a su tabla.
 - Cuando llega una referencia a un objeto remoto, en un mensaje de petición o respuesta, se le pide al módulo de referencia remota la referencia al objeto local correspondiente, que se referirá bien a un proxy o a un objeto remoto. En el caso de que el objeto remoto no esté en la tabla, el software RMI crea un

nuevo proxy y pide al módulo de referencia remota que lo añada a la tabla.

- Los componentes del módulo software de RMI llaman a este módulo cuando realizan el empaquetado y el desempaquetado de las referencias a objetos remotos. Por ejemplo, cuando llega un mensaje de petición, se emplea su tabla para encontrar qué objeto local se va a invocar.
- **El software de RMI** consiste en una capa de software entre:
 - los objetos del nivel de aplicación y
 - los módulos de comunicación y de referencia remota.

- Los papeles de los objetos de middleware mostrados en la Figura 5.6 son como sigue:
 - **Proxy:** hace que la invocación al método remoto sea transparente para los clientes. Para ello se comporta como un objeto local para el que invoca; pero en lugar de ejecutar la invocación, dirige el mensaje al objeto remoto.
 - Hay un proxy para cada objeto remoto del que el cliente disponga de una referencia de objeto remoto. La clase de un proxy implementa los métodos de la interfaz remota del objeto remoto al que representa. Esto asegura que las invocaciones al método remoto son adecuadas según el tipo del objeto remoto.
- **Cada servidor tiene un distribuidor y un esqueleto para cada clase que represente a un objeto remoto.**
- **Distribuidor:** El distribuidor recibe el mensaje de petición desde el módulo de comunicación. Emplea el **idMetodo** para seleccionar el método apropiado del esqueleto, pasándole el mensaje de petición. El distribuidor y el proxy emplean los mismos métodos de asignación de cada **idMetodo** para los métodos de la interfaz remota.
- **Esqueleto:** la clase de un objeto remoto tiene un esqueleto, que implementa los métodos interfaz remota. Se encuentran

implementados de forma muy diferente de los métodos del objeto remoto. Un método del esqueleto desempaquetá los argumentos del mensaje de petición, invoca el método correspondiente en el objeto remoto.

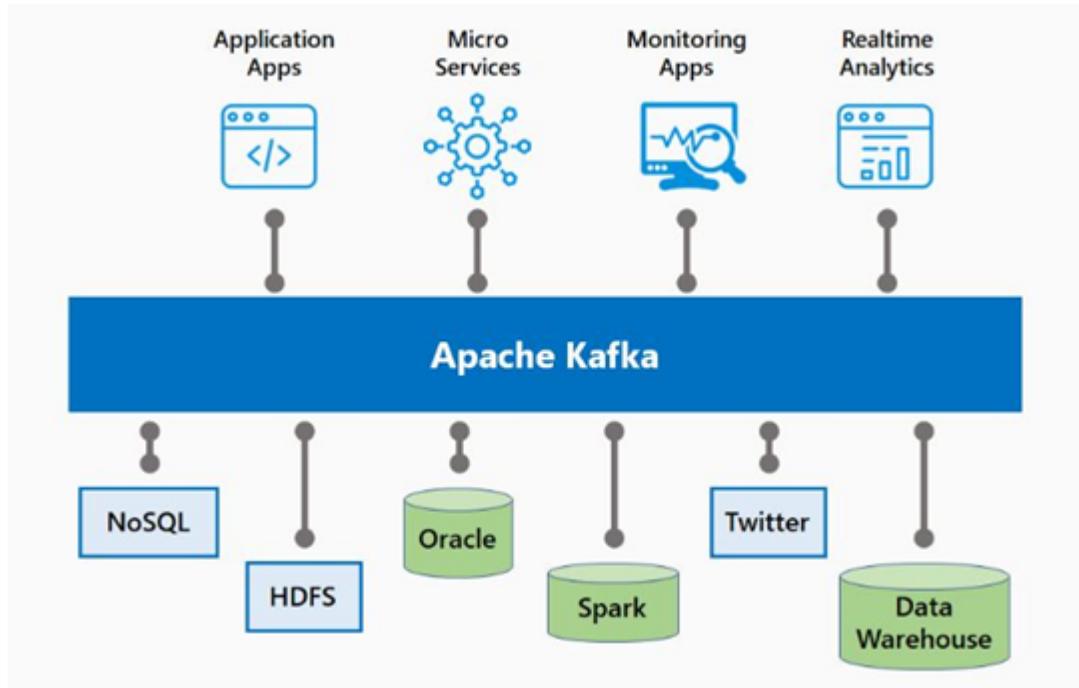
Referencias a Objetos Remotos

- Cuando un cliente invoca un método en un objeto remoto, se envía un mensaje de invocación al proceso servidor que alberga el objeto remoto.
- Este mensaje necesita especificar el objeto particular cuyo método se va a invocar.
- Una **referencia a un objeto remoto** es un identificador para un objeto remoto que es válida a lo largo y ancho de un sistema distribuido.
- En el mensaje de invocación se incluye una referencia a objeto remoto que especifica cuál es el objeto invocado.

▼ Apache Kafka

Apache Kafka es una plataforma de mensajería de suscripción/publicación, transmisión de eventos distribuida de código abierto utilizado ampliamente para encaminar datos de alto rendimiento, análisis de transmisión, integración de datos y aplicaciones de misión crítica.

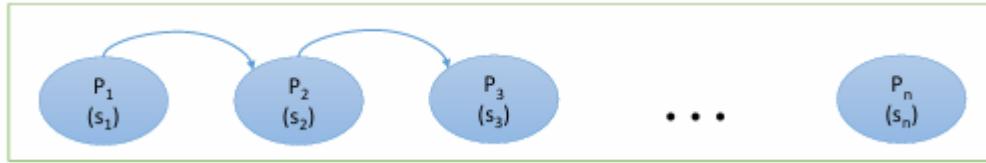
- Intermediario de mensajería (broker de mensajes de alta carga).
- Captura de los cambios de estado de una aplicación.
- Seguimiento de actividad de sitios web.
- Procesamiento de flujo de datos.



▼ Unidad 5: Tiempos y Estados Globales

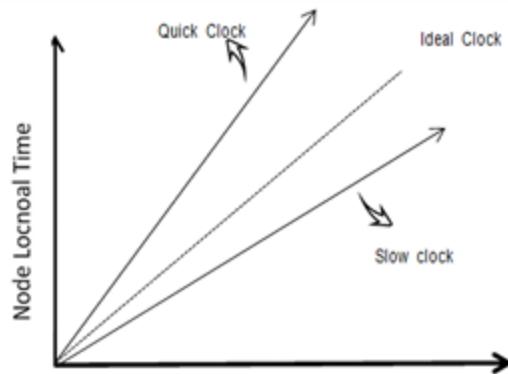
▼ Relojes, eventos y estados de proceso

- Un sistema distribuido consta de una colección Ω de N procesos $i=1, 2, \dots, N$.
- Cada proceso se ejecuta en un único procesador, y los procesadores no comparten memoria.
- Cada proceso p_i en Ω tiene un estado s_i que, en general, se transforma a medida que se ejecuta. El estado del proceso (s_i) incluye los valores de todas sus variables.
- El estado puede incluir también los valores de cualquiera de los objetos en el entorno de su sistema operativo local que lo afecte, como archivos.
- A medida que cada proceso p_i se ejecuta, toma una serie de acciones, cada una de las cuales es un mensaje **Envía** o **Recibe**, o una operación que transforma el estado p_i que cambia uno o más valores de s_i .



- **Evento:** ocurrencia de una única acción que un proceso realiza a medida que se ejecuta. Una acción de comunicación o una acción de transformación del estado.
 - La secuencia de sucesos en un único proceso p_i puede ser colocada en orden único, que se indica con la relación \rightarrow_i entre eventos. $e \rightarrow_i e'$ si y sólo si el evento e ocurre antes del e' en p_i .
- **Historia del proceso p_i :** serie de eventos ordenados que tienen lugar en el proceso p_i , ordenados de la forma que hemos descrito con la relación \rightarrow_i :
 - historia (p_i) = $h_i = < e_i^0, e_i^1, e_i^2 \dots >$.
- **Relojes:** Los computadores pueden disponer de su propio reloj físico. Estos relojes son dispositivos electrónicos que cuentan las oscilaciones que ocurren en un cristal a una frecuencia definida, y que normalmente dividen esta cuenta y almacenan el resultado en un registro contador.
 - El sistema operativo lee el valor del reloj hardware $H_i(t)$ del nodo o proceso p_i , lo escala y añade una compensación para producir un reloj software.
 - $C_i(t) = \alpha H_i(t) + \beta$ que mide aproximadamente el tiempo real, físico t para el proceso p_i .
 - Los relojes de los computadores, como los otros, tienden a no estar en perfecto acuerdo.
- **Sesgo:** diferencia instantánea entre las lecturas de dos relojes cualesquiera.
- **Derivas de reloj:** significa que cuentan el tiempo a diferentes ritmos, y por lo tanto divergen. Los relojes basados en cristal utilizados en los computadores están sujetos a la deriva de reloj.

- Un **ritmo de deriva de reloj** es el cambio en la compensación entre el reloj y un reloj de referencia nominal perfecto por unidad de tiempo, normalmente expresado en partes por millón.
- Para relojes basados en un cristal de cuarzo, suele ser de aproximadamente 10^{-6} segundos/segundo, dando una diferencia de 1 segundo cada 1.000.000 segundos, o 11,6 días. Reloj de cuarzo de alta precisión 10^{-7} o 10^{-8} .

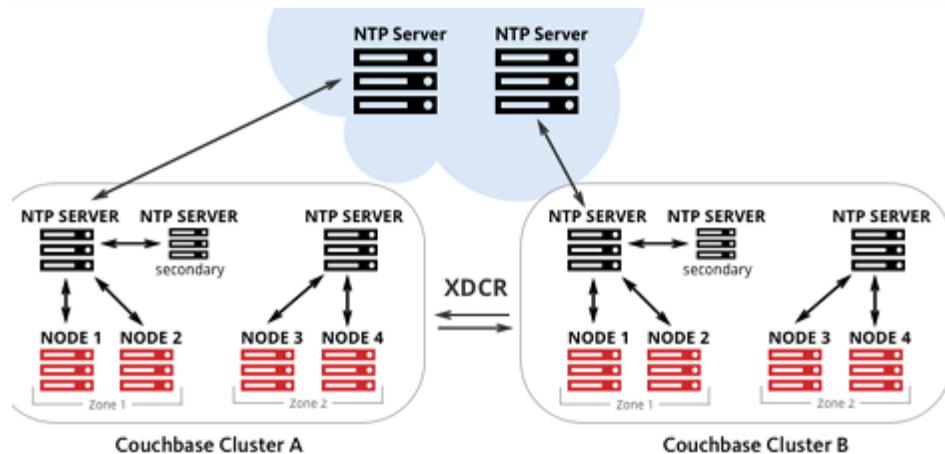


- Los relojes pueden sincronizarse a fuentes externas de tiempo de gran precisión. Los relojes físicos más precisos utilizan osciladores atómicos cuyo ritmo de deriva es aprox 10^{-13} .
- La salida de relojes atómicos se utiliza como estándar para el tiempo real transcurrido, conocido como Tiempo Atómico Internacional.
- Los segundos, los años y otras unidades de tiempo que nosotros utilizamos están basadas en el tiempo astronómico. Fueron definidos originalmente en términos de la rotación de la tierra sobre su eje y su rotación alrededor del sol.
- Sin embargo, el período de rotación de la tierra sobre su eje va siendo gradualmente más largo. Por lo tanto, el tiempo astronómico y el tiempo atómico tienen una tendencia a separarse.
- **El Tiempo Universal Coordinado UTC es un estándar internacional de cronometraje.** Basado en el tiempo atómico, aunque ocasionalmente se inserta un salto de un segundo (o, más raramente, borrado) para mantenerse en sintonía con el tiempo astronómico.

- Las señales UTC se sincronizan y difunden regularmente desde las estaciones de radio terrestres y los satélites, que cubren muchas partes del mundo. Por ejemplo, en USA la estación de radio WWV difunde señales de tiempo en frecuencias de onda corta.

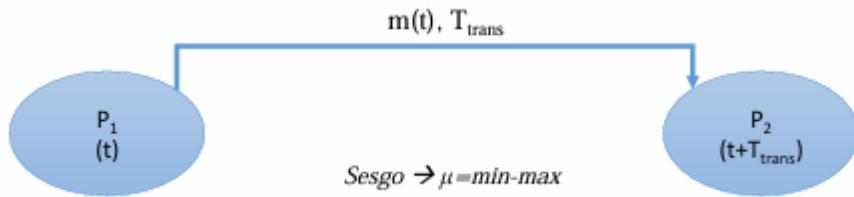
▼ Sincronización de relojes físicos

- Para conocer el tiempo en el que ocurrieron los eventos en un sistema distribuido: es necesario sincronizar los relojes de los procesos.
- **Sincronización externa:** cuando se utiliza una fuente de tiempo externa autorizada (S_i).
 - $|S_i(t) - C_i(t)| < D$ donde D es un límite conocido; $i = 1, 2, \dots, N$.
- **Sincronización interna:** no existe una fuente de tiempo externa y los nodos (procesos) están sincronizados unos con otros con un grado de precisión conocido.
 - $|C_i(t) - C_j(t)| < D$ donde D es un límite conocido; $i, j = 1, 2, \dots, N$.
- Monotonicidad: Si $t' > t$, entonces $C(t') > C(t)$



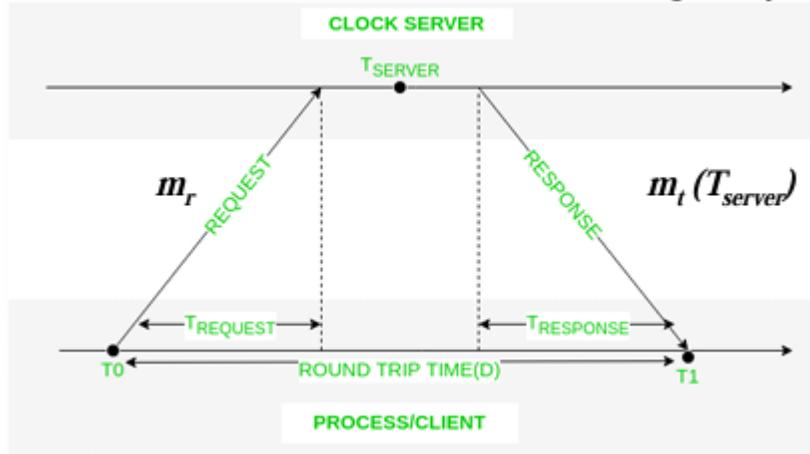
▼ Sincronización de un Sistema Síncrono.

- **Sistemas síncronos:** conocen los límites (mínimo-máximo) de deriva de reloj, retardo de mensajes, y tiempo de ejecución de cada paso de un proceso.



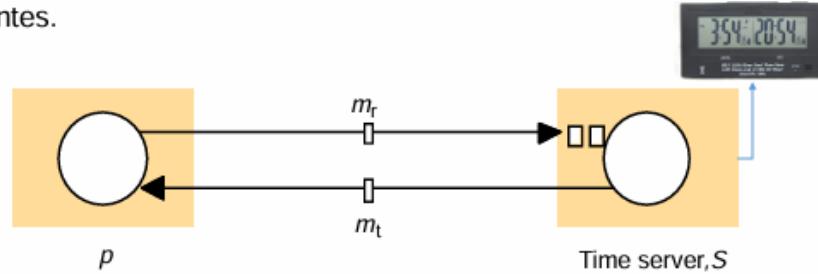
▼ Método de Cristian

- Cristian [1989] sugirió la utilización de un **servidor de tiempo**, conectado a un dispositivo que recibe señales de una fuente de UTC, para sincronizar computadores externamente. Bajo solicitud, el proceso servidor S proporciona el tiempo de acuerdo con su reloj.
- Cristian observó que aunque **no hay límite superior** en los retardos de transmisión de mensajes en un sistema asíncrono, los tiempos de ida y vuelta de los mensajes intercambiados entre cada par de procesos son a menudo razonablemente cortos, una pequeña fracción de un segundo.
- El describe el **algoritmo como probabilístico**: el método consigue sincronización sólo si los tiempos de ida y vuelta entre el cliente y el servidor son suficientemente cortos comparados con la precisión requerida.
- Un proceso p solicita el tiempo en un mensaje m_r y recibe el valor del tiempo t en un mensaje m_t .
- El proceso p registra el tiempo total de ida y vuelta T_{round} tomado para enviar la solicitud m_r y recibir la respuesta m_t . Se puede medir este tiempo con precisión razonable si su ritmo de deriva de reloj es pequeño.



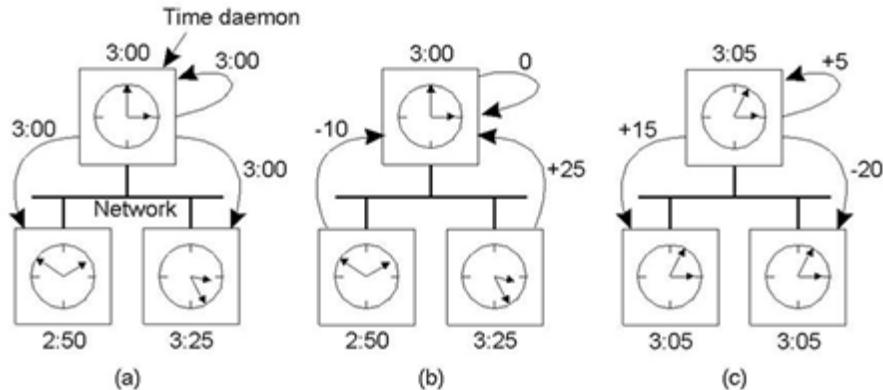
- Una estimación sencilla del tiempo al que p debe fijar su reloj es $t_p = t_{server} + t_{round}/2$
 - Supone que el tiempo transcurrido se desdoble igualmente antes y después de que S coloque t en m_t . Esto es normalmente una suposición razonable de precisión, a menos que los dos mensajes sean transmitidos sobre redes diferentes.

intes.



▼ Algoritmo de Berkeley

- No existe servidor de tiempo, el algoritmo elige un nodo como **master** y los demás serán **slaves**.

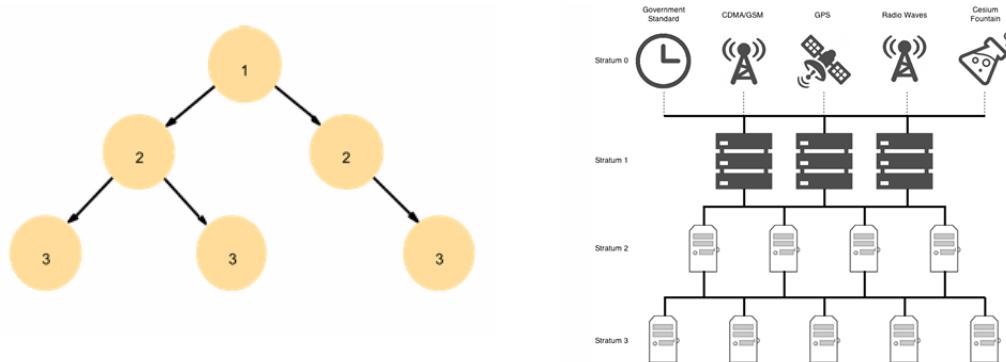


- a) The time daemon asks all the other machines for their clock values
- b) The machines answer
- c) The time daemon tells everyone how to adjust their clock

▼ El protocolo de Tiempo de Red - NTP

- El Protocolo de Tiempo de Red (Network Time Protocol, NTP) define una arquitectura para un servicio de tiempo y un protocolo para distribuir la información sobre Internet.
 - Cristian y Berkeley se aplican a intranets, mientras que NTP se utiliza en internet.
- Los objetivos y las metas principales de diseño de NTP son los siguientes:
 - Proporcionar un servicio que permita a los clientes a lo largo de Internet estar sincronizados de forma precisa a UTC.
 - Proporcionar un servicio fiable que pueda sobrevivir a pérdidas largas de conectividad.
 - Permitir a los clientes re sincronizar con suficiente frecuencia para compensar las tasas de deriva encontradas en la mayoría de los computadores.
 - Proporcionar protección contra la interferencia con el servicio de tiempo, ya sea maliciosa o accidental.
- Los servidores están conectados en una jerarquía lógica llamada una subred de sincronización, cuyos niveles se llaman estratos.
 - Estrato 1: Servidores primarios, en la raíz.

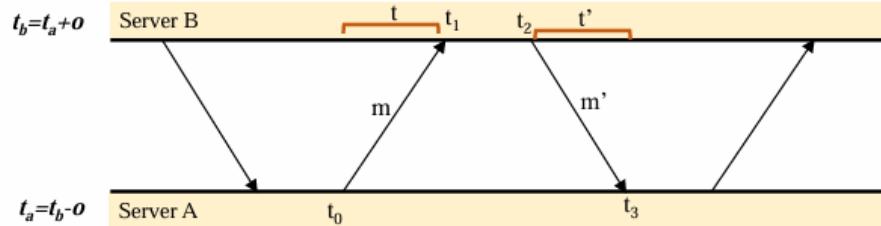
- Estrato 2: Servidores secundarios que están sincronizados directamente con los servidores primarios.
- Estrato 3: Servidores que están sincronizados con los del estrato 2, y así sucesivamente.
- Los servidores del nivel más bajo (hojas) se ejecutan en las estaciones de trabajo de los usuarios.



- Los servidores NTP se sincronizan entre sí en uno de estos tres modos: multidifusión, llamada a procedimiento y modo simétrico.
 - **Multidifusión:** está pensado para su uso en una LAN de alta velocidad. Uno o más servidores reparten periódicamente el tiempo a los servidores que se ejecutan en otros computadores conectados en la LAN, que fijan sus relojes suponiendo un pequeño retardo.
 - **Llamada a procedimiento** es similar al funcionamiento del algoritmo de Cristian. Un servidor acepta solicitudes de otros computadores, que el procesa respondiendo con su marca de tiempo (lectura actual del reloj).
 - **Simétrico:** está aplicado en los estratos más bajos (niveles más altos). Se usa entre servidores NTP para sincronizarse entre sí, como mecanismo de respaldo cuando no pueden alcanzar el servidor NTP (externo).

- En todos los modos, los mensajes se entregan de modo no fiable, utilizando el protocolo de transporte estándar Internet UDP.

NTP Simétrico

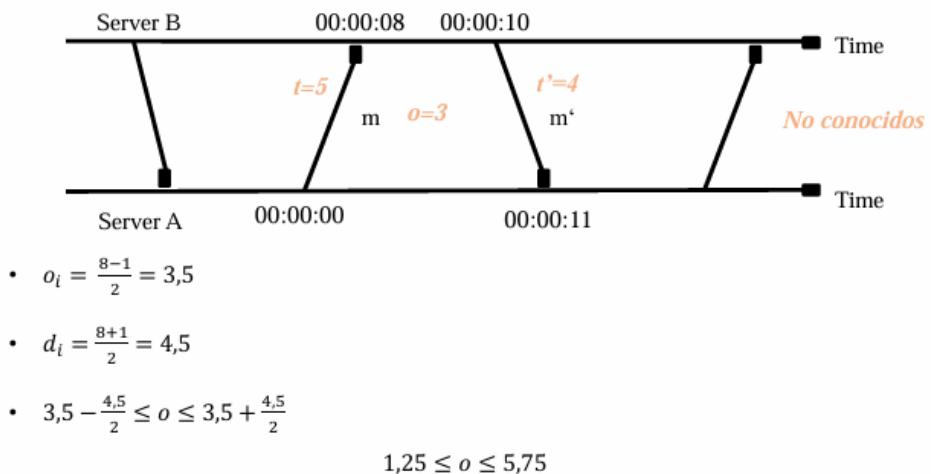


- Los mensajes llegan con un retardo de transmisión d (t y t') y con desplazamiento de B respecto a A (offset o)

$$o_i - \frac{d_i}{2} \leq o \leq o_i + \frac{d_i}{2}$$

- Retardo:* $d_i = t + t' = t_1 - t_0 + t_3 - t_2$
- Offset:* $o_i = (t_1 - t_0 + t_2 - t_3)/2$

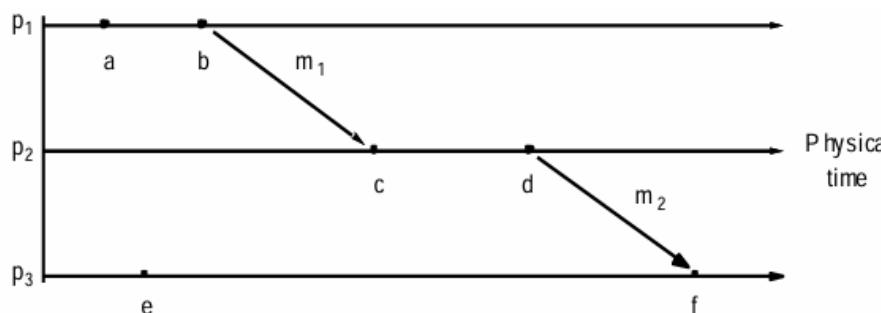
NTP simétrico



▼ Tiempo lógico y Reloj lógicos

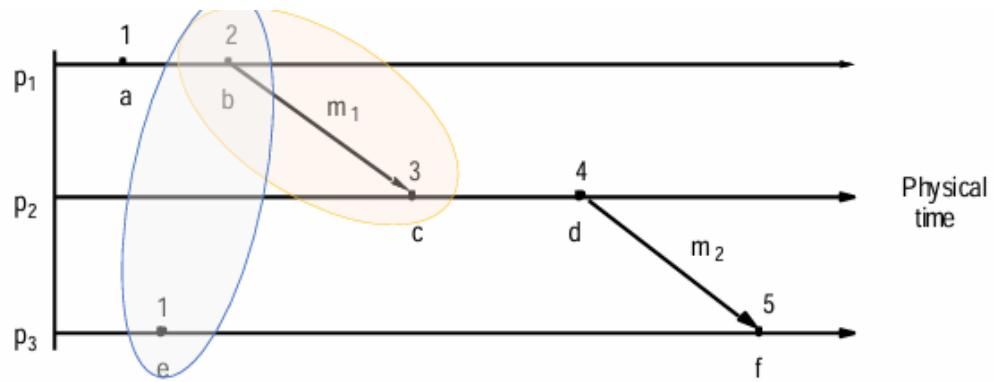
- Desde el punto de vista de un único proceso, los sucesos están ordenados de forma única por los tiempos mostrados en el reloj lógico.
- Sin embargo, como apuntó Lamport [1978], puesto que no podemos sincronizar perfectamente los relojes a lo largo de un sistema distribuido, no podemos usar, en general, el tiempo físico para obtener el orden de cualquier par arbitrario de sucesos que ocurran en él.

- En general, podemos utilizar un esquema que es similar a la causalidad física, pero que se aplica en los sistemas distribuidos, para ordenar algunos de los sucesos que ocurren en diferentes procesos.
- Esta ordenación está basada en dos puntos sencillos e intuitivamente obvios:
 - Si dos sucesos han ocurrido en el mismo proceso p_i ($i = 1, 2, \dots, N$), entonces ocurrieron en el orden en el que les observa p_i , este es el orden \rightarrow_i que hemos definido anteriormente.
 - Cuando se envía un mensaje entre procesos, el suceso de enviar el mensaje ocurrió antes del de recepción del mismo.
- Lamport llamó a la ordenación parcial obtenida al generalizar estas dos relaciones la **realización "sucedió antes"**. Esto también se conoce a veces como la **relación de orden causal** o **ordenación causal potencial**.
- Reglas:
 - SA1: $\exists p_i : e \rightarrow_i e'$ entonces $e \rightarrow e'$.
 - SA2: $\forall m, \text{envía } m \rightarrow \text{recibe } m$
 - SA3: Si e, e' y e'' son sucesos tal que $e \rightarrow e'$ y $e' \rightarrow e''$, entonces $e \rightarrow e''$.



- SA1 => $p_1: a \rightarrow_1 b = a \rightarrow b$
 - SA2 => $b = \text{envía}(m_1); c = \text{recibe}(m_1) = b \rightarrow c$
 - SA3 => $b \rightarrow c \rightarrow d \rightarrow f = b \rightarrow f$
- ¿Y que pasa con a y e?

- **Lamport** inventó un mecanismo simple por el que la relación "sucedió antes" puede capturarse numéricamente, denominado **reloj lógico**. **Un reloj lógico de Lamport es un contador software que se incrementa monótonamente, cuyos valores no necesitan tener ninguna relación particular con ningún reloj físico.**
- Cada proceso p_i mantiene su propio reloj lógico, L_i , que él utiliza para aplicar las llamadas marcas de tiempo de Lamport a los sucesos.
- Representamos la marca de tiempo del suceso e en p_i por $L_i(e)$, y representamos por $L(e)$ la marca de tiempo del suceso e cualquiera que sea el proceso en el que ocurrió.
- Para capturar la relación "sucedió antes" →, los procesos actualizan sus relojes lógicos y transmiten los valores de sus relojes lógicos en mensajes como sigue:
 - RL1: L_i se incrementa antes de emitir cada suceso en el proceso $p_i : L_i = L_i + 1$.
 - RL2:
 - (a) Cuando un proceso p_i envía un mensaje m , acarrea en m el valor de $t = L_i$.
 - (b) Al recibir (m, t) , cada proceso p_i calcula $L_j = \max(L_j, t)$ y entonces aplica RL1 antes de realizar la marca de tiempo del suceso $receive(m)$.



- El incremento puede ser con cualquier otro número
- Por inducción: $b \rightarrow c \Rightarrow L(b) < L(c)$
- Pero si $L(e) < L(e')$ no podemos inferir que $e \rightarrow e'$

▼ Relojos lógicos totalmente ordenados

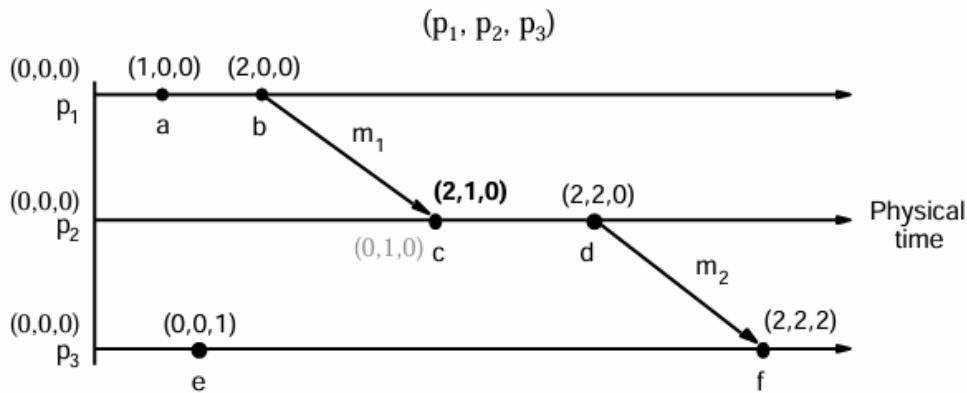
- Algunos pares de sucesos distintos, generados por diferentes procesos, tienen marcas de tiempo de Lamport numéricamente idénticas.
- Podemos crear un orden total sobre los sucesos, esto es, uno para el que todos los pares de sucesos distintos están ordenados, teniendo en cuenta los identificadores de los procesos en los que ocurren los sucesos.
 - Si e es un suceso que ocurre en p_i con marca de tiempo local T_i
 - Si e' es un suceso que ocurre en p_j con marca de tiempo local T_j .
- Definimos las marcas de tiempo globales para esos sucesos como
- (T_i, i) y (T_j, j) respectivamente, y definimos $(T_i, i) \leq (T_j, j)$ si y solo si:
 - $T_i < T_j$
 - $T_i = T_j$ siendo $i < j$.
- Relojos totalmente ordenados del ejemplo anterior: $H = < a_1^1, e_3^1, b_1^2, c_2^3, d_2^4, f_3^5 >$.

▼ Reloj Vectoriales

- Mattern [1989] y Fidge [1991] desarrollaron relojes vectoriales para vencer la deficiencia de los relojes de Lamport: del hecho que $L(e) < L(e')$ no podemos deducir que $e \rightarrow e'$.
- Un reloj vectorial para un sistema de N procesos es un vector de N enteros.
- Cada proceso mantiene su propio reloj vectorial V_i , que utiliza para colocar marcas de tiempo en los sucesos locales.
- Como las marcas de tiempo de Lamport, cada proceso adhiere el vector de marcas de tiempo en los mensajes que envía al resto, y hay unas reglas sencillas para actualizar los relojes.

Reglas de los relojes vectoriales:

- RV1: Inicialización. $V_i[j] = 0$ para $i, j = 1, 2, \dots, N$.
- RV2: Justo antes de que p_i coloque una marca de tiempo en un suceso, coloca $V_i[i] = V_i[i] + 1$.
- RV3: p_i incluye el valor $t = V_i$ en cada mensaje que envía.
- RV4: Cuando p_i recibe una marca de tiempo t en un mensaje, establece $V_i[j] = \max(V_i[j], t[j])$ para $j = 1, 2, \dots, N$. Esta operación de mezcla toma el vector máximo entre los dos, componente a componente.
- Sea $V(e)$ el vector de marcas de tiempo aplicadas por el proceso en el que ocurre e . Es sencillo mostrar, por inducción sobre la longitud de cualquier secuencia de sucesos que relacione los sucesos e y e' , que si $e \rightarrow e'$, entonces $V(e) < V(e')$.

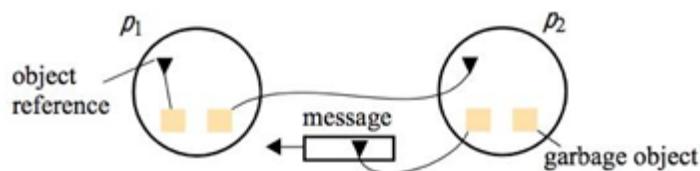


- Por inducción: $a \rightarrow f \Rightarrow V(a) < L(f)$ y también podemos inferir lo inverso
 $V(a) < L(f) \Rightarrow a \rightarrow f$
- No podemos verificar $V(e) \leq V(c)$ ni $V(c) \leq V(e)$, por tanto $e \parallel c$

▼ Estados Globales

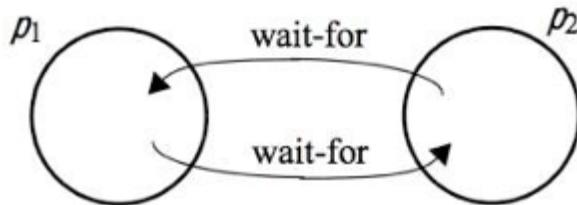
▼ Compactación automática de memoria:

- Un objeto se considera desecharable si no hay posteriormente ninguna referencia a él desde cualquier parte del sistema distribuido. La memoria ocupada por el objeto puede ser reclamada, una vez que se sabe que éste es desecharable. Para comprobar, debemos verificar que no hay referencias a él en cualquier parte del sistema.
- El proceso p_1 tiene dos objetos, ambos con referencias; uno tiene una referencia al propio p_1 y el otro tiene una referencia a p_2 .
- El proceso p_2 tiene un objeto desecharable, al que no existe ninguna referencia en el sistema. Hay también otro objeto en p_2 al que ni p_1 ni p_2 tienen acceso, pero hay una referencia a él en un mensaje que transita entre ambos. Esto nos muestra que cuando consideramos las propiedades de un sistema, debemos incluir el estado de los canales de comunicación así como el de los procesos.



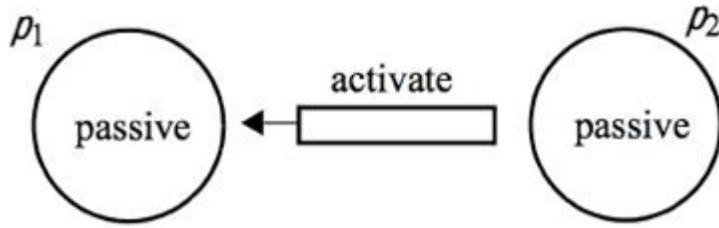
▼ Detección distribuida de bloqueos indefinidos:

- Un bloqueo indefinido distribuido ocurre cuando cada uno de los procesos de una colección espera por otro proceso para enviarle un mensaje, y donde hay un ciclo en el grafo de esta relación "espera por".
- En la figura se muestra que cada uno de los procesos p_1 y p_2 espera un mensaje del otro, por lo que el sistema nunca progresará.



▼ Detección de la terminación distribuida:

- El problema aquí es detectar que un algoritmo distribuido ha terminado. La detección de la terminación es un problema que parece muy fácil de resolver: al principio parece sólo necesario comprobar si cada proceso se ha detenido.
- Para ver que esto no es así, consideremos un algoritmo distribuido ejecutado por dos procesos p_1 y p_2 , cada uno de los cuales puede necesitar valores del otro.
- Instantáneamente, podemos encontrar que un proceso es activo o pasivo. Un proceso pasivo no está ligado a ninguna actividad por sí mismo, pero está preparado para responder con un valor solicitado por el otro. Supongamos que descubrimos que p_1 y p_2 están pasivos. ¿Podemos afirmar que el algoritmo ha terminado?

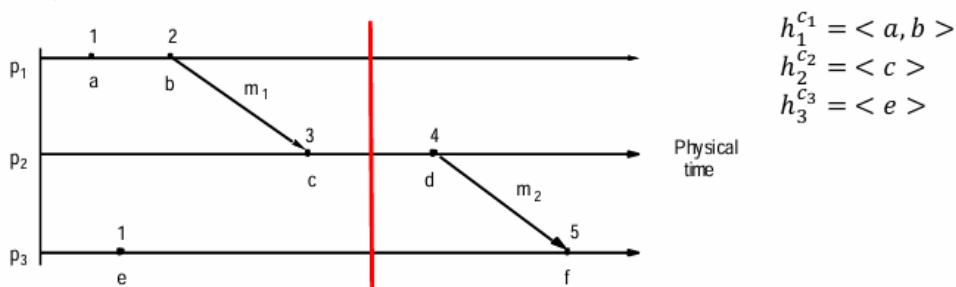


▼ Cortes Consistentes

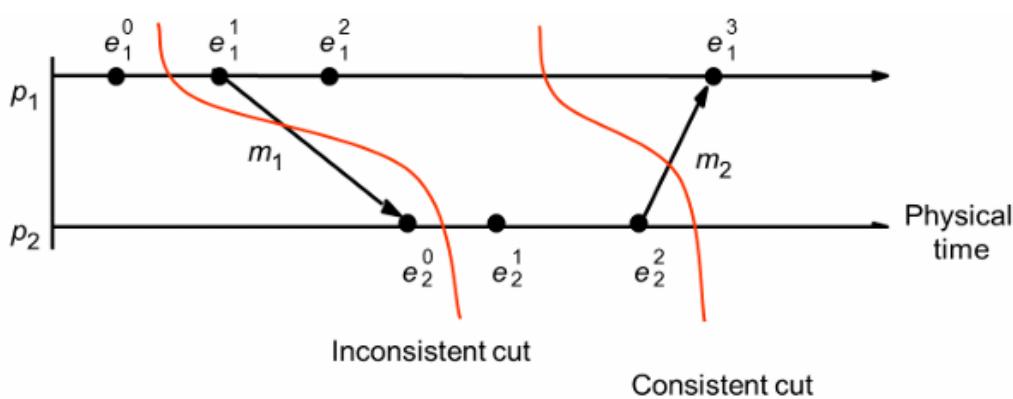
- En principio, es posible observar la sucesión de estados de un proceso individual, pero cómo establecer un estado global del sistema, el estado de la colección de procesos, es más difícil de tratar.
- El problema esencial es la ausencia de **tiempo global**. **Si todos los procesos tuvieran los relojes perfectamente sincronizados, ellos podrían estar de acuerdo en un tiempo en el que cada proceso debía registrar su estado, el resultado sería un estado global actual del sistema**. Si es así, podremos decir, por ejemplo, qué procesos estaban bloqueados indefinidamente, cuáles objetos son desecharables, etc.
- No podemos tener sincronización perfecta de los relojes, por lo que este método no es posible.
- Entonces... ¿podríamos tener un estado global significativo con los estados locales, aún cuando no tengamos sincronización perfecta?
- La respuesta es Sí, pero necesitamos algunas definiciones:
 - Ya habíamos definido la historia de un proceso
 - $historia(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2 \dots e_i^k \dots \rangle$.
 - Podemos formar la historia global de Ω como la unión de las historias individuales de los procesos:
 - $H = h_1 \cup h_2 \cup h_3 \dots \cup h_n$.
 - Matemáticamente, podemos tomar cualquier conjunto de estados de los procesos individuales para formar un estado global $S = (s_1, s_2, \dots, s_n)$.

- Pero ¿qué estados son significativos? Esto es, ¿cuál de los estados de los procesos podrían haber ocurrido al mismo tiempo? Un estado global corresponde a los prefijos iniciales de las historias individuales de los procesos.
 - Un corte de la ejecución del sistema es un subconjunto de su historia global que es la unión de los prefijos de las historias de los procesos:

$$\circ \ C = h_1^{c_1} \cup h_2^{c_2} \cup h_3^{c_3} \dots \cup h_n^{c_n}$$



- El estado s_i en el estado global S correspondiente al corte C es el de p_i inmediatamente después del último suceso procesado por p_i en el corte, $\{e_i^{c_i} : i = 1, 2, 3...n\}$. Este conjunto se llama **la frontera del corte**.
 - La frontera de corte en la fig. es $\langle b, c, e \rangle$
 - **Ejemplos:** Los procesos p_1 y p_2 de la figura muestran dos cortes.



- Un corte consistente incluye los eventos de la frontera de corte y por cada evento de la frontera incluye todos los eventos que sucedieron antes de ese evento.
- Un estado global consistente es uno que corresponde con un corte consistente. Podemos caracterizar un sistema distribuido como una serie de transiciones entre los estados globales del sistema: $S_0 \rightarrow S_1 \rightarrow S_2 \dots$

▼ Algoritmo de instantánea de Chandy y Lamport

- Chandy y Lamport [1985] describen un algoritmo de instantánea para determinar estados globales de sistemas distribuidos.
- **El objetivo es registrar un conjunto de estados de procesos y canales** (una instantánea) **para un conjunto de procesos p_i tal que, incluso a través de una combinación de estados registrados que nunca podrían haber ocurrido al mismo tiempo, el estado global registrado sea consistente.**
- Veremos que el estado que registra el algoritmo de instantánea tiene propiedades convenientes para evaluar predicados del estado global.
- El algoritmo registra el **estado localmente en los procesos**; no proporciona un método para recoger el estado global en un sitio.
- El algoritmo supone que:
 - No fallan ni los canales ni los procesos; la comunicación es fiable por lo que cada mensaje enviado es recibido intacto, exactamente una vez.
 - Los canales son unidireccionales y proporcionan la entrega de los mensajes con ordenación FIFO.
 - El grafo de los procesos y canales está fuertemente conectado (hay un recorrido entre dos procesos cualquiera).
 - Cualquier proceso puede iniciar una instantánea global en cualquier instante.

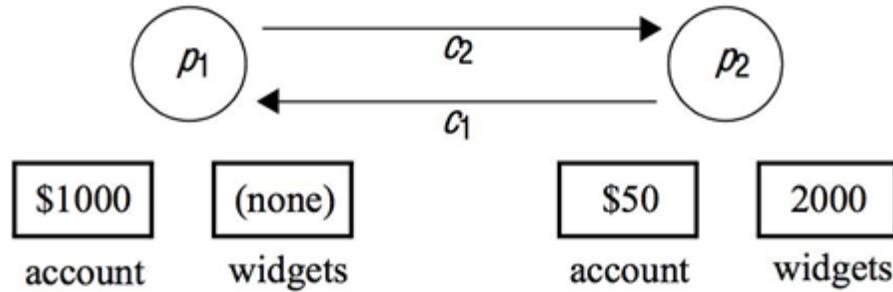
- Los procesos pueden continuar su ejecución y enviar y recibir mensajes normales mientras tiene lugar la instantánea.
- El algoritmo se define mediante **dos reglas**:
 - **La regla de envío del marcador:** obliga a los procesos a enviar un marcador después de haber registrado su estado, pero antes de que envíen cualquier otro mensaje.
 - **La regla de recepción del marcador:** obliga a un proceso que no ha registrado su estado a hacerlo. En ese caso, éste es el primer marcador que ha recibido. Él observará qué mensajes llegan posteriormente en los otros canales entrantes. Cuando un proceso que ya ha guardado su estado recibe un marcador (en otro canal), registra el estado de ese canal bajo la forma del conjunto que recibió desde que guardó su estado.
- Cualquier proceso puede comenzar el algoritmo en cualquier instante. Actúa como si hubiera recibido un marcador y sigue la regla de recepción del marcador. Por tanto, registra su estado y comienza a registrar los mensajes que llegan sobre todos los canales entrantes.

Marker receiving rule for process p_i
 On p_i 's receipt of a *marker* message over channel c :
 if (p_i has not yet recorded its state)
 it records its process state now;
 records the state of c as the empty set;
 turns on recording of messages arriving over other incoming channels;
 else
 p_i records the state of c as the set of messages it has received over c since it saved its state.
 end if

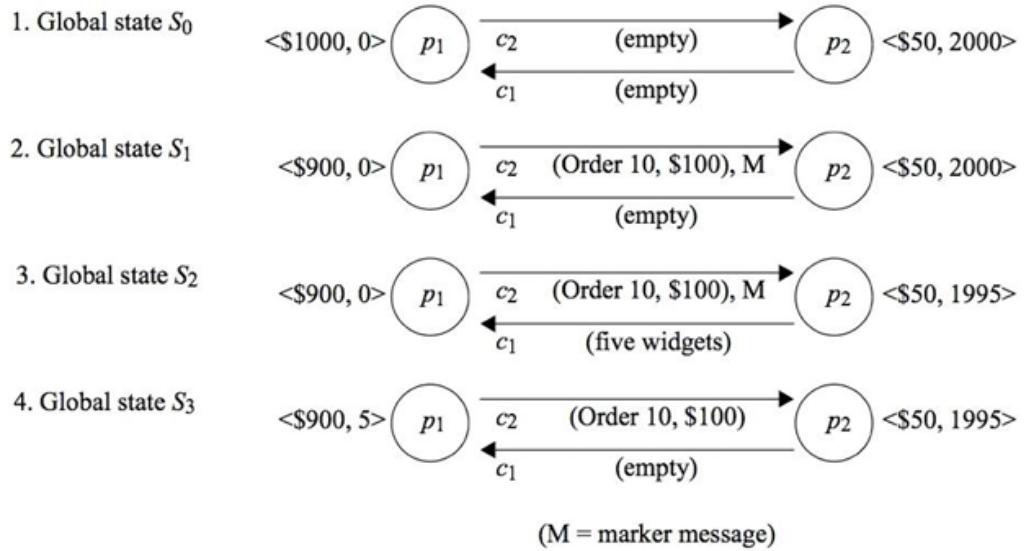
Marker sending rule for process p_i
 After p_i has recorded its state, for each outgoing channel c :
 p_i sends one marker message over c
 (before it sends any other message over c).

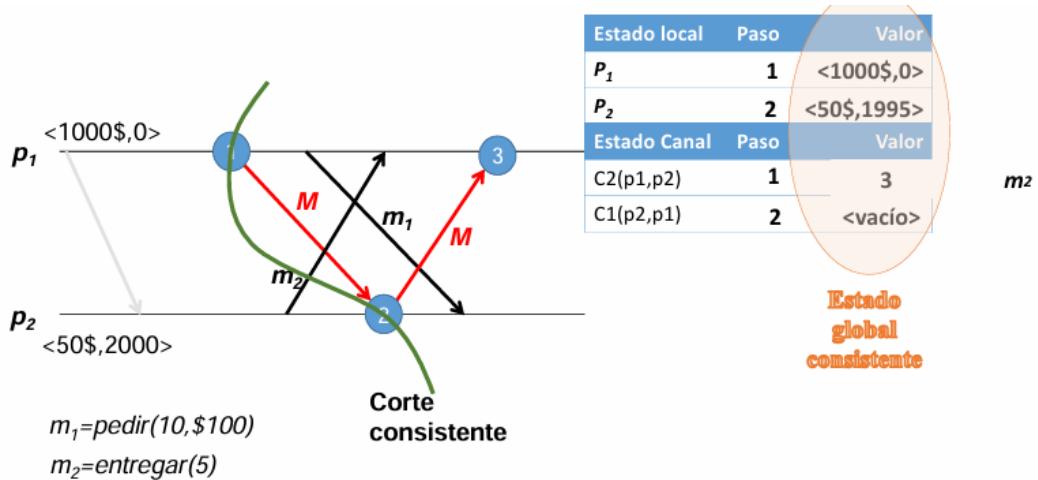
- **Ejemplo:** dos procesos p_1 y p_2 conectados por dos canales unidireccionales c_1 y c_2 . Los dos procesos comercian con artículos. El proceso p_1 envía una orden de compra de artículos sobre c_2 hacia p_2 incluyendo el pago al precio de 10 dólares por artículo.

- Algun tiempo después, el proceso p_2 envía artículos a través del canal c_1 hacia p_1 . Los procesos están en el estado inicial que se ve en la figura. El proceso p_2 ya ha recibido una orden por cinco artículos que serán enviadas en breve a p_1 .

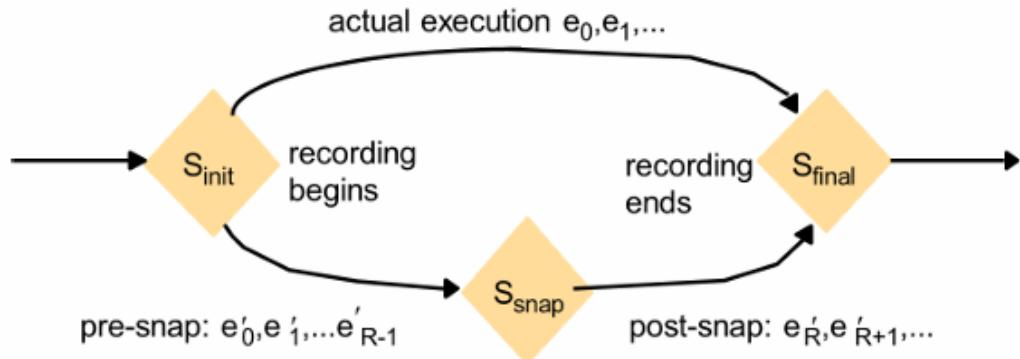


- La ejecución de los procesos se visualiza en la siguiente figura:





- Sea $\text{Sys} = \{e_0, e_1, e_2, \dots, e_{n-1}\}$ el conjunto de eventos ordenados del sistema.
- Sea $\text{Sys}' = \{e'_0, e'_1, e'_2, \dots\}$.



▼ Depuración Distribuida

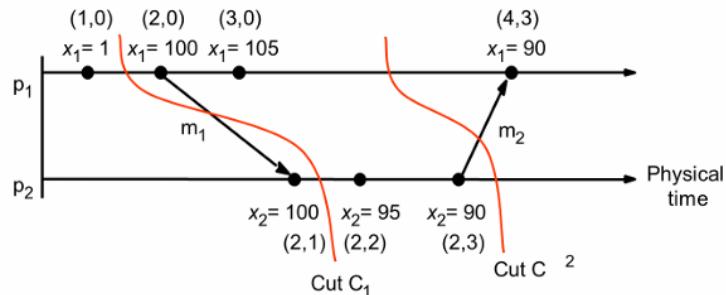
- Los sistemas distribuidos son complejos de depurar.
- **Ejemplo:** Un desarrollador ha escrito una aplicación en la que cada proceso p_i contiene una variable

$x_i (i = 1, 2, \dots, N)$. Las variables cambian a medida que se ejecuta el programa, pero se precisa que estén a menos de cierto valor δ de otras. Pero hay un error en el programa, y el desarrollador sospecha que bajo ciertas circunstancias $|x_i - x_j| > \delta$, incumpliendo la restricción de consistencia.

- Problema: ¿Dónde ocurre eso? ¿En qué proceso/s? ¿Cuáles son las circunstancias?

Predicados:

- La ejecución de un sistema distribuido se puede caracterizar (y depurar) por las transiciones entre estados globales consistentes: $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \dots \rightarrow S_n$.
- Un predicado de estado global es una función
 - $\{\text{Conjunto de estados globales}\} \rightarrow \{\text{V, F}\}$.
 - Determinar una condición del sistema distribuido equivale a evaluar su predicado.
- Características posibles de un predicado:
 - **Estabilidad:** el valor del predicado no varía con los nuevos sucesos (por ejemplo, en el caso de interbloqueo o terminación).
 - **Seguridad:** el predicado tiene valor falso para cualquier estado alcanzable desde S_0 (deseable para errores).
 - **Vitalidad:** el predicado tiene valor verdadero para algún estado alcanzable desde S_0 (deseables para situaciones necesarias).
- Ejemplo: Imaginemos un sistema de 2 procesos donde queremos controlar el predicado $\Phi: |x_i - x_j| > 50$



En $C_1: |1 - 100| > 50 \Rightarrow \Phi \text{ verdadero}$
 En $C_2: |105 - 90| > 50 \Rightarrow \Phi \text{ falso}$

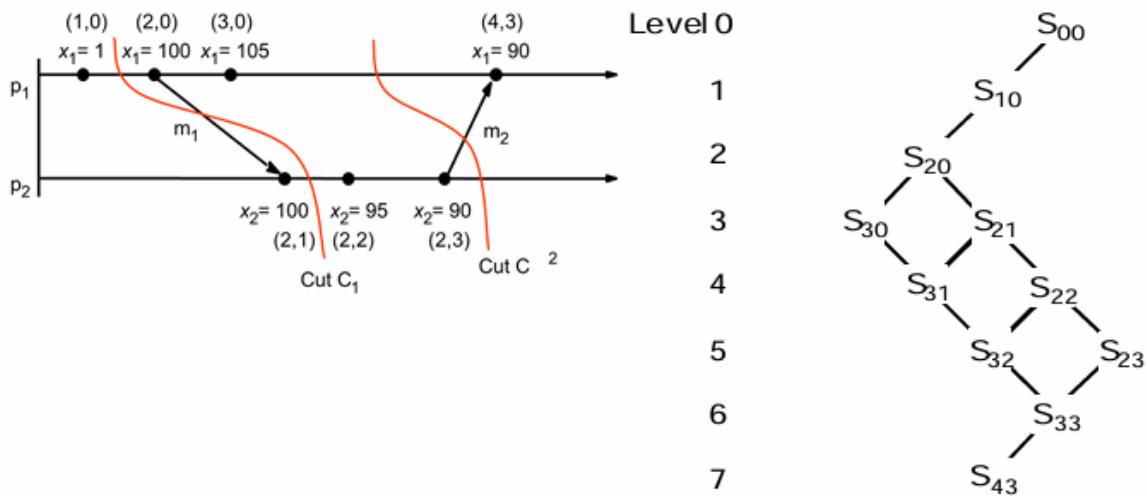
Monitorización:

- Depurar un sistema distribuido requiere registrar su estado global, para poder hacer evaluaciones de predicados en dichos estados.
- Generalmente, la evaluación trata de determinar si el predicado Φ cumple con la condición “posiblemente” o “sin duda alguna”.
- Monitorización del estado global:
 - **Descentralizada:** algoritmo de instantánea de Chandy y Lamport.
 - **Centralizada:** algoritmo de Marzullo y Neiger.
 - Los procesos envían su estado inicial al proceso monitor.
 - Periódicamente, le vuelven a enviar su estado.
 - El monitor registra los mensajes de estado en colas de proceso: una por proceso.

Evaluación de predicados:

- Objetivo de la monitorización:
 - Determinar si un predicado Φ es “posiblemente” o “sin duda alguna” verdadero en un determinado punto de la ejecución.
 - El proceso monitor sólo registra los estados globales consistentes. Son los únicos en que podemos evaluar el predicado con certeza.
- Linealización: es la ruta entre estados.
- La afirmación “**posiblemente Φ** ” significa que hay un estado consistente S a través del cual pasa una linealización de H tal que $\Phi(S)$ es Verdadero.
- La afirmación “**sin duda alguna Φ** ” significa que para todas las linealizaciones L de H , hay un estado consistente S a través del cual pasa L tal que $\Phi(S)$ es Verdadero.

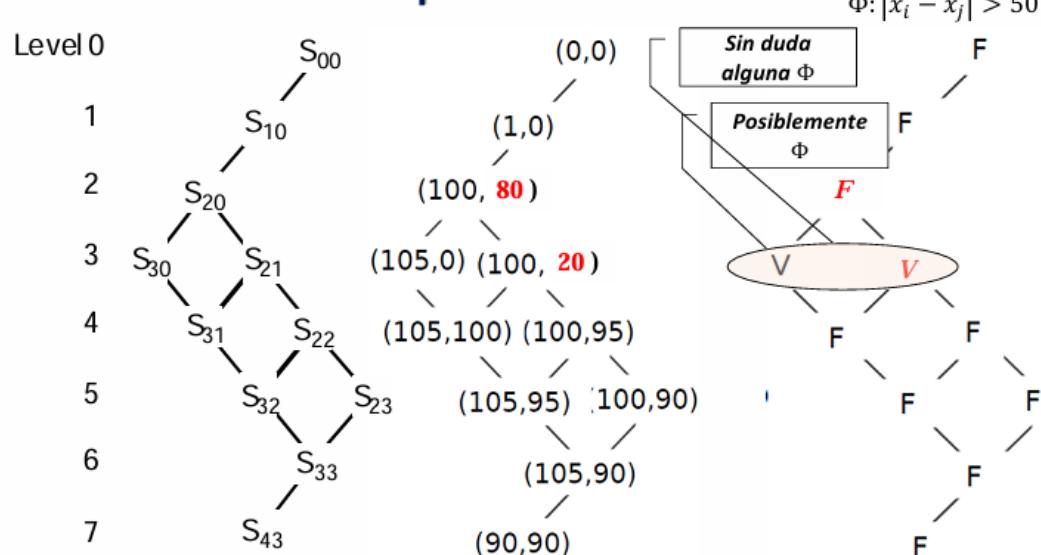
Red de estados globales:



S_{ij} = estado global después de i eventos en el proceso 1 y j eventos en el proceso 2.

Evaluación instantánea de predicados:

Evaluación instantánea de predicados



- Algoritmos para evaluar posiblemente \square y sin duda alguna \square

```

1. Evaluating possibly  $\phi$  for global history  $H$  of  $N$  processes
   $L := 0;$ 
   $States := \{ (s_1^0, s_2^0, \dots, s_N^0) \};$ 
  while ( $\phi(S) = False$  for all  $S \in States$ )
     $L := L + 1;$ 
     $Reachable := \{ S' : S' \text{ reachable in } H \text{ from some } S \in States \wedge \text{level}(S') = L \};$ 
     $States := Reachable$ 
  end while
  output "possibly  $\phi$ ";
```



```

2. Evaluating definitely  $\phi$  for global history  $H$  of  $N$  processes
   $L := 0;$ 
  if( $\phi(s_1^0, s_2^0, \dots, s_N^0)$ ) then  $States := \{ \}$  else  $States := \{ (s_1^0, s_2^0, \dots, s_N^0) \};$ 
  while ( $States \neq \{ \}$ )
     $L := L + 1;$ 
     $Reachable := \{ S' : S' \text{ reachable in } H \text{ from some } S \in States \wedge \text{level}(S') = L \};$ 
     $States := \{ S \in Reachable : \phi(S) = False \}$ 
  end while
  output "definitely  $\phi$ ";
```

▼ Resumen

Depuración distribuida

- **Consiste en:**

- Determinar el predicado que queremos evaluar.
- Especificar un método para construir una red o historia de estados globales consistentes.
 - Teniendo en cuenta el predicado, optimizar el tráfico.
 - Quizás también considerar los canales de comunicación.
- Evaluar si nuestro predicado se cumple en algún momento.
 - Si es *possible*, se cumplirá para alguna de las linealizaciones.
 - Si es *sin duda*, se cumplirá para todas las linealizaciones.

Tiempo y Estados Globales:

- Cualquier medición de tiempo **requiere de referencias**.
- Dichas referencias nunca son totalmente constantes, con lo que las mediciones de tiempo reales **siempre tienen una deriva**.
- Para el **uso diario**, obtenemos la máxima precisión de las mediciones **TAI** y **UT1**, pero ni siquiera estas son precisas.

- Mediante **relojes físicos** podemos hacer sincronización externa a una referencia local (Cristian), global (NTP) o hacer sincronización interna (Berkeley). En cualquier caso, **no son suficientemente precisos** para algoritmos delicados.
- **Lamport** definió las **dos condiciones temporales lógicas** que podemos conocer: el evento de envío ocurre antes que el evento de recepción y los eventos locales ocurren en orden. Estas condiciones permiten establecer **relaciones antes-que** y trabajar mediante **relojes lógicos**.
- El **estado global** de un sistema distribuido se refiere al estado de todos sus procesos y mensajes en tránsito. Un algoritmo para capturarlo es el de **instantánea de Chandy/Lamport**.
- La **depuración distribuida** determina las posibles transiciones entre estados globales, para identificar la posibilidad o certeza de que un determinado evento ocurra.

▼ Unidad 6: Web Services

▼ Introducción

- Los servicios web o **Web Services** por sus siglas en inglés, son interfaces de programación disponibles para la comunicación de aplicaciones en la World Wide Web.
- Proporcionan una forma estándar de interoperabilidad entre diferentes aplicaciones de software que se ejecuta en una variedad de plataformas y/o frameworks.
- Son componentes software que utilizan protocolos abiertos y son independientes y autónomos.
- Proporciona una interfaz de servicio que permite a los clientes interactuar con servidores de una manera más general que los navegadores web.
- Los clientes acceden a las operaciones en la interfaz de un servicio web **mediante solicitudes y respuestas formateadas en XML** y normalmente transmitidas a través de HTTP/S.

- Las interfaces de los servicios web se pueden describir en un IDL. Se debe proporcionar información adicional, incluyendo los protocolos de codificación y comunicación en uso y los lugares de servicio.
- Se necesitan medios seguros para crear, almacenar y modificar documentos e intercambiarlos a través de Internet. Los canales seguros de capa de transporte (TLS) no proporcionan todos los requisitos necesarios. La seguridad XML tiene como objetivo cerrar esta brecha.
- Los servicios web son cada vez más importantes en los sistemas distribuidos: soportan la interoperabilidad a través de Internet, incluyendo el área clave de la integración entre empresas y también la emergente cultura 'mashup'.
- Permite a los desarrolladores crear software creativo e innovador además del existente en base a servicios.
- Los servicios web también proporcionan el middleware subyacente tanto para Grid como para cloud computing.

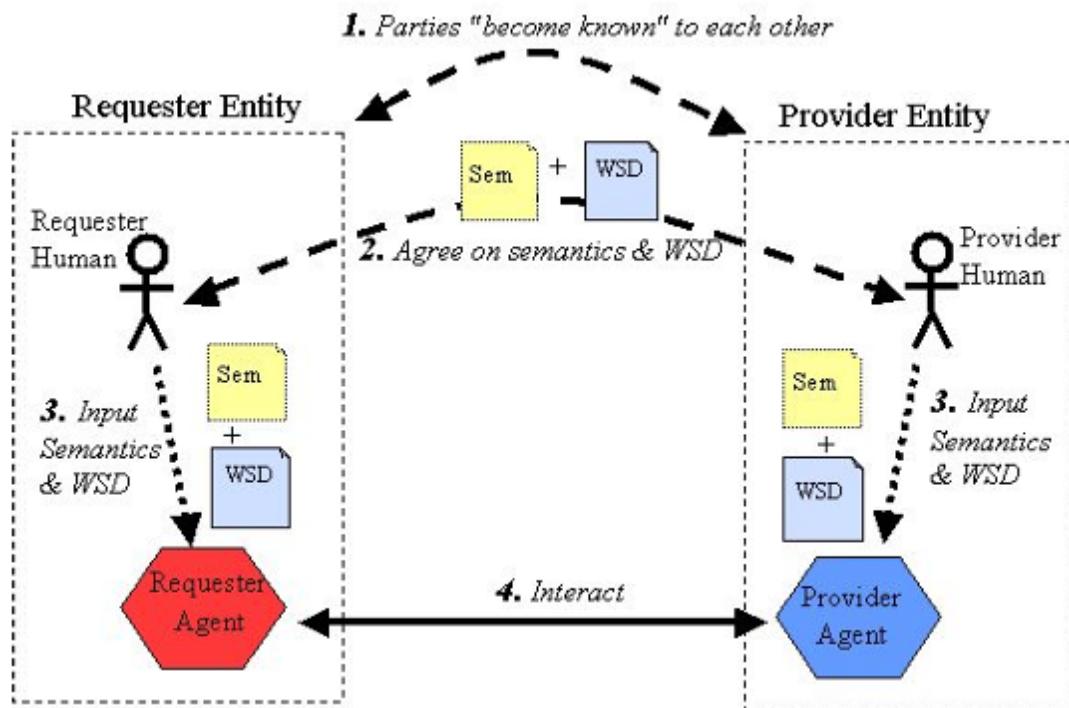
▼ Concepto

- Web Services o Servicio Web, es un sistema de software diseñado para apoyar la **interacción interoperable máquina a máquina** sobre una red de computadoras. Tiene una interfaz descripta en un formato procesable por máquina (específicamente Web Services Description Language WSDL).
- Otros sistemas interactúan con el servicio Web de una manera prescrita por su descripción
 - usando mensajes SOAP (Simple Object Access Protocol)
 - típicamente transmitido a través de HTTP (Hypertext Transfer Protocol)
 - con una serialización XML (Extensible Markup Language) en conjunción con otras normas relacionadas.

▼ URI, URL y URN

- El URI (Uniform Resource Identifier) es un identificador de recurso general, cuyo valor puede ser una URL o una URN.

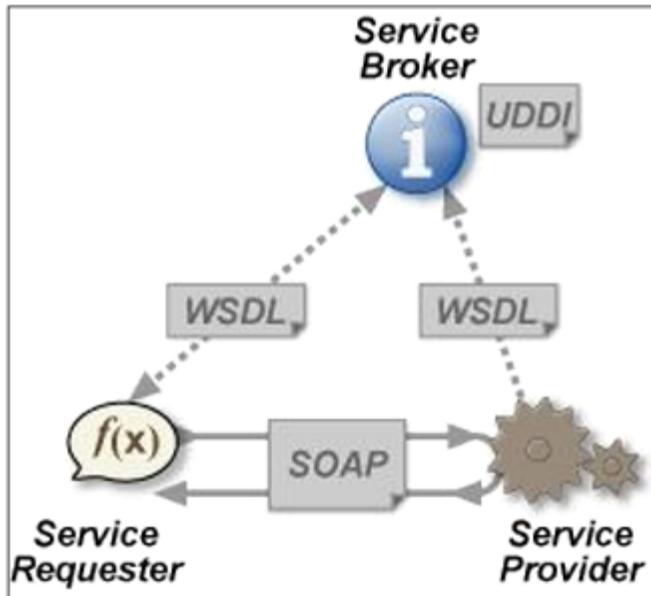
- Las URL incluyen información de ubicación de recursos como el nombre de dominio del servidor de un recurso, son bien conocidas por los usuarios de la web. Ejemplo: <https://www.paraguay.gov.py/>
- Los nombres de recursos uniformes (URN) no dependen de la ubicación; dependen de un servicio de búsqueda para asignarlos a las URL de los recursos. Ejemplo: urn:isbn:0451450523, urn:ISSN:0167-6423.



▼ Servicios SOAP, protocolos: XML, SOAP, WSDL, UDDI

Protocolos:

- Su diseño está basado en mensajes sobre protocolos abiertos, se basan en tecnologías tales como: HTTP, XML, SOAP, WSDL, SPARQL, y otros.



▼ XML – Extensible Markup Language:

- Es un lenguaje extensible de etiquetas desarrollado por el (W3C).
- Permite definir una gramática de lenguajes específicos de forma extensible y personalizable.
- No ha nacido sólo para su aplicación sobre Internet, sino que se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas.
- Se puede usar en bases de datos, editores de texto, hojas de cálculo y casi cualquier aplicación en representación de datos.
- En la actualidad permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.
- Ofrece un formato de datos estándar, flexible y extensible.
- Reduce significativamente la carga de la implementación de las muchas tecnologías necesarias para garantizar el éxito de los servicios Web.
- El conjunto de información XML no es un formato de datos en sí, sino un conjunto formal de elementos de información y sus propiedades asociadas que conforman una descripción abstracta de un documento XML.

▼ **SOAP (Simple Object Access Protocol):**

- SOAP es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML.
- Proporciona un framework (marco de trabajo) estándar y extensible para el empaquetado e intercambio de mensajes XML.
- En el contexto de los Web Services, SOAP también proporciona un mecanismo conveniente para referenciar capacidades o estructuras (normalmente por el uso de los encabezados).
- Los mensajes SOAP pueden ser transportados por una variedad de protocolos de red, tales como HTTP, SMTP, FTP, RMI/IOP, o un protocolo propio de mensajería.
- En la definición, se cuenta con componentes opcionales:
 - Un conjunto de reglas de codificación para expresar instancias de tipos de datos definidos por la aplicación.
 - Una convención para representar llamadas a procedimiento remoto (RPC) y respuestas.
 - Un conjunto de reglas para el uso de SOAP con HTTP.

▼ **WSDL (Web Service Description Language)**

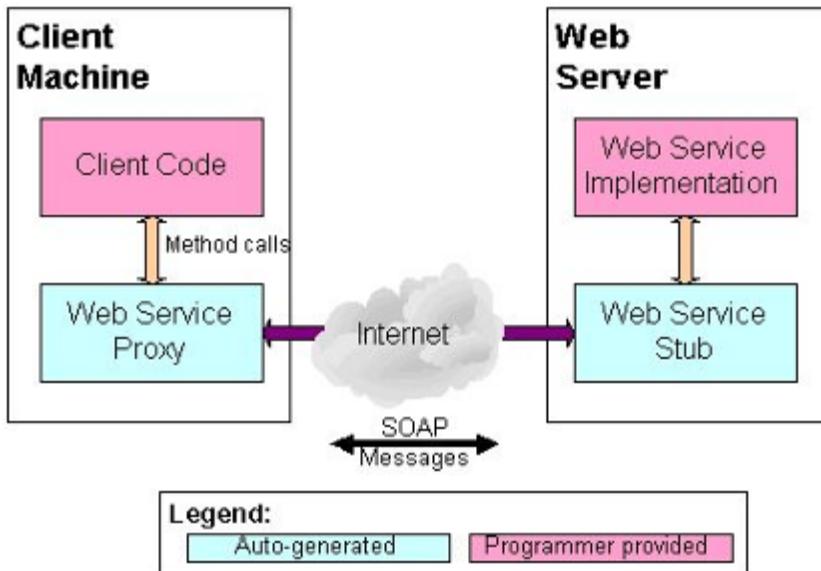
- WSDL (Web Services Description Language) es un lenguaje para describir servicios Web.
- Es un lenguaje de interfaz pública para los servicios Web de los requisitos funcionales necesarios para establecer una comunicación con los servicios Web.
- Describe servicios Web a partir de los mensajes que se intercambian entre los agentes del request (solicitante) y el provider (proveedor). El mensaje en sí se describe de forma abstracta y luego relacionado a un protocolo de red y formato de mensajes concreto.

- Las definiciones de Web Services pueden ser asignadas a cualquier lenguaje de implementación, plataforma, modelo de objetos, o sistema de mensajería.
- Extensiones simples sobre la infraestructura de Internet existente pueden implementar servicios Web para la interacción a través de navegadores o directamente dentro de una aplicación.
- La aplicación puede ser implementada usando librerías como por ejemplo COM, JMS, CORBA, COBOL, etc. Mientras que el emisor y el receptor estén de acuerdo en la descripción del servicio (por ejemplo, el archivo WSDL), detrás de las implementaciones de los servicios Web pueden realizar las acciones que deseen.

▼ **UDDI: Universal Description, Discovery and Integration**

- UDDI es un proyecto propuesto por Ariba, Microsoft e IBM adoptado por OASIS (Organización para el Avance de Estándares de Información Estructurada)
- Es un estándar para registrar y descubrir los Web Services.
- Las empresas registran sus servicios. Dicho registro está basado en XML
- La idea es una imputación en el registro de empresas UDDI, un servicio centralizado y replicado a distintos nodos en forma regular, quedando disponible para ser descubierta por otras empresas.

▼ **Funcionamiento**



▼ Servicios RestFul

- La característica clave de la mayoría de los servicios web es que pueden procesar mensajes SOAP con formato XML. Una alternativa es el enfoque REST. Cada servicio web utiliza su propia descripción de servicio para tratar las características específicas del servicio de los mensajes que recibe.
- **REST (Representational State Transfer):** es un enfoque con un estilo de operación muy limitado, en el que los clientes usan URL y operaciones HTTP: GET, PUT, DELETE y POST para **manipular recursos representados en XML/JSON**.
- El énfasis está en la manipulación de recursos de datos en lugar de en interfaces. Cuando se crea un nuevo recurso, tiene una nueva URL por la que se puede acceder o actualizar.
- Los clientes reciben todo el estado de un recurso en lugar de llamar a una operación para obtener parte de ella. Fielding sostiene que en el contexto de Internet, la proliferación de diferentes interfaces de servicio no será tan útil como un simple conjunto mínimo de operaciones. Es interesante notar que, según Greenfield y Dornan [2004], el 80% de las solicitudes a los servicios web en [Amazon.com](#) son a través de la interfaz REST, y el 20% restante utiliza SOAP.

- GET: permite que el servidor encuentre el recurso solicitado y lo envíe de vuelta.
- PUT: permite indicar al servidor que debe actualizar una entrada en la base de datos del recurso.
- POST: permite que el servidor cree una nueva entrada del recurso.
- DELETE: permite que el servidor elimine un recurso.

▼ WebSocket

Permite la comunicación bidireccional entre un cliente que ejecuta un código no confiable en un entorno controlado y un host remoto que ha optado por recibir comunicaciones de ese código.

El modelo de seguridad utilizado para esto es el modelo de seguridad basado en Origin que utilizan habitualmente los navegadores web.

El protocolo consiste en un protocolo de enlace de apertura seguido de un marco de mensajes básico, distribuido en capas sobre TCP. El objetivo de esta tecnología es proporcionar un mecanismo para aplicaciones basadas en navegador que necesitan una comunicación bidireccional con servidores que no dependa de la apertura de múltiples conexiones HTTP (por ejemplo, utilizando XMLHttpRequest o <iframe>s y sondeos largos).

▼ Casos de Uso: utilidad, aplicaciones, implementaciones

- **Google:** El producto denominado "Google Web API" permite a los desarrolladores poder interrogar y tomar información de casi tres mil millones de documentos Web directamente desde Google. Para lograr eso, Google usa SOAP y WSDL de forma que los desarrolladores puedan programar en su entorno favorito tal como PHP, Java, Perl, o Visual Studio .NET.
- **NetworkSolutions:** Si queremos que nuestro programa sepa a quién pertenece un determinado dominio de Internet, es un Servicio Web prestado por NetworkSolutions.
- **Barnes and Noble:** Si queremos que nuestra aplicación sepa el precio de un determinado libro dado su ISBN, es un Servicio Web que ofrece Barnes and Noble.

- **Dollar rent-a-car:** Automatizó sus sistemas de reservas, que ya se podían hacer vía web, para que se conecten con los de Southwest Airlines, de manera que en el mismo sitio de SWA los pasajeros, al reservar sus pasajes, puedan reservar autos.
- **Líneas Aéreas Escandinavas:** Estas líneas aéreas han desarrollado un servicio web que permite a los usuarios comprar billetes y chequear el estado de los vuelos, mediante el uso del teléfono móvil.
- **Microsoft:** MapPoint.Net. Mediante este servicio, el usuario puede conocer su localización exacta y otros datos adicionales relacionados con su posición actual, como información de tráfico, rutas posibles o puntos comerciales cercanos.
- Los servicios web de Amazon [associates.amazon.com] pueden ser accedidos por SOAP o por REST. Esto permite a las aplicaciones de terceros crear servicios de valor añadido sobre los proporcionados por Amazon.com.
- Por ejemplo, una aplicación de control de inventario y de compras podría ordenar suministros de varios productos según se necesiten de Amazon.com y realizar un seguimiento automático del estado cambiante de cada pedido. Más de 50.000 desarrolladores se registraron para usar estos servicios web en los primeros dos años después de su introducción [Greenfield y Dornan 2004].

▼ Unidad 7: Transacciones y Control de Concurrencia

▼ Introducción

- **Transacción:** define **una secuencia de operaciones que se realiza por el servidor y se garantiza por el mismo que es atómica, ya sea en presencia de múltiples usuarios e incluso de caídas del servidor.**
- Los servidores deben garantizar que se realiza completamente y que los resultados se almacenan en memoria permanente o no.
- Cada transacción de un cliente es considerada invisible desde el punto de vista de otras transacciones; estas no pueden observar los efectos

parciales.

▼ Sincronización simple (sin transacciones):

- Sistemas cuidadosamente diseñados (que implican mayor costo y esfuerzo) aseguran que las operaciones realizadas en nombre de diferentes clientes no interfieren entre sí. En caso de no contemplarlo, esta interferencia puede resultar en valores incorrectos en los objetos. Se detalla cómo se pueden sincronizar las operaciones sin recurrir al manejo de transacciones.
- **Operaciones atómicas en el servidor:**
 - La utilización de múltiples hilos es beneficiosa para las prestaciones.
 - Permite que se ejecuten concurrentemente las operaciones de varios clientes, aún accediendo, posiblemente, a los mismos objetos.
- **Ejemplo:** Si los métodos **deposita** (deposit) y **extrae** (withdraw) no están diseñados para su utilización en un programa multihilo, es posible que las acciones de dos o más ejecuciones concurrentes del método puedan entremezclarse arbitrariamente y tener efectos extraños en las variables de instancia de los objetos **cuenta**.
- En Java, los métodos “**synchronized**” aseguran que solo puede acceder a un objeto un hilo cada vez. Si un hilo invoca un método sincronizado de un objeto, entonces el objeto es bloqueado efectivamente, y otro hilo que invoque uno de sus métodos sincronizados será bloqueado hasta que el bloqueo anterior sea liberado.
- Esta forma de sincronización fuerza a que la ejecución de los hilos sea separada en el tiempo y asegura que las variables instancia de un único objeto sean accedidas de forma consistente.
- Sin sincronización, dos invocaciones separadas de **deposita** podrían leer el balance antes de que ninguna lo hubiera actualizado, lo que produciría un valor incorrecto. Cualquier método que acceda a una variable instancia que pueda variar debe ser sincronizado.
- Las operaciones que están libres de interferencia de operaciones concurrentes que se están realizando en otros hilos se llaman

operaciones atómicas.

- **Mejora de la colaboración del cliente mediante sincronización de las operaciones del servidor:** Los clientes pueden utilizar un servidor como un medio de compartir algunos recursos.
- Esto se consigue por algunos clientes utilizando operaciones para actualizar los objetos del servidor y otros utilizando operaciones para acceder a ellos.
- El esquema anterior para acceso sincronizado a objetos proporciona todo lo que se precisa en muchas aplicaciones, previene que los hilos interfieran unos con otros. Sin embargo, algunas aplicaciones necesitan una forma para que los hilos se comuniquen unos con otros.
- Por ejemplo, **puede surgir una situación en la que la operación requerida por un cliente no puede completarse hasta que se haya realizado otra operación requerida por otro usuario.**
- Esto puede ocurrir cuando algunos clientes son **productores** y otros **consumidores**; los **consumidores** deben esperar hasta que un productor haya proporcionado algún elemento más del artículo en cuestión. **Puede también ocurrir cuando los clientes comparten un recurso; pueden necesitar esperar hasta que otros clientes lo liberen.**
- **Los métodos wait (espera) y notify (notifica) de Java permiten que los hilos se comuniquen con otros de una manera que resuelva los problemas anteriores.**
- Deben utilizarse dentro de los métodos sincronizados de un objeto.
- **Un hilo llama a wait en un objeto para suspenderse él mismo y permitir a otro hilo ejecutar un método en ese objeto.**
- **Un hilo llama a notify para informar a cualquier hilo que esté esperando en el objeto que ha cambiado algunos de sus datos.**
- El acceso a un objeto es todavía atómico cuando un hilo espera por otro: un hilo que llama a wait activa su bloqueo y se suspende como una acción atómica. Cuando el hilo es activado después de ser notificado, adquiere un nuevo bloqueo en el objeto y recupera la ejecución del wait.

▼ Ejemplos:

Ejemplo: Cola compartida entre productor y consumidor

Imaginemos que tenemos una cola que funciona como una especie de "bandeja" para almacenar tareas o elementos.

Hay dos tipos de hilos:

- **Productores** que añaden elementos a la cola (con el método put).
- **Consumidores** que toman elementos de la cola (con el método take).

Para que esta cola funcione bien, debemos evitar que:

- Un productor añada elementos cuando la cola está llena.
- Un consumidor intente tomar un elemento cuando la cola está vacía.

Método put (usando wait y notify)

El método put añade un elemento a la cola.

Sin embargo:

- Si la cola está **llena**, el productor debe **esperar** hasta que haya espacio. Aquí se utiliza wait() para que el hilo se detenga hasta que otro hilo lo "despierte".
- Una vez que hay espacio y el productor añade el elemento, el productor debe llamar a notify() para **despertar** a cualquier consumidor que esté esperando un elemento en la cola.

Método take (usando wait y notify)

El método take toma un elemento de la cola.

Sin embargo:

- Si la cola está **vacía**, el consumidor debe **esperar** hasta que haya un elemento disponible. Aquí se utiliza wait() para que el hilo se detenga hasta que otro hilo lo "despierte".
- Una vez que el consumidor toma el elemento, llama a notify() para **despertar** a cualquier productor que esté esperando espacio en la cola.

```
public class BlockingQueue<T> {  
    private Queue<T> queue = new LinkedList<T>();  
    private int capacity;  
  
    public BlockingQueue(int capacity) {  
        this.capacity = capacity;  
    }  
  
    public synchronized void put(T element) throws InterruptedException {  
        while(queue.size() == capacity) {  
            wait();  
        }  
        queue.add(element);  
        notify();  
    }  
  
    public synchronized T take() throws InterruptedException {  
        while(queue.isEmpty()) {  
            wait();  
        }  
        T item = queue.remove();  
        notify();  
        return item;  
    }  
}
```

Operaciones de la interfaz Cuenta

deposit(amount)
deposit amount in the account
withdraw(amount)
withdraw amount from the account
getBalance() -> amount
return the balance of the account
setBalance(amount)
set the balance of the account to amount

Operations of the Branch interface

create(name) -> account
create a new account with a given name
lookUp(name) -> account
return a reference to the account with the given name
branchTotal() -> amount
return the total of all the balances at the branch

Una transacción bancaria de un cliente

- Un cliente, que realiza una secuencia de operaciones en una cuenta particular del banco por encargo de un usuario, realizará primero una búsqueda (**busca**) de la cuenta por nombre, después aplicará las operaciones de depositar (**deposita**), extraer (**extrae**) y obtener balance (**obténBalance**) directamente sobre la cuenta considerada.
- En nuestros ejemplos, utilizamos cuentas con nombres A, B y C. El cliente las localiza y almacena las referencias a ellas en las variables a, b y c de tipo cuenta (**Cuenta**).

Transaction T:

```
a.extrae(100);  
b.deposita(100);  
c.extrae(200);  
b.deposita(200);
```

- La Figura muestra un ejemplo de la transacción de un cliente que especifica una serie de acciones relacionadas en las que están implicadas las cuentas A, B y C. La primera acción transfiere 100\$ de A a B y la segunda 200\$ de C a B. Un cliente consigue una operación de transferencia realizando un reintegro en una cuenta seguido de un depósito en la otra.
- Las transacciones provienen de los sistemas de gestión de bases de datos. En ese contexto una transacción es la ejecución de un programa que accede a la base de datos.

▼ Transacciones

- Las transacciones pueden venir dadas como una parte del middleware. En estos contextos, una transacción se aplica a objetos recuperables y está pensada para ser atómica. Hay dos aspectos de atomicidad:
 1. **Todo o nada:** Una transacción o finaliza correctamente, y los efectos de todas sus operaciones son registrados en los objetos, o no tiene ningún efecto (si falla o es abortada deliberadamente). Este efecto todo o nada tiene otros dos aspectos en sí mismo:
 - **Atomicidad de fallo:** los efectos son atómicos aun en el caso de ruptura del servidor.
 - **Durabilidad:** después que una transacción ha finalizado con éxito, todos sus efectos son guardados en almacenamiento permanente. Utilizamos el término “almacenamiento permanente” para referirnos a archivos que se mantienen en disco o cualquier otro soporte de datos guardados en disco sobrevivirán incluso en el caso de ruptura del servidor.
 2. **Aislamiento:** **Cada transacción debe ser realizada sin interferencia de otras transacciones, es decir, los efectos intermedios de una transacción no deben ser visibles para los demás.**
 - Para dar soporte al requisito de atomicidad de fallo y durabilidad, los objetos deben ser recuperables. Cuando un proceso servidor cae inesperadamente, debido a un fallo hardware o a un error software, los cambios debidos a todas las transacciones completadas deben estar disponibles en el almacenamiento permanente de forma que cuando el servidor sea reemplazado por un nuevo proceso, se pueden recuperar los objetos para reflejar el efecto todo o nada.
 - Cada vez que un servidor reconoce la finalización de una transacción del cliente, todos los cambios de la transacción en los objetos deben haber sido registrados en almacenamiento permanente.

▼ Propiedades ACID

ACID properties

Härder and Reuter [1983] suggested the mnemonic ‘ACID’ to remember the properties of transactions, which are as follows:

Atomicity: a transaction must be all or nothing;

Consistency: a transaction takes the system from one consistent state to another consistent state;

Isolation;

Durability.

Las propiedades ACID son fundamentales para que las transacciones en sistemas distribuidos se realicen de manera confiable. ACID es un acrónimo de cuatro propiedades:

1. Atomicidad (Atomicity):

Esta propiedad **asegura que una transacción se realice por completo o no se realice en absoluto.** Si ocurre un error en algún paso de la transacción, todos los cambios se revierten, como si nada hubiera pasado.

Ejemplo: Imagina que quieres transferir dinero de una cuenta bancaria a otra. Si ocurre un problema después de que se haya descontado el dinero de la primera cuenta, pero antes de que llegue a la segunda, la transacción debe revertirse para que el dinero no desaparezca.

2. Consistencia (Consistency):

La consistencia **garantiza que las transacciones lleven al sistema de un estado válido a otro también válido, cumpliendo siempre con las reglas establecidas en la base de datos.**

Ejemplo: Supón que en el sistema bancario ninguna cuenta puede tener saldo negativo. La propiedad de consistencia asegura que no se complete ninguna transacción que deje una cuenta con saldo negativo.

3. Aislamiento (Isolation):

Esta propiedad **asegura que cada transacción se ejecute de forma independiente de otras, sin interferencias. Aunque varias transacciones**

se estén ejecutando al mismo tiempo, el resultado será el mismo que si se hubieran ejecutado en secuencia.

Ejemplo: Dos personas intentan transferir dinero de la misma cuenta al mismo tiempo. El aislamiento asegura que cada transacción se procese sin interferencias entre sí, evitando problemas como que ambas retiren el mismo saldo.

4. Durabilidad (Durability):

La durabilidad **asegura que una vez que se confirma una transacción, sus cambios permanecen, incluso si hay un fallo en el sistema o se apaga el servidor.**

Ejemplo: Después de completar una transferencia bancaria, si se apaga el servidor, los cambios (como la reducción del saldo en una cuenta y el aumento en la otra) se mantienen cuando el sistema vuelve a encenderse.

▼ Control de Concurrencia (Problemas)

- **El problema de las actualizaciones perdidas:** Este problema se muestra mediante el siguiente par de transacciones sobre las cuentas bancarias A, B y C, cuyos balances iniciales son 100\$, 200\$ y 300\$ respectivamente. La transacción T transfiere cierta cantidad desde la cuenta A a la cuenta B. La transacción U transfiere otra cantidad desde la cuenta C a la B. En ambos casos, la cantidad transferida se calcula para incrementar el balance de B en un 10%. Los efectos netos sobre la cuenta B al ejecutar las transacciones T y U debieran hacer incrementar el balance de la cuenta B en un 10% dos veces, por lo que el valor final será 242\$.
- Consideremos ahora los 20\$. El resultado es incorrecto, al incrementar el balance de efectos de permitir que las transacciones T y U se ejecuten concurrentemente, como en la Figura 9.5. Ambas transacciones obtienen el balance de B como 200\$ y después depositan la cuenta B en 20\$ en lugar de 42\$. Esto es un ejemplo del problema de las «actualizaciones perdidas». La actualización de U se pierde porque T escribe sin mirar.

Transaction T:	Transaction U:
<i>balance = b.getBalance();</i> <i>b.setBalance(balance*1.1);</i> <i>a.withdraw(balance/10)</i>	<i>balance = b.getBalance();</i> <i>b.setBalance(balance*1.1);</i> <i>c.withdraw(balance/10)</i>
<i>balance = b.getBalance();</i> \$200	<i>balance = b.getBalance();</i> \$200
<i>b.setBalance(balance*1.1);</i> \$220	<i>b.setBalance(balance*1.1);</i> \$220
<i>a.withdraw(balance/10)</i> \$80	<i>c.withdraw(balance/10)</i> \$280

- **Recuperaciones inconsistentes:** La Figura 9.6 muestra otro ejemplo relacionado con una cuenta bancaria en la que la transacción V transfiere una suma desde la cuenta A hasta la B y la transacción W invoca el método totalSucursal para obtener la suma de los balances de todas las cuentas del banco. Los balances de dos cuentas bancarias, A y B, son ambos inicialmente 200\$. El resultado de totalSucursal incluye la suma de A y B como 300\$, que es incorrecto.
- Esto es un ejemplo del problema de las «recuperaciones inconsistentes». Las recuperaciones de W son inconsistentes porque V había realizado sólo la parte de extracción de su transferencia mientras se calculaba la suma.

Transaction V:	Transaction W:
<i>a.withdraw(100)</i> <i>b.deposit(100)</i>	<i>aBranch.branchTotal()</i>
<i>a.withdraw(100);</i> \$100	<i>total = a.getBalance()</i> \$100
<i>b.deposit(100)</i> \$300	<i>total = total+b.getBalance()</i> \$300 <i>total = total+c.getBalance()</i> ⋮

- **Equivalencia secuencial:** Si se sabe que cada una de las distintas transacciones tiene el efecto correcto cuando se realiza ella sola, podemos inferir que si estas transacciones se realizan una cada vez en el mismo orden, el efecto combinado también será correcto.
- Un solapamiento de las operaciones en las que el efecto combinado es el mismo que si las transacciones hubieran sido realizadas una cada vez en el mismo orden es un solapamiento secuencialmente equivalente.
- Cuando decimos que dos transacciones diferentes tienen el mismo efecto, significa que las operaciones devuelven los mismos valores y que las variables de instancia de objetos tienen los mismos valores al final.
- El uso de la equivalencia secuencial como un criterio para la ejecución concurrente previene la ocurrencia de actualizaciones perdidas y recuperaciones inconsistentes.
- El problema de las actualizaciones perdidas ocurre cuando dos transacciones leen el valor antiguo de una variable y la utilizan para calcular el nuevo valor. Esto no puede ocurrir si una transacción se realiza antes que otra, porque la última transacción leerá el valor escrito por la anterior.
- Como un solapamiento secuencialmente independiente de dos transacciones produce el mismo efecto de una secuencial, podemos resolver el problema de las actualizaciones perdidas mediante la equivalencia secuencial.

- Solapamiento 1: La Figura 16.7 (Coulouris 5^a edición) muestra dicho solapamiento en el que las operaciones que afectan a la cuenta compartida B son realmente secuenciales, puesto que la transacción T realiza todas las operaciones antes que las realice la transacción U. Otro solapamiento de T y U que tenga esta propiedad es uno en el que la transacción U finaliza sus operaciones en la cuenta B antes de que comience la transacción T.

Transaction T:		Transaction U:	
<i>balance = b.getBalance()</i>		<i>balance = b.getBalance()</i>	
<i>b.setBalance(balance*1.1)</i>		<i>b.setBalance(balance*1.1)</i>	
<i>a.withdraw(balance/10)</i>		<i>c.withdraw(balance/10)</i>	
<i>balance = b.getBalance()</i>	\$200	<i>balance = b.getBalance()</i>	\$220
<i>b.setBalance(balance*1.1)</i>	\$220	<i>b.setBalance(balance*1.1)</i>	\$242
<i>a.withdraw(balance/10)</i>	\$80	<i>c.withdraw(balance/10)</i>	\$278

- Solapamiento 2: Equivalencia secuencial con relación al problema de las recuperaciones inconsistentes, en la que la transacción V está transfiriendo una suma desde la cuenta A a la cuenta B y la transacción W está obteniendo la suma de todos los balances.
- El problema de las recuperaciones inconsistentes puede ocurrir cuando una transacción de recuperación se ejecuta concurrentemente con una transacción de actualización. No puede ocurrir si la transacción de recuperación se realiza antes o después de una transacción de actualización.
- Un solapamiento secuencialmente equivalente de una transacción de recuperación y una de actualización, por ejemplo como en la Figura 16.8 (Coulouris 5^a edición), impedirá que ocurran las recuperaciones inconsistentes.

Transaction V:		Transaction W:	
<i>a.extrae(100);</i>		<i>unaSucursal.totalSucursal()</i>	
<i>b.deposita(100)</i>			
<i>a.extrae(100);</i>	\$100	<i>total = a.getBalance()</i>	\$100
<i>b.deposita(100)</i>	\$300	<i>total = total+b.getBalance()</i>	\$400
		<i>total = total+c.getBalance()</i>	
		<i>...</i>	

Reglas de conflicto en las operaciones lee y escribe

<i>Operaciones de Diferentes transac.</i>	<i>Conflictos</i>		<i>Causa</i>
<i>read</i>	<i>read</i>	No	Porque el efecto de un par de operaciones de lectura no depende del orden en que se ejecutan.
<i>read</i>	<i>writ e</i>	Yes	Porque el efecto de una operación de lectura y una de escritura dependen del orden en que se ejecutan.
<i>writ e</i>	<i>writ e</i>	Yes	Porque el efecto de un par de operaciones de escritura depende del orden en que se ejecutan.

- Para cualquier par de transacciones, es posible determinar el orden de pares de operaciones conflictivas sobre objetos accedidos por ambas.
- La equivalencia secuencial puede definirse en términos de las operaciones conflictivas como sigue:
 - Para que dos transacciones sean secuencialmente equivalentes, es necesario y suficiente que todos los pares de operaciones conflictivas de las dos transacciones se ejecuten en el mismo orden sobre los objetos a los que ambas acceden.
- Consideremos como ejemplo las transacciones T y U, definidas como sigue:
 - T: $x = \text{lee}(i); \text{escribe}(i, 10); \text{escribe}(j, 20);$
 - U: $y = \text{lee}(j); \text{escribe}(j, 30); z = \text{lee}(i);$
- Consideremos ahora el solapamiento de sus ejecuciones, mostrado en la Figura 9.10. El acceso de cada transacción a los objetos i y j es secuencial con respecto al otro; T realiza todos sus accesos a i antes de que lo haga U y U hace todos sus accesos a j antes de que lo haga T .
- Pero la ordenación no es secuencialmente equivalente, porque los pares de operaciones conflictivas no se hacen en el mismo orden en ambos objetos. Las ordenaciones secuencialmente equivalentes requieren una de las dos condiciones siguientes:

1. T accede a i antes de U y T accede a j antes de U.
2. U accede a i antes de T y U accede a j antes de T.

Un solapamiento de las operaciones de las transacciones T y U no secuencialmente equivalente.

Transaction T:	Transaction U:
$x = \text{lee}(i)$ $\text{escribe}(i, 10)$ $\text{escribe}(j, 20)$	$y = \text{lee}(j)$ $\text{escribe}(j, 30)$ $z = \text{lee}(i)$

- La equivalencia secuencial se utiliza como un criterio para la obtención de protocolos de control de concurrencia.
- Estos protocolos intentan secuenciarizar las transacciones en sus accesos a los objetos. Para el control de concurrencia se utilizan normalmente tres aproximaciones: bloqueo, control de concurrencia optimista, y ordenación por marca de tiempo.

▼ Recuperabilidad de transacciones abortadas

- Los servidores deben registrar los efectos de todas las transacciones finalizadas y ninguno de los efectos de las transacciones abortadas. Deben considerar, por tanto, el hecho de que una transacción pueda ser abortada en previsión de que afecte a otras transacciones concurrentes si es el caso.
- Existen dos **problemas asociados con transacciones abortadas**. Estos problemas se conocen como «**lecturas sucias**» y «**escrituras prematuras**», y ambas pueden aparecer en presencia de ejecuciones secuencialmente equivalentes.
- Se consideran dos categorías de operaciones: operaciones de lectura (lectura) y de escritura (escritura). En nuestras ilustraciones,

obténBalance es una operación de lectura y ponBalance es una de escritura.

- **Lecturas sucias:** La propiedad de aislamiento de las transacciones requiere que éstas no vean el estado no finalizado de las demás. **El problema de la «lectura sucia» está causado por la interacción entre una operación de lectura en una transacción y una operación temprana de escritura en otra transacción sobre el mismo objeto.**
- Considérense las trazas mostradas en la Figura 9.11, donde T consigue el balance de la cuenta A y le añade 10\$ más, a continuación U consigue el balance de A y le añade 20\$ más, y las dos ejecuciones son secuencialmente equivalentes.
- Supongamos ahora que la transacción T aborta después de que U ha finalizado. Entonces, la transacción U habrá visto un valor que nunca ha existido, puesto que se restaurará el valor original en A. En este caso se dice que la transacción U ha realizado una lectura sucia. Como ha finalizado no puede ser deshecha.

Transaction T:	Transaction U:
<i>a.getBalance()</i>	<i>a.getBalance()</i>
<i>a.setBalance(balance + 10)</i>	<i>a.setBalance(balance + 20)</i>
<i>balance = a.getBalance()</i> \$100	<i>balance = a.getBalance()</i> \$110
<i>a.setBalance(balance + 10)</i> \$110	<i>a.setBalance(balance + 20)</i> \$130
<i>commit transaction</i>	
<i>abort transaction</i>	

- **Recuperación de transacciones:** Si una transacción (como U) ha finalizado después de que ha visto los efectos de una transacción que posteriormente ha sido abortada, la situación no es recuperable.
- Para asegurar que tales situaciones no se planteen, cualquier transacción (como U) que esté en peligro de tener una lectura sucia

rechaza la finalización de su operación. **La estrategia para la recuperación es retrasar la finalización hasta después de que haya finalizado cualquier otra transacción cuyo estado no finalizado haya sido observado.**

- En el ejemplo, U retrasa su finalización hasta después de que se produzca la finalización de T. En el caso en que T sea abortada, U debe abortar también.
- **Abortos en cascada:** En la Figura 9.11, suponemos que la transacción U retrasa su consumación hasta después de que T aborde.
- Como hemos dicho, U debe abortar también. Desafortunadamente, si algunas otras transacciones han visto los efectos debidos a U, deberían también ser abortadas. El aborto de estas últimas transacciones puede causar que se aborden todavía más transacciones. Tales situaciones se conocen como abortos en cascada.
- **Para evitar los abortos en cascada, solo se permite a las transacciones leer objetos que fueron escritos por transacciones consumadas.** Para asegurar que esto es así, debe retrasarse cualquier operación de lectura hasta que haya sido consumada o abortada cualquier otra transacción que haya realizado una operación de escritura sobre el mismo objeto. Evitar los abortos en cascada es una condición más fuerte que la recuperabilidad.
- ¿Retrasar la finalización o abortar transacciones en cascada?

- **Escrituras prematuras:** Consideraremos que una transacción pueda abortar. Esto está relacionado con las interacciones entre operaciones de **escritura** en el mismo objeto que se realizan en diferentes transacciones.
- Como ilustración, consideraremos dos transacciones InicializaBalance T y U sobre la cuenta A, como se muestra en la Figura 9.12.
- Antes de las transacciones, el balance de la cuenta A era de 100\$.
- Las dos ejecuciones son secuencialmente equivalentes, con T poniendo el balance a 105\$ y U a 110\$.

- Si la transacción U aborta y T se consuma, el balance debería ser 105\$.
- Algunos sistemas de bases de datos implementan la acción aborta restableciendo las «**imágenes anteriores**» de todas las escrituras de una transacción. En nuestro ejemplo, A es 100\$ inicialmente, que es la «imagen anterior» de la escritura de T; de forma similar, 105\$ es la «imagen anterior» de la escritura de U.
- Por tanto, si U aborta, conseguimos el balance correcto de 105\$.
- Ahora consideremos el caso en que U se consume y después T aborta. El balance debería estar en 110\$, pero la imagen anterior de la escritura de T es 100\$, por lo que conseguimos el balance incorrecto de 100\$.
- De forma similar, si T aborta y después U aborta, la «imagen anterior» de la escritura de U es 105\$, por lo que conseguimos el balance erróneo de 105\$; el balance debería proporcionar 100\$.
- **Para garantizar los resultados correctos en un esquema de recuperación que utiliza las imágenes anteriores, las operaciones de escritura deben ser retrasadas hasta que hayan sido consumadas o abortadas las transacciones anteriores que actualizaron los mismos objetos.**

Transaction T: <i>a.setBalance(105)</i>		Transaction U: <i>a.setBalance(110)</i>
	\$100	
<i>a.setBalance(105)</i>	\$105	<i>a.setBalance(110)</i>

-
- **Ejecuciones estrictas de las transacciones:** Generalmente, se requiere que las transacciones retrasen sus operaciones de lectura y escritura lo suficiente como para impedir tanto las «lecturas sucias» como las «escrituras prematuras».
 - **Las ejecuciones de las transacciones se llaman estrictas si el servicio de las operaciones de lectura y de escritura sobre un objeto se retrasa**

hasta que todas las transacciones que previamente escribieron el objeto han sido consumadas o abortadas.

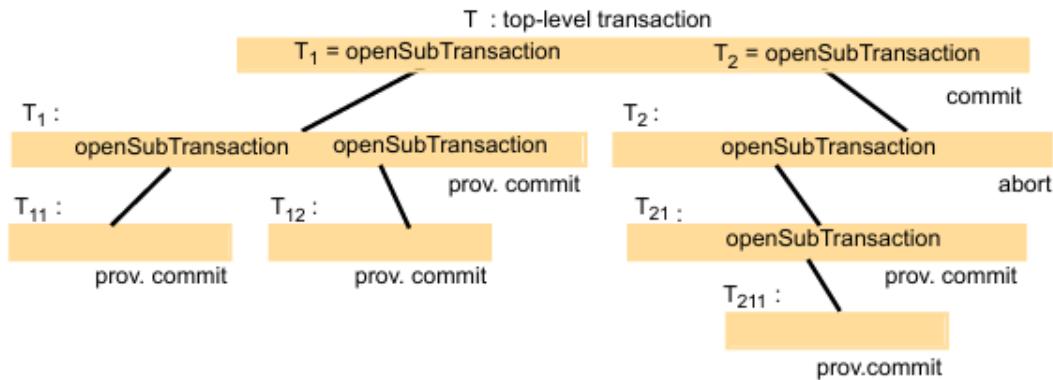
- La ejecución estricta de las transacciones hace cumplir la deseada propiedad de aislamiento.
- **Versiónes provisionales:** Para que un servidor de objetos recuperables participe en las transacciones, debe estar diseñado de forma que las actualizaciones de los objetos puedan ser eliminadas si la transacción aborta.
- **Para esto, todas las operaciones de actualización realizadas durante una transacción se hacen sobre versiones provisionales de los objetos en memoria volátil.**
- A cada transacción se le proporciona su propio conjunto privado de versiones provisionales de cualquiera de los objetos que ella ha alterado.
- Todas las operaciones de actualización de una transacción almacenan valores de los objetos en el propio conjunto privado de la transacción si es posible, o fallan.
- **Las versiones provisionales son transferidas a los objetos solo cuando una transacción se consuma, en cuyo caso ellas serán también registradas en memoria permanente.**
- Esto se realiza en un único paso, durante el cual las otras transacciones son excluidas de acceder a los objetos que están siendo alterados.
Cuando una transacción aborta, sus versiones provisionales se borran.

▼ Transacciones anidadas

- Las **transacciones anidadas extienden el modelo anterior de transacción permitiendo que las transacciones estén compuestas de otras. Desde una transacción se pueden arrancar varias transacciones, permitiendo considerarlas como módulos que pueden componerse cuando se precise.**
- La transacción más exterior en un conjunto de transacciones anidadas se llama la transacción de nivel superior. Las transacciones diferentes

de la del nivel superior se llaman subtransacciones.

- Por ejemplo, en la Figura 16.13, T es una transacción de nivel superior, que inicializa un par de subtransacciones T1 y T2. La subtransacción T1 inicializa su propio par de transacciones T11 y T12. Del mismo modo, la transacción T2 inicia su propia subtransacción T21, que inicia otra subtransacción T211.



- Una subtransacción se presenta como atómica a su padre con respecto a los fallos en la transacción y al acceso concurrente.
- Las subtransacciones del mismo nivel, como T1 y T2, pueden ejecutarse concurrentemente, pero sus accesos a objetos comunes son secuenciados, por ejemplo, mediante un esquema de bloqueo.
- Cada subtransacción puede fallar independientemente de su padre y otras subtransacciones. Cuando la subtransacción aborta, la transacción padre puede elegir una subtransacción alternativa para completar su tarea.
- Por ejemplo, una transacción para entregar un mensaje de correo a una lista de destinatarios podría estar estructurada como un conjunto de subtransacciones, cada una de las cuales entrega el mensaje a uno de los destinatarios.
- Si una o más subtransacciones falla, la transacción padre podría registrar el hecho y después consumarse con el resultado de que todas las subtransacciones hijas con éxito se consuman. Podría empezar

entonces otra transacción que intentara redirigir los mensajes que no fueron enviados la primera vez.

- Las transacciones anidadas tienen las siguientes ventajas principales:
 - **Las subtransacciones en un nivel se ejecutan concurrentemente con otras en el mismo nivel. Esto permite concurrencia adicional.** Cuando las subtransacciones se ejecutan en diferentes servidores, pueden trabajar en paralelo.
 - **Las subtransacciones se pueden consumar o abortar independientemente.** En comparación con una única transacción, un conjunto de subtransacciones anidadas es potencialmente más robusto. El ejemplo anterior de entrega de correo muestra que esto es cierto (con una transacción plana, un fallo en la transacción produciría el que la transacción fuera totalmente reiniciada). **De hecho, un padre puede decidir diferentes acciones de acuerdo con si una subtransacción ha abortado o no.**
- Las **reglas para la consumación de transacciones anidadas** son bastante sutiles:
 - Una transacción se puede consumar o abortar sólo después de que se han completado sus transacciones hijas.
 - Cuando una subtransacción finaliza, hace una decisión independiente sobre si consumarse provisionalmente o abortar. Su decisión de abortar es final.
 - Cuando un padre aborta, todas sus subtransacciones son abortadas.
 - Cuando una subtransacción aborta, el padre puede decidir si abortar o no.
 - Si se consuman las transacciones de alto nivel, entonces todas las subtransacciones que se han consumado provisionalmente pueden consumarse también, proporcionando que ninguno de sus antecesores haya abortado.

- En nuestro ejemplo, la consumación de T permite que se consumen T1, T11 y T12, pero no T21 y T211 puesto que su padre T2 ha abortado. Hay que considerar que los efectos de una subtransacción no son permanentes hasta que no se consuma la transacción de nivel superior.

▼ Bloqueos

- La Figura 9.14 ilustra el uso de bloqueos exclusivos. Muestra las mismas transacciones de la Figura 9.7, pero con una columna extra para cada transacción que representa el **bloqueo**, la **espera** y el **desbloqueo**.

Transaction T:		Transaction U:	
Operations	Locks	Operations	Locks
<i>openTransaction</i>		<i>openTransaction</i>	
<i>bal = b.getBalance()</i>	lock B	<i>bal = b.getBalance()</i>	waits for T's lock on B
<i>b.setBalance(bal*1.1)</i>		<i>• • •</i>	
<i>a.withdraw(bal/10)</i>	lock A	<i>lock B</i>	
<i>closeTransaction</i>	unlock A, B	<i>b.setBalance(bal*1.1)</i>	lock C
		<i>c.withdraw(bal/10)</i>	
		<i>closeTransaction</i>	unlock B, C

- En este ejemplo se supone que cuando las transacciones T y U comienzan, los balances de las cuentas A, B y C no están todavía bloqueados. Cuando la transacción T va a utilizar la cuenta B, B se bloquea para T.
- Consecuentemente, cuando la transacción U va a utilizar B, ésta todavía está bloqueada para T, y la transacción U espera. Cuando se consuma la transacción T, B es desbloqueado, después de lo cual se reanuda la transacción U. El uso del bloqueo en B serializa efectivamente el acceso a B.

- Hay que tener en cuenta que si, por ejemplo, T ha liberado el bloqueo en B entre sus operaciones getBalance y setBalance, la operación getBalance de la transacción U puede solaparse entre ellas.
- La equivalencia secuencial precisa que todos los accesos de una transacción a un objeto particular sean secuenciados con respecto a los accesos por otras transacciones. Todos los pares de operaciones conflictivas de dos transacciones debieran ser ejecutados en el mismo orden.
- Para asegurarse de esto, **no está permitido a una transacción ningún nuevo bloqueo después de que ha liberado uno**.
- La primera fase de cada transacción se conoce como «**fase de crecimiento**», durante la cual se adquieren nuevos bloqueos. En la segunda fase, se liberan los bloqueos (una «**fase de acortamiento**»). Esto se llama bloqueo en dos fases.
- Se utilizan dos tipos de bloqueos: **bloqueos de lectura y bloqueos de escritura**. Antes de realizar una operación de lectura en una transacción, se debe activar un bloqueo de lectura en el objeto. Antes de que se realice una operación de escritura, se debe activar un bloqueo de escritura en el objeto.
- Cuando es imposible activar un bloqueo inmediatamente, la transacción (y el cliente) debe esperar hasta que sea posible hacerlo; tenga en cuenta que nunca se rechaza una solicitud de un cliente.
- Como un par de operaciones de lectura, desde transacciones diferentes, no entra en conflicto, un intento de activar un bloqueo de lectura en un objeto que presente un bloqueo de lectura siempre tendrá éxito. **Todas las transacciones que leen los mismos objetos comparten su bloqueo de lectura. Por esta razón, los bloqueos de lectura se llaman, a veces, bloqueos compartidos.**

Las reglas de conflicto de la operación nos dicen que:

1. Si una transacción T ha realizado ya una operación de lectura en un objeto particular, entonces una transacción concurrente U no debe escribir ese objeto hasta la consumación de T o que aborte.

- Si una transacción T ha realizado ya una operación de escritura en un objeto particular, entonces una transacción concurrente U no debe leer o escribir ese objeto hasta la consumación de T o que aborte.

Compatibilidad de bloqueos

<i>Para un objeto</i>		<i>Bloqueo solicitado</i>	
		<i>Lectura</i>	<i>Escritura</i>
<i>Bloqueo ya activado</i>	<i>Ninguno</i>	OK	OK
	<i>Lectura</i>	OK	Espera
	<i>Escritura</i>	Espera	Espera

Uso de bloqueos en un sistema de bloqueo en dos fases estricto

- Cuando una operación accede a un objeto en una transacción:
 - Si el objeto no estaba ya bloqueado, es bloqueado y comienza la operación.
 - Si el objeto tiene activado un bloqueo conflictivo con otra transacción, la transacción debe esperar hasta que esté desbloqueado.
 - Si el objeto tiene activado un bloqueo no conflictivo de otra transacción, se comparte el bloqueo y comienza la operación.
 - Si el objeto ya ha sido bloqueado en la misma transacción, el bloqueo será promovido si es necesario y comienza la operación. (Donde la promoción está impedida por un bloqueo conflictivo, se utiliza la regla (b).)
- Cuando una transacción se consuma o aborta, el servidor desbloquea todos los objetos bloqueados por la transacción.

▼ Bloqueos indefinidos

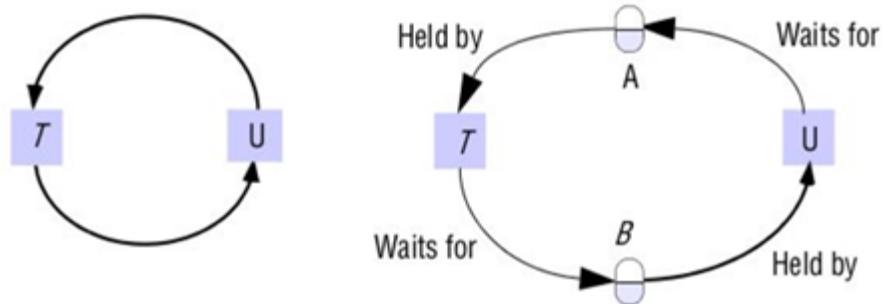
- El uso de bloqueos puede conducir a bloqueos indefinidos. Consideremos el uso de bloqueos representado en la Figura 9.19. Puesto que los métodos deposita y extrae son atómicos, los representamos adquiriendo bloqueos de escritura, aunque en la práctica ellos leen el balance y escriben en él. Cada uno de ellos adquiere un bloqueo y se inmoviliza cuando intenta acceder a la cuenta que ha bloqueado el otro. Ésta es una situación de bloqueo indefinido: dos transacciones están esperando y cada una depende de la otra para liberar un bloqueo y poder reanudarse.

- El bloqueo indefinido es una situación particularmente común cuando los clientes están implicados en un programa interactivo, una transacción en un programa interactivo puede durar un período largo de tiempo, produciendo que muchos objetos queden inmovilizados y permanezcan así, impidiendo por tanto que otros clientes los utilicen.

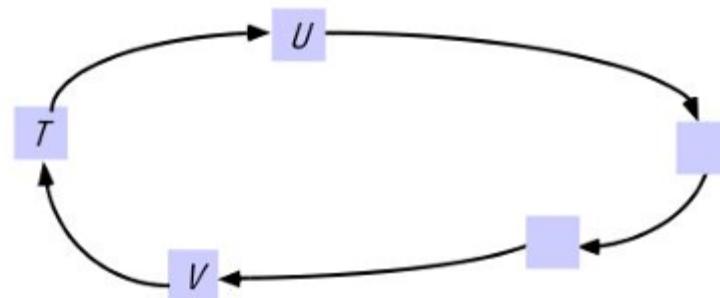
Transaction <i>T</i>		Transaction <i>U</i>	
Operations	Locks	Operations	Locks
<i>a.deposit(100);</i>	write lock <i>A</i>	<i>b.deposit(200)</i>	write lock <i>B</i>
<i>b.withdraw(100)</i>		<i>a.withdraw(200);</i>	waits for <i>T</i> 's
...	waits for <i>U</i> 's lock on <i>B</i>	...	lock on <i>A</i>
...		...	
...		...	

- **Un bloqueo indefinido es un estado en el que cada miembro de un grupo de transacciones está esperando por algún otro miembro para liberar un bloqueo.**
- Se puede utilizar un grafo “espera por” para representar las relaciones de espera entre las transacciones actuales.
- En un grafo espera por, los nodos representan las transacciones y los arcos representan las relaciones espera por entre transacciones. Hay un arco del nodo T al nodo U cuando la transacción T está esperando que la transacción U libere un bloqueo.
- Inspeccionando la Figura 9.20, vemos que el grafo espera por corresponde a la situación de bloqueo indefinido ilustrada en la Figura 9.19.
- Un bloqueo indefinido surge porque las transacciones T y U intentaron cada una adquirir un objeto mantenido por la otra. Por tanto, T espera por U y U espera por T.
- La dependencia entre las transacciones es indirecta, mediante una dependencia de los objetos.

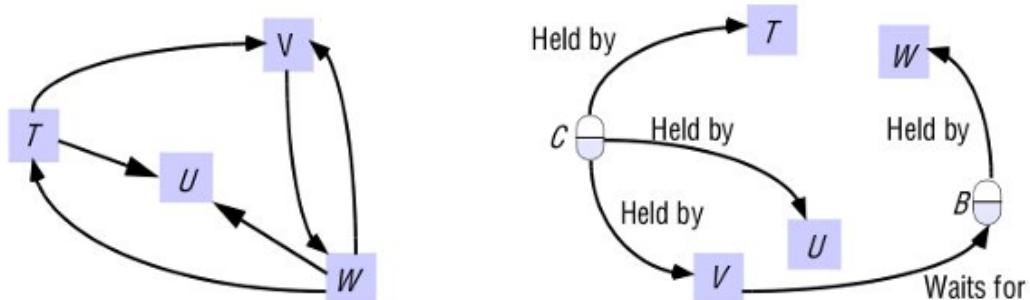
- El diagrama de la derecha muestra los objetos mantenidos y esperados por las transacciones T y U.



A cycle in a wait-for graph



Another wait-for graph



-
- Los timeouts de bloqueo son un método para la resolución de **bloqueos indefinidos** que se utiliza normalmente. A cada bloqueo se le proporciona un período limitado en el que es invulnerable.

- Después de este tiempo, es vulnerable. Supuesto que ninguna otra transacción está compitiendo por el objeto que está bloqueado, un objeto con un bloqueo vulnerable continúa bloqueado.
- Sin embargo, si cualquier otra transacción está esperando para acceder al objeto protegido por un bloqueo vulnerable, el bloqueo se rompe (es decir, el objeto es desbloqueado) y se reanuda la transacción que estaba esperando. **La transacción cuyo bloqueo se ha roto normalmente aborta.** Hay muchos problemas con el uso de timeouts como un remedio para bloqueos indefinidos: el peor problema es que a veces se abortan transacciones debido a que sus bloqueos llegan a ser vulnerables cuando otras transacciones están esperando por ellas, aunque en realidad no hay bloqueo indefinido.

▼ Incrementando la concurrencia en esquemas de bloqueo (Bloqueo de dos versiones)

- **Bloqueo de dos versiones:** Esquema optimista que permite que una transacción escriba versiones tentativas de objetos mientras otras transacciones lean de la versión consumada de los mismos objetos.
- Esta variación del bloqueo en dos fases estricto utiliza tres tipos de bloqueo:
 - un bloqueo de lectura,
 - uno de escritura, y
 - uno de consumación.

Compatibilidad de bloqueos (lectura, escritura, consumación)

Para un objeto		Bloqueo solicitado		
		Lectura	Escritura	consumación
Bloqueo ya activado	Ninguno	OK	OK	OK
	Lectura	OK	OK	ESPERA
	Escritura	OK	ESPERA	
	Consumación	ESPERA	ESPERA	

▼ Control optimista de la concurrencia

- Kung y Robinson [1981] identificaron un número de desventajas inherentes al bloqueo y propusieron una aproximación optimista, alternativa a la secuenciación de transacciones que evita esas desventajas. Desventajas del bloqueo:
 - Mantener el bloqueo representa sobrecarga que no está presente en los sistemas que no soportan acceso concurrente a los datos compartidos.
 - Puede producir un bloqueo indefinido. Su prevención reduce la concurrencia de forma severa, y por tanto las situaciones de bloqueo indefinido deben ser resueltas o por el uso de timeouts o por la detección de bloqueo indefinido. Ninguna de ellas es totalmente satisfactoria para uso en programas interactivos.
 - Para impedir abortos en cascada, los bloqueos no pueden ser liberados hasta el final de la transacción. Esto puede reducir significativamente el potencial de concurrencia.
- La aproximación alternativa propuesta por Kung y Robinson es optimista porque se basa en la observación de que, en la mayoría de las aplicaciones, la similitud entre las transacciones de dos clientes que acceden al mismo objeto es baja. Se permite que las transacciones procedan como si no hubiera posibilidad de conflicto con otras transacciones hasta que el cliente complete su tarea y publique una petición cierraTransacción.
- Cuando aparece un conflicto, habitualmente se abortará alguna transacción y se necesitará reinicializar el cliente.

Cada transacción presenta las siguientes fases:

- **Fase de trabajo:** durante esta fase, cada transacción tiene una versión tentativa de cada uno de los objetos que actualiza. Ésta es una copia de la versión del objeto más recientemente consumada.
 - Las versiones tentativas permiten a la transacción abortar (sin efecto alguno sobre los objetos), tanto durante la fase de trabajo como si falla su validación debido a otras transacciones en conflicto.

- Operaciones de lectura se realizan inmediatamente: si existe una versión tentativa para la transacción, la operación de lectura accederá a ella; de otro modo, accederá al valor más recientemente consumado del objeto.
- La operación de escritura almacena los nuevos valores de los objetos bajo versiones tentativas (las cuales son invisibles al resto de transacciones).
- Cuando hay varias transacciones concurrentes, podrán coexistir valores tentativos diferentes sobre el mismo objeto.
- Además, se almacenan dos registros para cada objeto al que se accede en la transacción:
 - un conjunto de lectura que contiene los objetos leídos por la transacción; y
 - un conjunto de escritura que contiene los objetos modificados por la transacción.
- **Fase de validación:** cuando se recibe la solicitud cierraTransacción, se valida la transacción para establecer si sus operaciones en los objetos entran en conflicto o no con las operaciones en otras transacciones sobre los mismos objetos. Si la validación tiene éxito, entonces se puede consumar la transacción. **Si la validación falla, se debe utilizar alguna forma de resolución de conflictos y bien habrá que abortar la transacción actual o, en algunos casos, aquellas con las que entra en conflicto.**
- **Fase de actualización:** si una transacción está validada, todos los cambios registrados en sus versiones provisionales se hacen permanentes. Las transacciones de solo lectura pueden consumirse inmediatamente después de pasar la validación. Las transacciones de escritura están dispuestas a consumarse una vez que las versiones provisionales de los objetos se hayan grabado en memoria permanente.

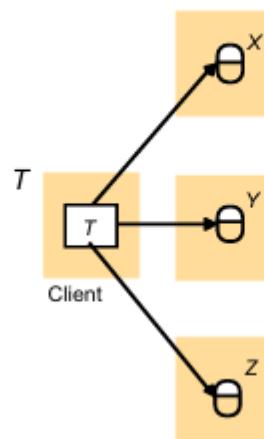
▼ Unidad 8: Transacciones Distribuidas

▼ Transacciones Distribuidas

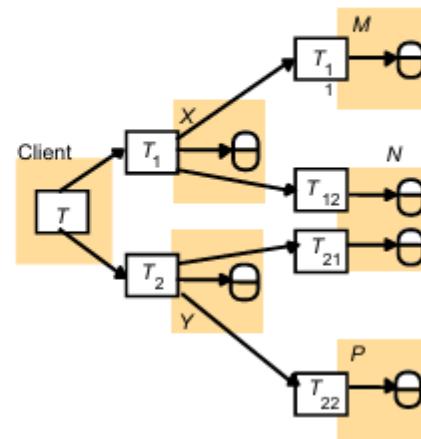
- **Transacción plana o anidada que accede a objetos gestionados por múltiples servidores.**
- Involucran más de un servidor.
- **Cuando una transacción llega a su fin, la propiedad de atomicidad requiere que todos los servidores completen su transacción o que todos aborten.**
- Uno de los servidores asume el rol de **coordinador**, que debe asegurar el mismo resultado en todos los servidores.

Transacciones distribuidas

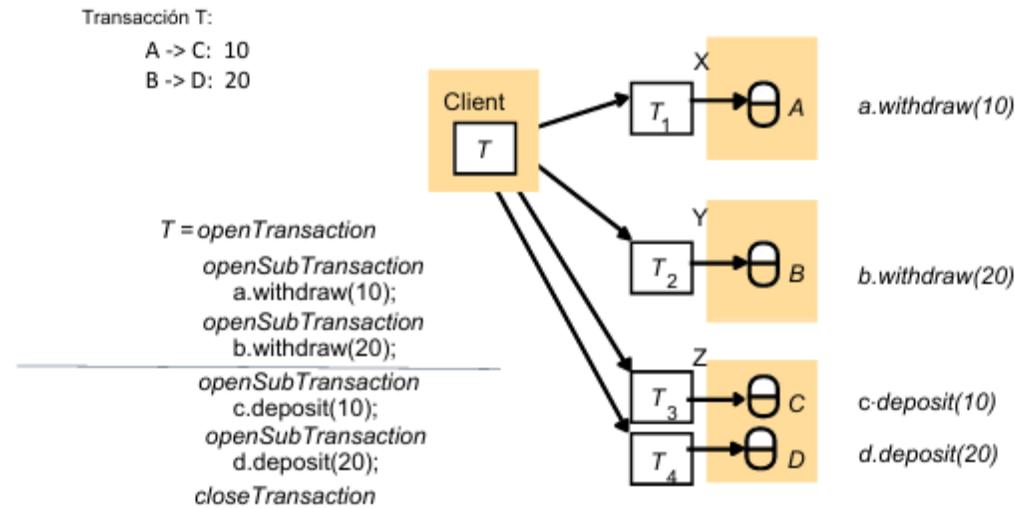
(a) Flat transaction



(b) Nested transactions



Transacción bancaria anidada

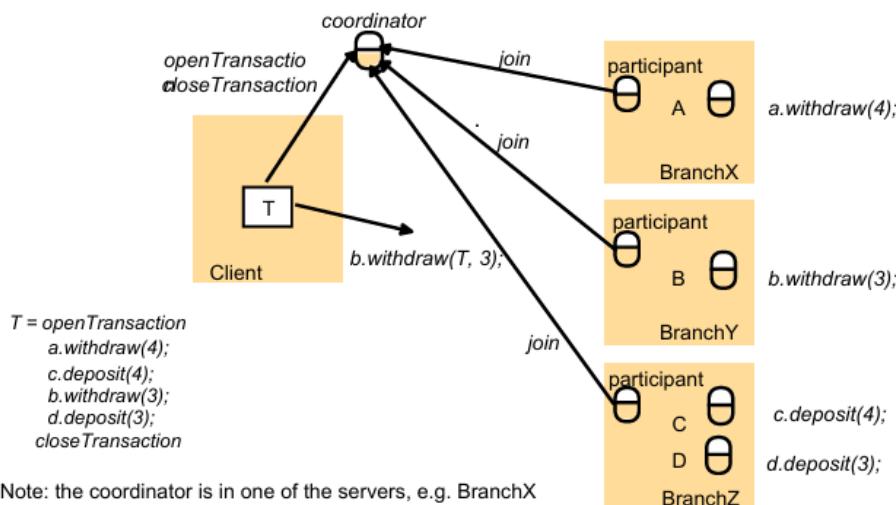


▼ El coordinador de una transacción distribuida

- Los servidores que ejecutan peticiones como parte de una transacción distribuida necesitan poder comunicarse entre ellos para **coordinar sus acciones** cuando se consuma la transacción.
- Un cliente comienza una transacción enviando una petición de abreTransacción al coordinador en cualquier servidor. El coordinador con el que se contacta lleva a cabo abrirTransacción y devuelve al cliente el identificador resultante de la transacción.
- Los identificadores de transacciones para transacciones distribuidas han de ser únicos dentro del sistema distribuido. Una forma sencilla de obtener esto es que cada TID contenga dos partes: el identificador del servidor (por ejemplo, una dirección IP) que la creó y un número único dentro del servidor.
- El coordinador que abrió la transacción se convierte en el coordinador para la transacción distribuida y es el responsable final de consumarla o abortarla.
- **Participante:** *Cada uno de los servidores que gestione un objeto al que accede la transacción* es un participante en la transacción y proporciona un objeto que llamaremos participante. Cada participante es responsable de seguir la pista de todos los objetos recuperables en el servidor implicado en la transacción.

- **Los participantes son responsables de cooperar con el coordinador para sacar adelante el protocolo de consumación.**
- **Referencias:** Durante el progreso de una transacción, el coordinador registra una lista de referencias participantes, y cada participante registra una referencia hacia el coordinador.
- El hecho de que el coordinador conozca a todos los participantes y que cada participante conozca al coordinador les permite recoger la información necesaria al momento de la consumación.

Un ejemplo de transacción bancaria distribuida



▼ Operaciones para el protocolo de consumación en dos fases

puedeConsumar?(trans) → Sí / No

Llamada desde el coordinador al participante para preguntar si puede consumar una transacción. El participante responde con su voto.

Consumo(trans)

Llamada desde el coordinador al participante para decirle que consuma su parte de una transacción.

Aborta(trans)

Llamada desde el coordinador al participante para decirle que aborte su parte de una transacción.

heConsumado(trans, participante)

Llamada desde el participante al coordinador para confirmar que ha consumado la transacción.

dameDecisión(trans) → Sí / No

Llamada desde el participante al coordinador para preguntar por la decisión sobre una transacción tras haber votado Sí aunque no ha obtenido respuesta tras cierto tiempo. Se utiliza para recuperarse de la caída de un servidor o de mensajes con retraso

puede consumar? Para el protocolo de dos fases jerárquico

puedeConsumar?(trans, listaAbortadas) → Sí/No

Llama a un coordinador para que pregunte al coordinador de una subtransacción hija si puede consumar una subtransacción subTrans. El primer argumento trans es el identificador de la transacción del nivel superior. El participante responde con su voto Sí/No.

puede consumar? Para el protocolo de dos fases plano

puedeConsumar?(trans, listaAbortadas) → Sí/No

Llamada desde el coordinador a un participante para preguntar si puede consumar una transacción. El participante responde con su voto Sí/No.

▼ El protocolo de consumación en dos fases

Fase 1 (fase de votación):

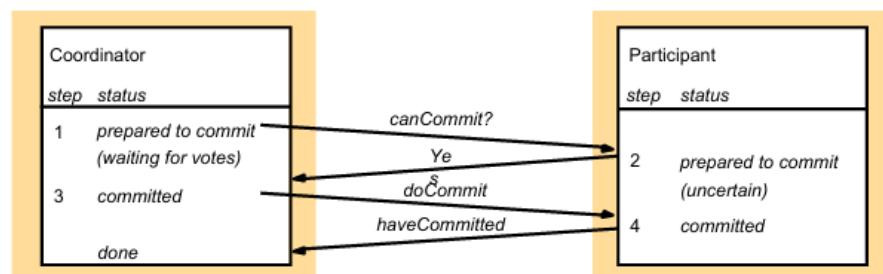
1. El coordinador envía una petición *puedeConsumar?* a cada participante en la transacción.
2. Cuando un participante recibe una petición *puedeConsumar?*, responde al coordinador con su voto (Sí o No). Antes de votar Sí, se prepara para consumar, guardando los objetos en un dispositivo de

almacenamiento permanente. Si el voto es No, el participante aborta inmediatamente.

Fase 2 (finalización en función del resultado de la votación):

3. El coordinador recoge los votos (incluyendo el propio).
 - a. (a) Si no hay fallos y todos los votos son Sí, el coordinador decide consumar la transacción y envía peticiones de *Consuma* a cada uno de los participantes.
 - b. En otro caso, el coordinador decide abortar la transacción y envía peticiones *Aborta* a todos los participantes que votaron Sí.
4. Los participantes que han votado Sí están esperando por una petición *Consuma* o *Aborta* por parte del coordinador. Cuando un participante recibe uno de estos mensajes, actúa en función de ellos, y en el caso de *Consuma*, realiza una llamada de *heConsumado* como confirmación hacia el coordinador.

Comunicación en el protocolo de consumación en dos fases



▼ Operaciones en el coordinador para las transacciones anidadas

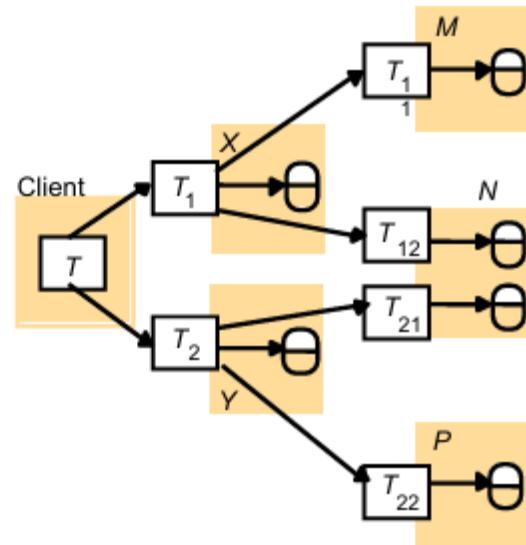
abreSubTransacción(trans) → subTrans

Abre una nueva subtransacción cuya madre es trans, y devuelve un identificador de subtransacción único.

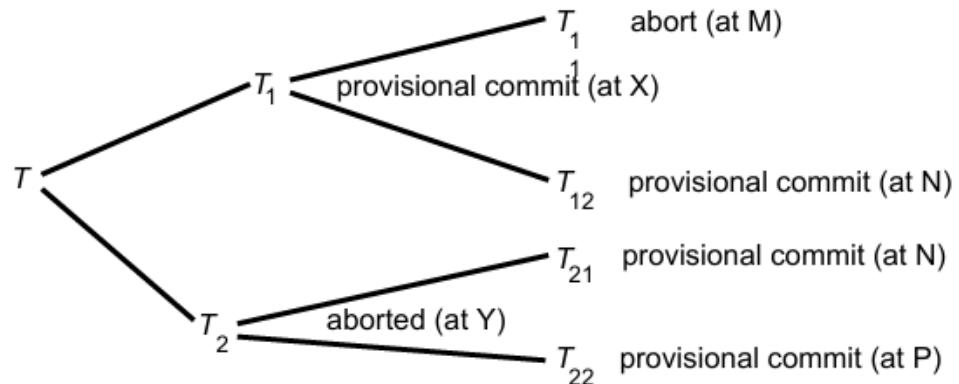
dameEstado(trans) → consumada, abortada, provisional

Pide al coordinador que informe del estado de la transacción trans.
 Devuelve valores que representan uno de los siguientes, consumada,
 abortada, provisional.

La transacción T decide si se consuma



La transacción T decide si se consuma



Información retenida por los coordinadores de transacciones anidadas

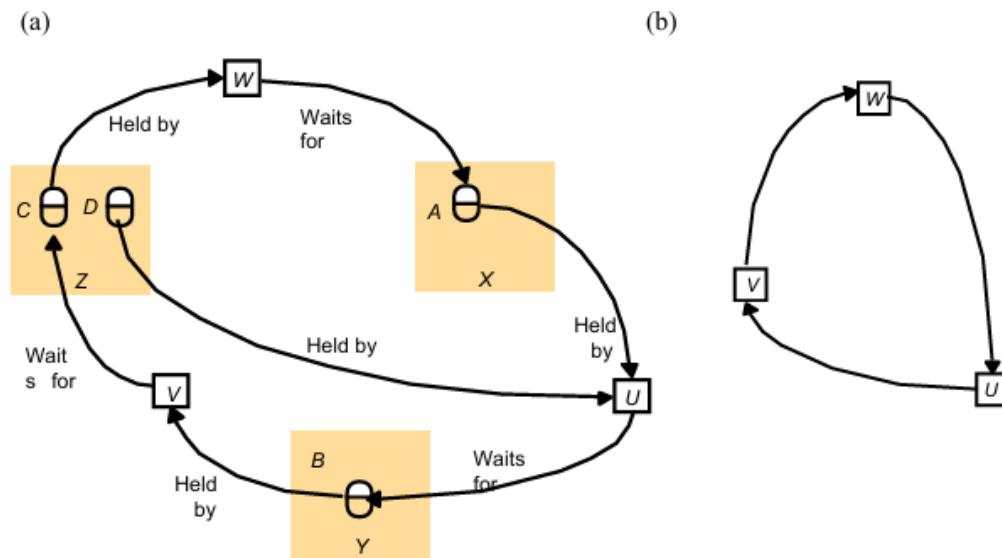
<i>Coordinator of transaction</i>	<i>Child transactions</i>	<i>Participant</i>	<i>Provisional commit list</i>	<i>Abort list</i>
T	T_1, T_2	yes	T_1, T_{12}	T_1, T_2
T_1	T_1, T_{12}	yes	T_1, T_{12}	T_1
T_2	T_{21}, T_{22}	no (aborted)		T_2
T		no (aborted)		T_1
T_{12}, T		T_{12} but not T_{21}	T_{21}, T_{12}	T_1
T_{22}		no (parent aborted)	T_{22}	

▼ Interbloqueos definidos

- Pueden surgir interbloqueos de un único servidor cuando se utilizan bloqueos para el control de concurrencia. Los servidores deben bien evitarlos, o bien detectarlos y resolverlos. La utilización de tiempos límite para posibles interbloqueos es una aproximación tosca, ya que es difícil elegir un intervalo de tiempo límite apropiado y las transacciones serán abortadas de forma innecesaria.
- Con los esquemas de detección de interbloqueos, se aborta una transacción sólo cuando se ha visto involucrada en un interbloqueo. **La mayoría de los esquemas de detección de interbloqueos funcionan detectando ciclos en los grafos espera por de las transacciones.**
- En teoría, en un sistema distribuido en el que están involucrados múltiples servidores a los que están accediendo múltiples transacciones, puede construirse un **grafo espera por global** a partir de los grafos espera por locales.
- La Figura 8.12 muestra el entrelazado de las transacciones U, V y W que involucran a los objetos A y B gestionados por los servidores X e Y, y los objetos C y D gestionados por el servidor Z.

<i>U</i>	<i>V</i>	<i>W</i>
<i>d.deposit(10)</i> lock <i>D</i>		
<i>a.deposit(20)</i> lock <i>A</i> at <i>X</i>	<i>b.deposit(10)</i> lock <i>B</i> at <i>Y</i>	
<i>b.withdraw(30)</i> wait at <i>Y</i>	<i>c.withdraw(20)</i> wait at <i>Z</i>	<i>c.deposit(30)</i> lock <i>C</i> at <i>Z</i>
		<i>a.withdraw(20)</i> wait at <i>X</i>

- En la Figura 8.13(a), el grafo **espera por** completo muestra que un ciclo asociado a un interbloqueo está formado por arcos alternados, que representan a una transacción esperando por un objeto y un objeto retenido por una transacción.
- Ya que cualquier transacción puede estar esperando sólo por un objeto cada vez, los objetos pueden dejarse fuera de los grafos **espera por**, como se muestra en la Figura 8.13(b).

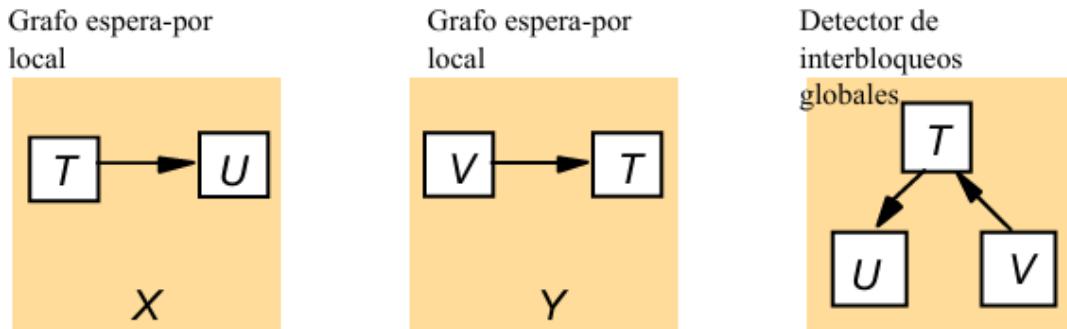


Interbloqueos fantasma:

- Un interbloqueo que se «detecta» pero que realmente no lo es se conoce como **interbloqueo fantasma**. En la detección de interbloqueos distribuidos, la información acerca de las relaciones **espera por** entre las transacciones se transmite de un servidor a otro. Si existe un

interbloqueo, la información necesaria se recogerá, al final, en algún lugar y se detectará el ciclo. **Ya que este procedimiento tarda un cierto tiempo, existe la posibilidad de que una de las transacciones que mantiene un bloqueo se haya liberado, en cuyo caso ya no existe.**

- Considérese el caso de un detector de interbloqueo global, que recibe los grafos *espera por* locales de los servidores X e Y, como se muestra en la Figura 9.14. Supóngase que la transacción U a continuación libera un objeto en el servidor X y solicita el que mantiene V en el servidor Y. Supóngase además que el detector global recibe el grafo local del servidor Y antes que el del servidor X. En este caso, detectaría un ciclo T - U - V - T, aunque el arco T - U ya no existe. Este es un ejemplo de interbloqueo fantasma.



Caza de arcos

- **Caza de arcos.** Una aproximación distribuida para la detección de interbloqueos utiliza una técnica denominada captura de los arcos (edge chasing) o empuje de caminos (path pushing).
- En esta aproximación, el grafo espera por global no se construye, sino que cada uno de los servidores implicados **posee información sobre algunos de sus arcos**.
- Los servidores intentan encontrar ciclos mediante el envío de mensajes denominados **sondas**, que siguen los arcos de un grafo a través del sistema distribuido.
- Un mensaje sonda está formado por las relaciones *espera por* de transacciones que representan un camino en el grafo *espera por* global.

-
- Los algoritmos de captura de arcos tienen **tres pasos: iniciación, detección y resolución.**

Iniciación: cuando un servidor percibe que una transacción T comienza a buscar a otra transacción U, y ésta a su vez está esperando para acceder a un objeto en otro servidor, iniciará la detección enviando una sonda que contiene el arco $\langle T \rightarrow U \rangle$ al servidor del objeto por el cual está bloqueada la transacción U. Si U está compartiendo un bloqueo, se envían sondas a todos los que poseen dicho bloqueo. Algunas veces, otras transacciones pueden comenzar a compartir el bloqueo más tarde, en cuyo caso también deben enviarse sondas a ellas.

Detección: la detección consiste en recibir sondas y decidir si se ha producido un interbloqueo o si hay que enviar nuevas sondas.

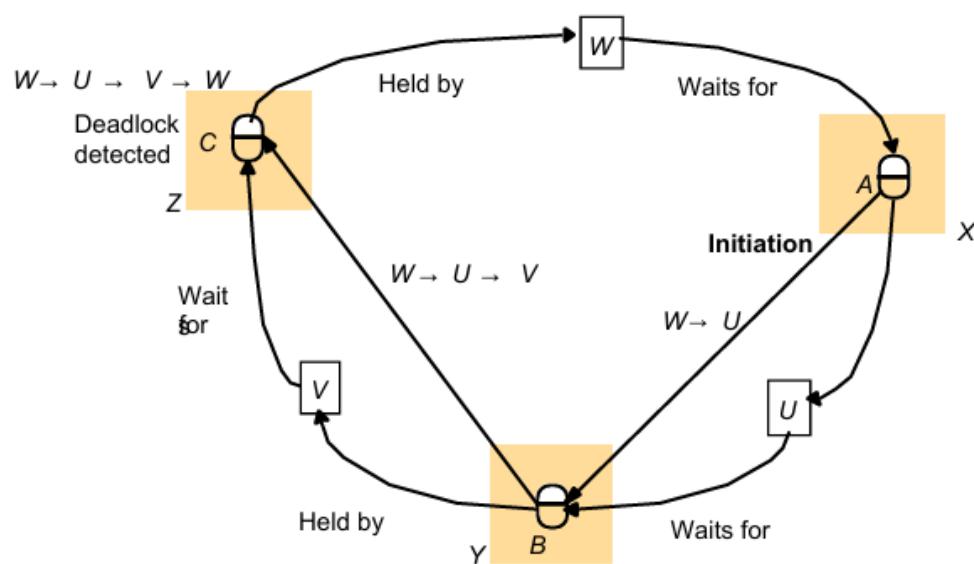
- Por ejemplo, cuando un servidor de un objeto recibe la sonda $\langle T - U \rangle$ (indicando que T está esperando por la transacción U, que retiene un objeto local) comprueba si U está también esperando. Si es así, la transacción por la que espera (por ejemplo, V) se añade a la sonda (consiguiendo $\langle T - U - V \rangle$), y si la nueva transacción (V) está esperando por otro objeto en otro sitio, se vuelve a enviar la sonda.
- De esta forma, se construye un camino a través del grafo espera-por global añadiendo un eje cada vez. Antes de volver a enviar una sonda, el servidor comprueba si la transacción que acaba de ser añadida (por ejemplo, T) ha ocasionado un ciclo en la sonda (por ejemplo $\langle T - U - V - T \rangle$).
- Si es éste el caso, ha encontrado un ciclo en el grafo y se ha detectado un interbloqueo.

Resolución: cuando se detecta un ciclo, se aborta una transacción en el ciclo para romper el interbloqueo.

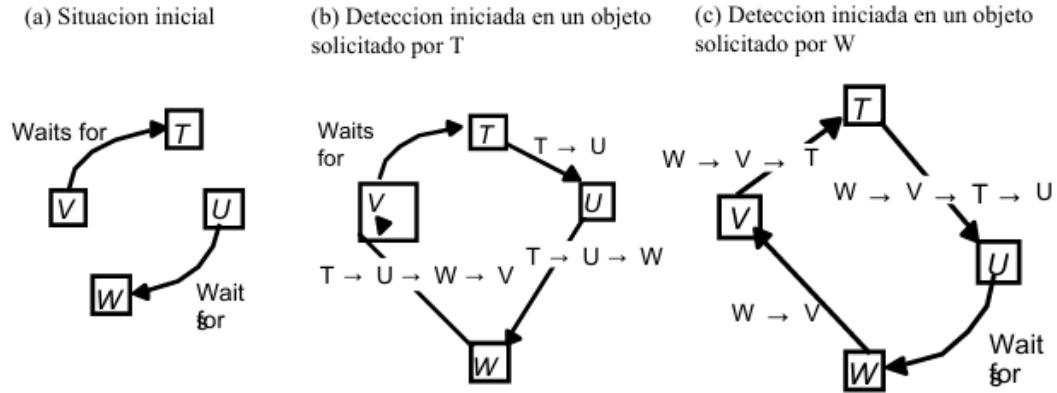
- En nuestro ejemplo, los siguientes pasos describen cómo se inicia la detección de interbloqueos y cómo se envían las sondas durante la correspondiente fase de detección.
 - El servidor X inicia la detección enviando la sonda $\langle W - U \rangle$ al servidor de B (Servidor Y).

- El servidor Y recibe la sonda $\langle W - U \rangle$, observa que B es retenida por V y concatena V a la sonda para producir $\langle W - U - V \rangle$. Observa que V está esperando por C en el servidor Z. La sonda se envía al servidor Z.
 - El servidor Z recibe la sonda $\langle W - U - V \rangle$ y observa que C es retenida por W y concatena W a la sonda para producir $\langle W - U - V - W \rangle$.

• Este camino contiene un ciclo. El servidor detecta un interbloqueo. Para romper el interbloqueo debe abortarse una de las transacciones en el ciclo.

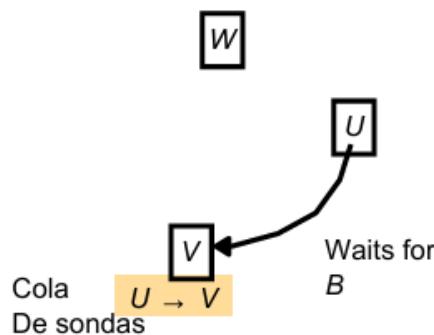


Dos sondas iniciadas

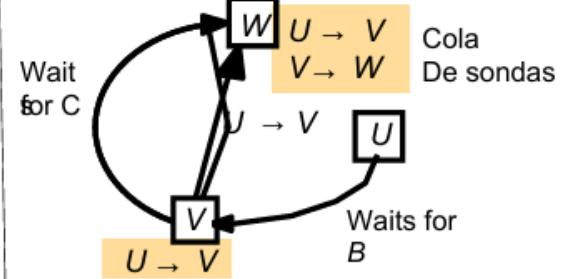


Sondas que viajan cuesta abajo

(a) V almacena la sonda cuando U comienza a esperar



(b) La sonda se reenvia cuando V comienza a esperar



▼ Recuperación de transacciones

- Aunque los servidores de archivos y base de datos mantienen los datos en almacenamiento permanente, otros tipos de servidores no necesitan hacerlo, excepto por motivos de recuperación.
- Se asume que cuando el servidor está funcionando, mantiene todos sus objetos en memoria volátil y registra sus objetos consumados en un archivo o archivos de recuperación.
- Por lo tanto, la recuperación consiste en restaurar el servidor a partir de los dispositivos de almacenamiento permanente y dejarlo con las últimas versiones consumadas de los objetos.

Tipos de entrada en un archivo de recuperación

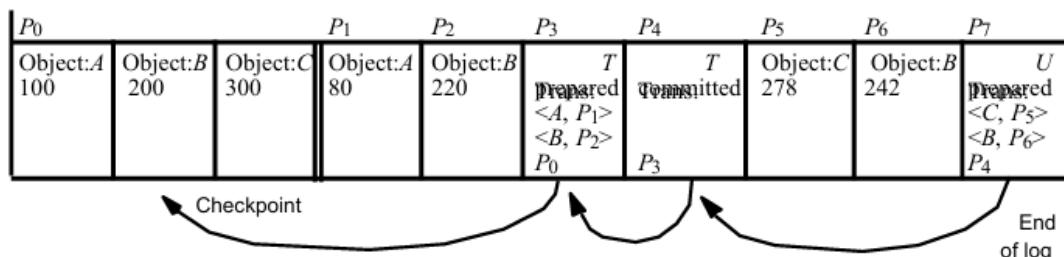
Tipo de entrada	Descripción de los contenidos de la entrada
Objeto	Un valor de un objeto
Estado de la transacción	Identificador de la transacción, estado de la transacción (<i>preparada</i> , <i>consumada</i> , <i>abortada</i>), y otros valores de estado utilizados para el protocolo de consumación en dos fases.
Lista de intenciones	Identificador de la transacción y una secuencia de intenciones, cada una de las cuales consiste en <identificador de objeto>, <posición en el archivo de recuperación del valor del objeto>.

Figura 13.18. Tipos de entradas en un archivo de recuperación.

▼ Registro histórico

- **El registro histórico es una técnica de recuperación de transacciones, en la que el archivo de recuperación representa a un registro que contiene el historial de todas las transacciones realizadas por el servidor.**
- **El historial consta de los valores de los objetos, las entradas del estado de la transacción y las listas de intenciones de las transacciones.**
- El orden de las entradas en el registro refleja el orden en el cual se prepararon, consumieron o abortaron las transacciones en el servidor. En la práctica, el archivo de recuperación contendrá una instantánea reciente de los valores de todos los objetos en el servidor, seguido de un historial de las transacciones ocurridas tras dicha instantánea.

Registro para un servicio bancario



▼ Unidad 9: Aplicaciones

▼ OAuth2

- El marco de trabajo (framework) de autorización OAuth 2.0 **permite que una aplicación de terceros obtenga acceso limitado a un servicio**

HTTP, ya sea en nombre del propietario de un recurso organizando una interacción de aprobación entre el propietario del recurso y el servicio HTTP, **o permitiendo que la aplicación de terceros obtenga acceso en su propio nombre.**

- **Es un protocolo de autorización que permite a terceros (clientes) acceder a contenidos propiedad de un usuario** (alojados en aplicaciones de confianza, servidor de **recursos**) **sin que éstos tengan que manejar ni conocer las credenciales del usuario.** Es decir, aplicaciones de terceros pueden acceder a contenidos propiedad del usuario, pero estas aplicaciones no conocen las credenciales de autenticación.

OAuth2 define 4 roles:

- **Propietario del recurso:** Una entidad capaz de otorgar acceso a un recurso protegido. Cuando el propietario del recurso es una persona, se lo denomina usuario final.
- **Servidor de recursos:** El servidor que aloja los recursos protegidos, capaz de aceptar y responder a solicitudes de recursos protegidos mediante tokens de acceso.
- **Cliente:** Una aplicación que realiza solicitudes de recursos protegidos en nombre del propietario del recurso y con su autorización. El término "cliente" no implica ninguna característica de implementación particular (por ejemplo, si la aplicación se ejecuta en un servidor, un escritorio u otros dispositivos).
- **Servidor de autorización:** El servidor que emite tokens de acceso al cliente después de autenticar con éxito al propietario del recurso y obtener la autorización.

OAuth2 - Flujo

1.2. Protocol Flow

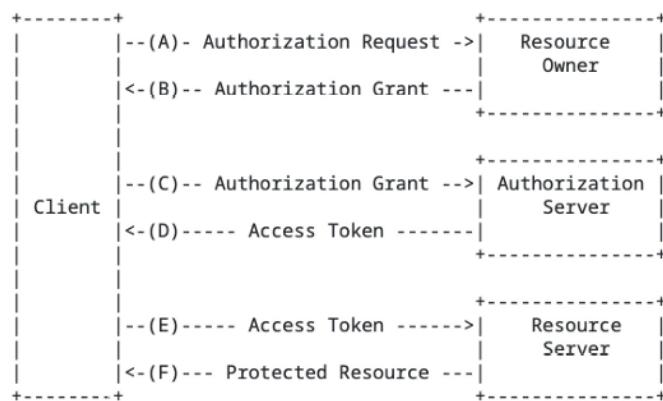
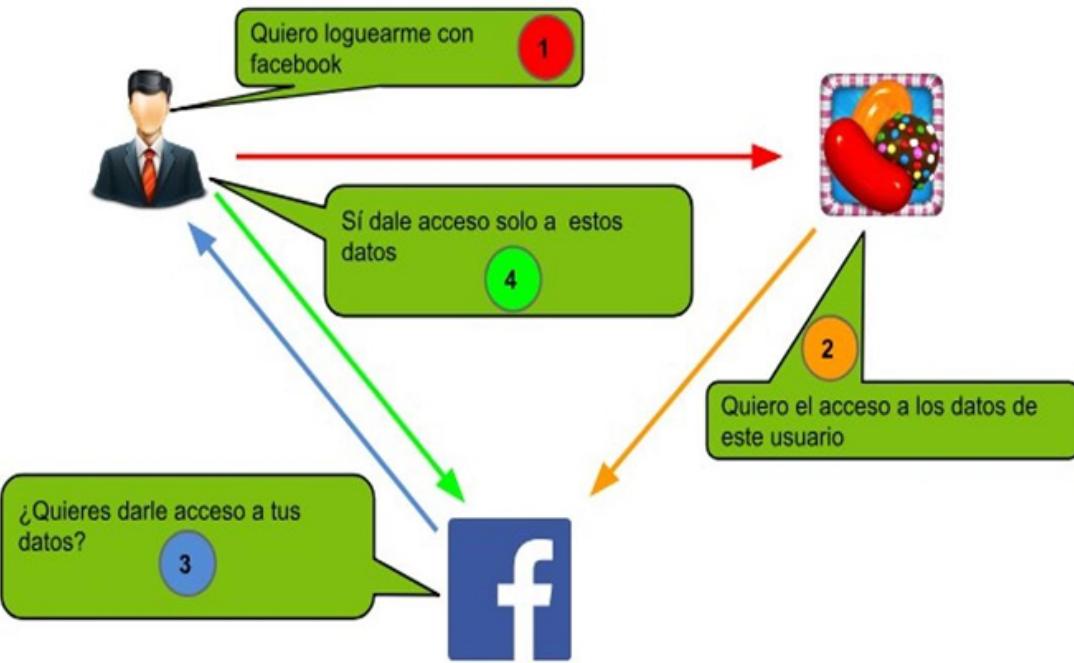


Figure 1: Abstract Protocol Flow

La interacción entre el servidor de autorización y el servidor de recursos está fuera del alcance de esta especificación.

El servidor de autorización puede ser el mismo servidor que el servidor de recursos o una entidad separada.

Un solo servidor de autorización puede emitir tokens de acceso aceptados por varios servidores de recursos.



▼ JSON Web Token (JWT)

- JSON Web Token (JWT) es un medio compacto y seguro para URL de representar reclamaciones que se transferirán entre dos partes. Las reclamaciones en un JWT se codifican como un objeto JSON que se utiliza como carga útil de una estructura de Firma Web JSON (JWS) o como texto sin formato de una estructura de Cifrado Web JSON (JWE), lo que permite que las reclamaciones se firmen digitalmente o se proteja su integridad con un Código de Autenticación de Mensajes (MAC) y/o se cifren.

- Los JSON Web Token son un método abierto, estándar de la industria RFC 7519 para representar reclamaciones de forma segura entre dos partes.

JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens:

- Esta especificación define un perfil para emitir tokens de acceso OAuth 2.0 en formato JSON Web Token (JWT). Los servidores de autorización y los servidores de recursos de diferentes proveedores pueden aprovechar este perfil para emitir y consumir tokens de acceso de manera interoperable.

```
GET /as/authorization.oauth2?response_type=code
&client_id=s6BhdRkqt3
&state=xyz
&scope=openid%20profile%20reademail
&redirect_uri=https%3A%2F%2Fcclient%2Eexample%2Ecom%2Fcbs
&resource=https%3A%2F%2Frss.example.com%2F HTTP/1.1
Host: authorization-server.example.com
```

Figure 1: Authorization Request with Resource and Scope Parameters

Once redeemed, the code obtained from the request above will result in a JWT access token in the form shown below:

Header:

```
{"typ": "at+JWT", "alg": "RS256", "kid": "RjEwOwOA"}
```

Claims:

```
{
  "iss": "https://authorization-server.example.com/",
  "sub": "5ba552d67",
  "aud": "https://rs.example.com/",
  "exp": 1639528912,
  "iat": 1618354090,
  "jti": "dbe39bf3a3ba4238a513f51d6e1691c4",
  "client_id": "s6BhdRkqt3",
  "scope": "openid profile reademail"
}
```

Figure 2: The Header and JWT Claims Set of a JWT Access Token