# TEMA 7

# Introducción a los Sistemas Distribuidos


# Introducción a los Sistemas Distribuidos

1. Introducción
2. Ventajas e inconvenientes de los sistemas distribuidos
3. Hardware de los sistemas distribuidos
4. Consideraciones de diseño de los sistemas distribuidos
5. Comunicación de procesos en sistemas distribuidos
6. Sincronización en sistemas distribuidos

# 1. Introducción

- Para **acelerar la velocidad del hardware** tenemos dos opciones:
  - Aumentar la **tecnología** del procesador (solución muy costosa)
  - Usar **varios procesadores** (múltiples máquinas de gran potencia, interconectadas entre sí, compartiendo una carga de trabajo)
- Para que las máquinas puedan compartir recursos y carga de trabajo necesitamos la utilización de sistemas operativos distribuidos, capaces de gestionar y coordinar las tareas.

# 1. Introducción

- **Sistemas fuertemente acoplados**. Los procesadores comparten la memoria y el reloj. Son sistemas **multiprocesadores** y la comunicación se hace a través de la **memoria compartida**.
- **Sistemas débilmente acoplados**. Los procesadores **no comparten la memoria ni el reloj** y cada uno tiene su propia memoria local. Se comunican mediante redes locales o de área extensa. Son los sistemas **distribuidos**. :
  - **Sistemas operativos de red.** Los usuarios saben de las múltiples máquinas y para acceder a sus recursos necesitan conectarse al computador remoto apropiado. Son sistemas **débilmente acoplados** tanto en hardware como en software.
  - **Sistemas operativos distribuidos**. Los usuarios no necesitan saber que existe una multiplicidad de máquinas y pueden acceder a los recursos remotos de la misma forma que lo hacen a los recursos locales. Son sistemas de **hardware débilmente acoplado**, pero de **software fuertemente acoplados**.

# 1. Introducción

- Vamos a considerar un sistema distribuido ideal como **un conjunto de computadoras independientes que se presenta al usuario como un sistema único**.

  - **Máquinas independientes**: Máquinas que podrían trabajar con absoluta autonomía, sin depender de las demás.
  - **Sistema único**: El usuario no debe ser consciente de que trabaja en un sistema formado por múltiples máquinas.

# 2. Ventajas e inconvenientes

## Ventajas de los sistemas distribuidos

- **Economía**. La relación precio/prestaciones es más adecuada en computadoras pequeñas, que tienen un mantenimiento más económico.
- **Limitaciones tecnológicas**. Un mainframe, con una potencia de P MIPS (millones de instrucciones por segundo) siempre puede ser sustituido por un número suficiente de computadoras más pequeñas de p MIPS que, trabajando conjuntamente, pueden superar la potencia del mainframe. Pero no al revés.
- **Limitaciones físicas**. A veces es necesario que distintas partes de un sistema se ubiquen en zonas geográficas distantes, con lo que la utilización de un único computador queda totalmente descartada.
- **Seguridad**. Ante la avería de un computador, un sistema distribuido puede reorganizarse y seguir funcionando con menos prestaciones. En un sistema centralizado, una avería acaba por completo con el sistema.
- **Escalabilidad**. Un sistema distribuido permite la adición de nuevos equipos al conjunto, lo cual le proporciona mayor potencia.

# 2. Ventajas e inconvenientes

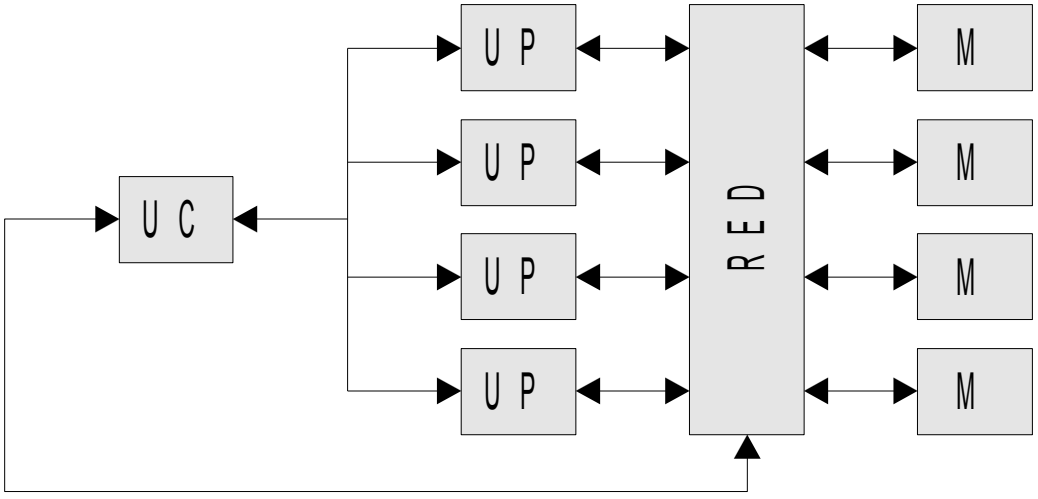
## Desventajas de los sistemas distribuidos

- **Dependencia de la red**. Los sistemas distribuidos dependen completamente de la red. Esto supone dos inconvenientes:
  - **Caída** del sistema en caso de averías en la red
  - **Degradación** del sistema en caso de saturación de la red. Este último inconveniente supone una grave limitación para el crecimiento del sistema. Para evitar la saturación, es necesario reforzar la red de comunicaciones y esto conlleva una fuerte inversión.
- **Falta de seguridad**. El hecho de que los datos se puedan acceder desde posiciones remotas a través de una gran red es un factor de riesgo que deben superar los sistemas distribuidos mediante mecanismos de protección. Pero no siempre son suficientemente seguros.

# 3. Hardware de los SS.DD.

## Clasificación de Flynn (1972)

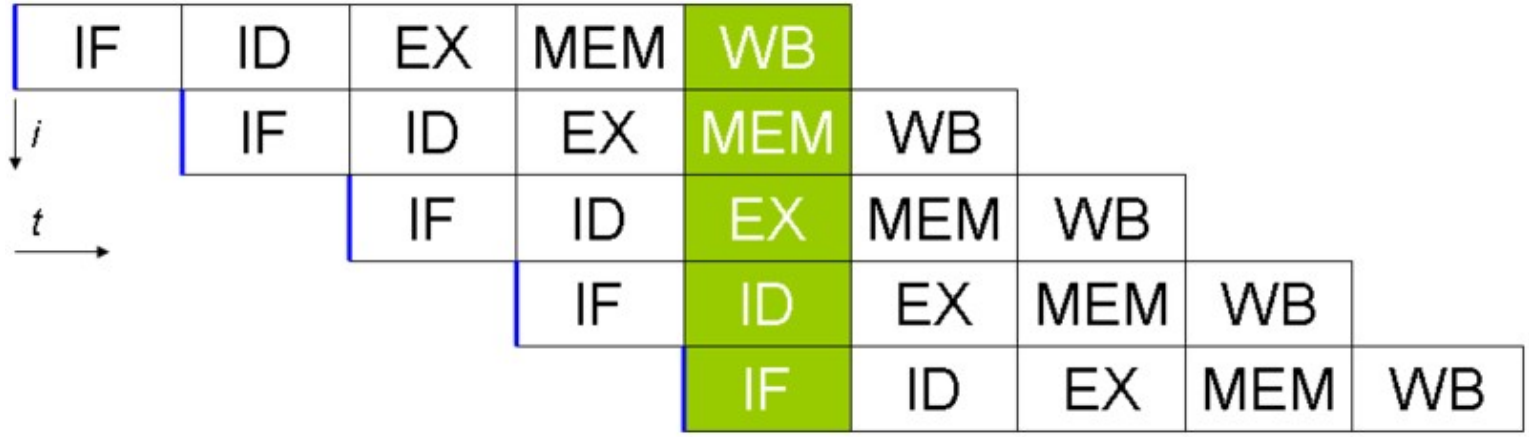
- **SISD** (*single instruction, single data*): ejecutan a la vez una instrucción con un dato (arquitectura *Von Neumann* tradicional).
- Máquinas **SIMD** (*single instruction, multiple data*): tienen múltiples unidades de proceso idénticas, que ejecutan la misma instrucción, pero con datos diferentes (por tanto la unidad de control es única). Son los **procesadores vectoriales**.




Esquema de una máquina SIMD

## Clasificación de Flynn (1972)

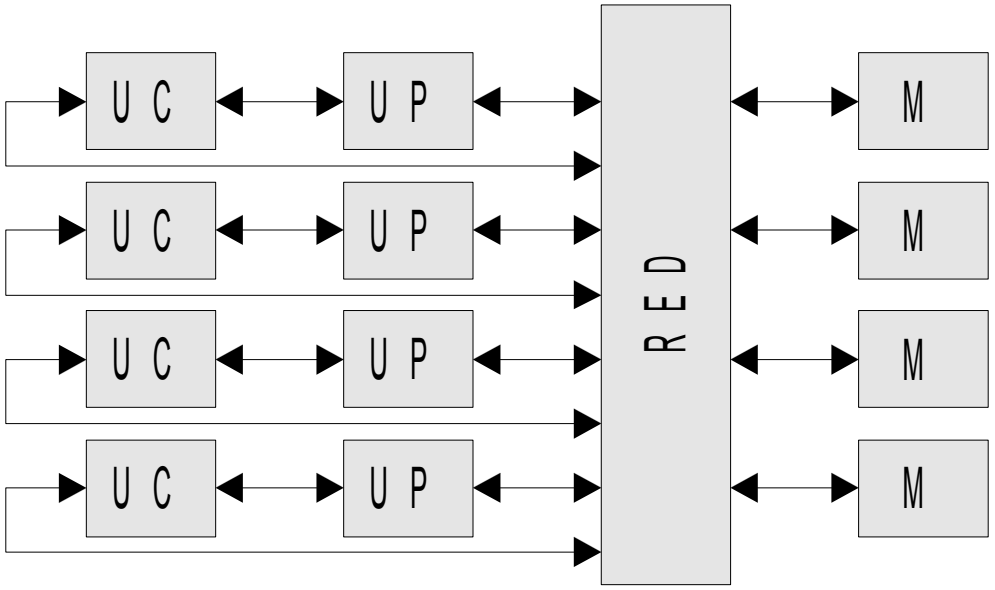
- Máquinas **MISD** (*multiple instruction, single data*): son las máquinas **segmentadas** (*pipe-lines*), en las que diferentes unidades de proceso realizan distintas partes del trabajo necesario para ejecutar un único flujo de instrucciones.



# 3. Hardware de los SS.DD.

## Clasificación de Flynn (1972)

- Máquinas **MIMD** (multiple instruction, multiple data): constan de múltiples unidades de proceso gobernadas cada una por su propia unidad de control, de manera que cada una de ellas puede ejecutar una instrucción diferente con un dato diferente.




Esquema de una máquina MIMD

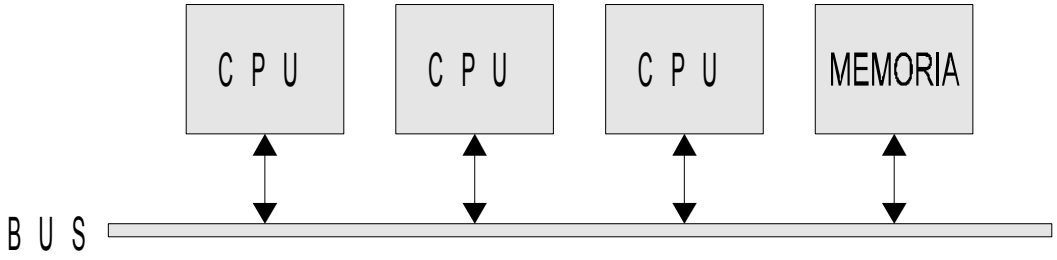
# 3. Hardware de los SS.DD.

- Las máquinas MIMD se pueden subdividir en dos categorías, dependiendo de la estructuración de memoria que utilicen:
  - Los **multiprocesadores** comparten una memoria común, y se dice que sus CPU están **fuertemente acopladas**.
  - Los **multicomputadores** no comparten una memoria común, sino que cada CPU dispone de se propia memoria, y se dice están **débilmente acopladas**.
- A su vez, cada una de estas categorías puede subdividirse en función del tipo de red de comunicación que empleen: un bus o una red de conmutación.

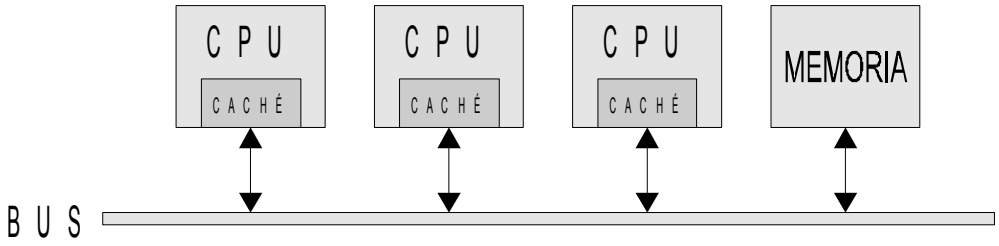
# 3. Hardware de los SS.DD.

## 3.1. Multiprocesadores en bus

- Constituidos por un número determinado de CPUs interconectadas mediante un único bus común, al cual también está conectada la memoria




- Al intentar acceder todas las CPUs a la memoria a través del mismo **bus**, se produce un **cuello de botella**, que hace que el rendimiento del sistema descienda de forma drástica. **Solución:**




Es necesario implementar mecanismos que eviten las incoherencias.

# 3. Hardware de los SS.DD.

## 3.2. Multiprocesadores conmutados

- La arquitectura en **bus** no es buena para más de **64** procesadores. La organización en multiprocesadores conmutados pretende solventar esta limitación. Se basa en dividir la memoria común en varios módulos y unir todos estos módulos con todos los procesadores mediante una **red de conmutadores**.

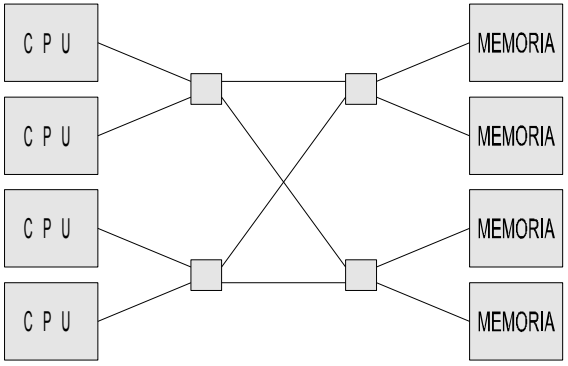
- En el caso más complejo, cada procesador se puede conectar a un módulo de memoria de forma simultánea. Este tipo de red se denomina **crossbar switch**. Si dos CPU necesitan utilizar el mismo módulo de memoria, una de las dos deberá esperar.

# 3. Hardware de los SS.DD.

## 3.2. Multiprocesadores conmutados

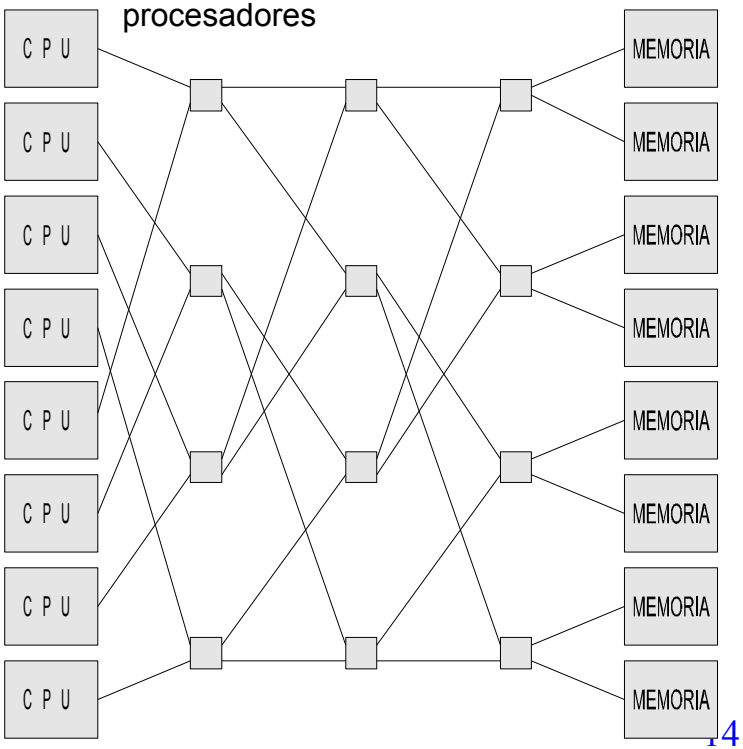
- **Inconveniente**: Si tenemos *n* unidades de proceso y *m* módulos de memoria, necesitamos *n*×*m* conmutadores. **Alto coste**.

- **Solución**: Redes de menores prestaciones. En un momento dado, sólo algunos procesadores pueden acceder a algunas memorias.




Red Omega de 4 procesadores

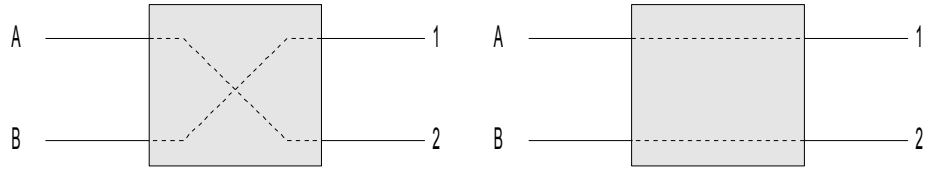
Red Omega de 8 procesadores



# 3. Hardware de los SS.DD.

## 3.2. Multiprocesadores conmutados

- Las redes Omega utilizan unos conmutadores capaces de establecer dos tipos de conexiones. Todas las unidades de proceso pueden acceder a todas las memorias, aunque no al mismo tiempo.




- **Inconvenientes**:

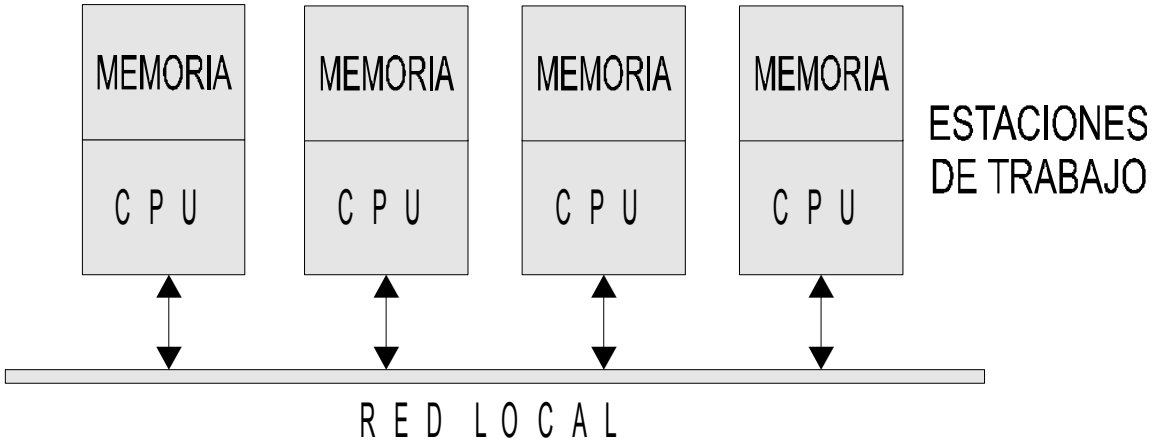
1. Cada conmutador tiene un tiempo de respuesta. Existe un **tiempo de establecimiento**, que es el tiempo que tarda un conmutador en establecer la conexión necesaria y un **tiempo de respuesta**, que es el tiempo que tardan los datos en atravesar un conmutador.
2. Incluso utilizando este tipo de redes, es necesario un gran número de conmutadores (y de muy alta velocidad).

- En consecuencia, este tipo de redes, además de caras, son lentas

# 3. Hardware de los SS.DD.

## 3.3. Multicomputadores en bus

- Al utilizar **multicomputadores**, puesto que cada CPU tiene su propia memoria, **el bus se utiliza únicamente cuando los procesadores necesitan intercambiar información**. Ahora el tráfico por el bus es mucho menor, y no supone un cuello de botella tan evidente como en multiprocesadores en bus.




- Típicamente el **bus es una red Ethernet** y las unidades de proceso son estaciones de trabajo, es decir, que tenemos una **red local**. Hemos pasado de un bus de una placa madre con una velocidad de 300 MBPS a una red Ethernet de 10 MBPS.

# 3. Hardware de los SS.DD.

## 3.4. Multicomputadores conmutados

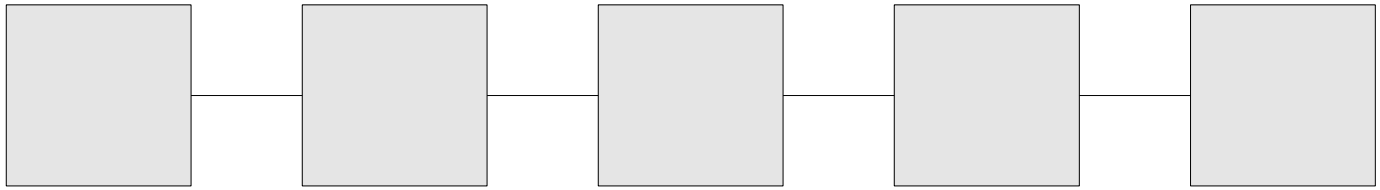
- Multitud de nodos independientes, formados por procesador más memoria, e interconectados mediante alguna red de estructura regular.
- En este tipo de redes **es interesante estudiar la conectividad.** Se utilizan estructuras regulares, lo cual permite conocer de forma determinista el **retardo de los mensajes** que se pasan entre los nodos. Éste no será otro que el **número de enlaces** que deba atravesar para llegar al nodo deseado. El máximo retardo corresponde al **diámetro de la red**.

# 3. Hardware de los SS.DD.

## 3.4. Multicomputadores conmutados

### Topología en línea

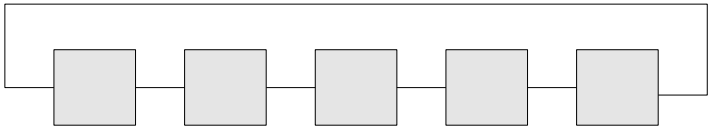
Cada nodo se conecta con otros dos excepto dos de ellos, situados en los extremos, que sólo se conectan con un nodo. El **diámetro** de red de *n* nodos es ***n-1***. Ejemplo con 5 nodos.



## 3.4. Multicomputadores conmutados

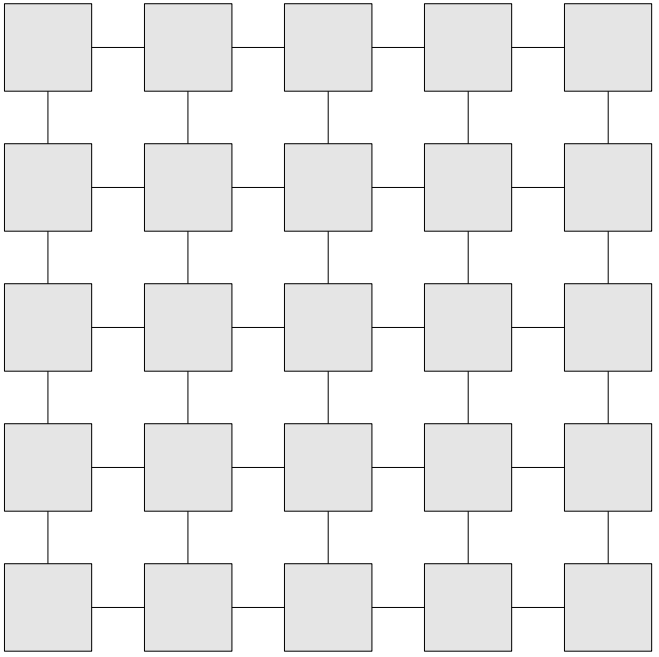
### Topología en anillo

Conectando los nodos de los extremos. Ahora el **diámetro** para una red de *n* nodos es de $\lfloor n/2 \rfloor$ . Ejemplo con 5 nodos.



### Malla bidimensional

Cada nodo se conecta con los *n* nodos contiguos, excepto los de los extremos. El **diámetro** de una malla bidimensional de *n*×*n* nodos es de ***2(n-1)***. Ejemplo con 5x5 nodos.

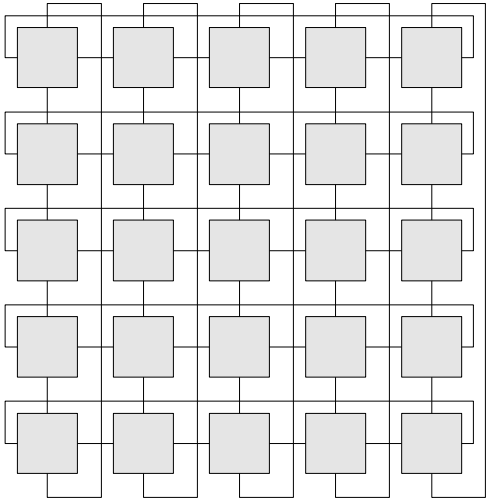


# 3. Hardware de los SS.DD.

## 3.4. Multicomputadores conmutados

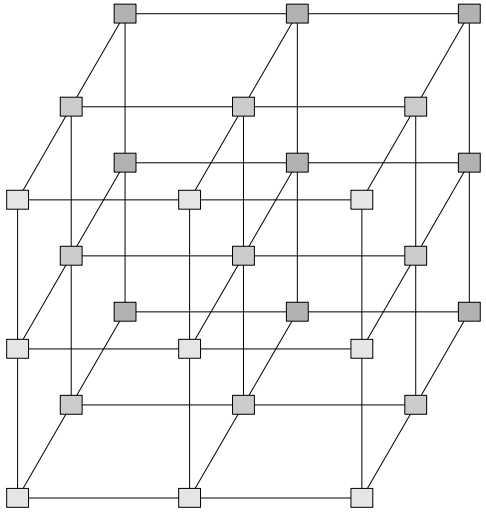
### Topología en Toro

Todos los nodos se conectan con otros *n* nodos (contiguos), incluidos los de los extremos. El **diámetro** de esta topología es de $2\times \lfloor n/2 \rfloor$ para una red de *n*×*n* nodos. Ejemplo con 5x5 nodos.



### Estructuras de malla n-dimensionales

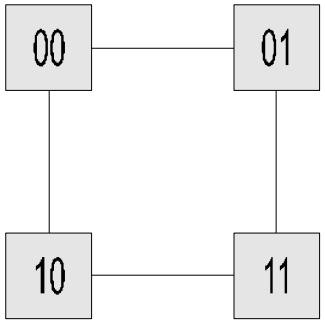
Cada nodo se conecta con *2×n* nodos contiguos, siendo *n* la dimensión, siendo el **diámetro** de *n(m-1)* para redes de *m* elementos en cada dimensión. Ejemplo de malla tridimensional



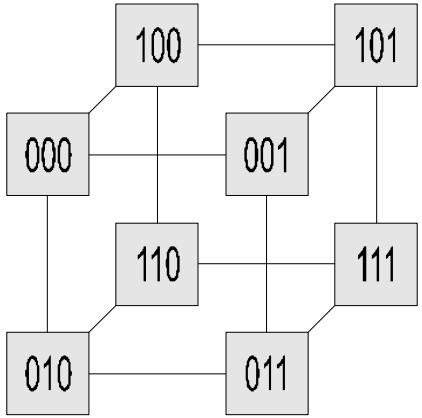
## 3.4. Multicomputadores conmutados

### Hipercubos

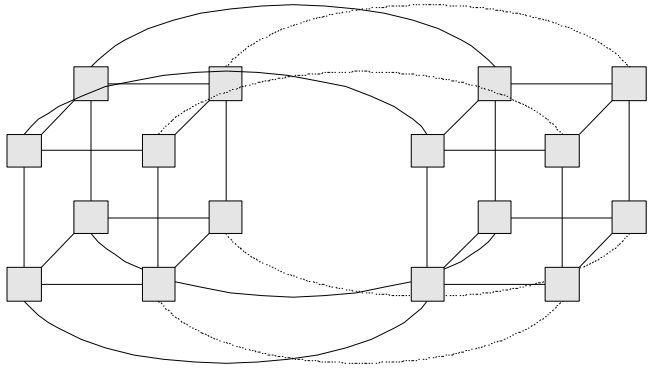
Arquitectura parecida a la de malla n-dimensional, pero con una conectividad menor, pero más regular y estructurada. En un hipercubo de orden *n* existen $2^n$ nodos, con *n* conexiones cada uno, y el **diámetro** es de *n*. Si cada nodo se identifica con un número entre *0* y $2^n$*-1*, y representamos esos números en binario, cada nodo se conecta con aquellos otros cuyo identificador se diferencia únicamente en un bit.



Hipercubo de orden 2



Hipercubo de orden 3


Hipercubo de orden 4

# 4. Consideraciones de Diseño de los SS.DD

- **Transparencia**. El SSDD debe como si se tratara de un sistema monoprocesador tradicional. Se divide en:
  - **Transparencia a la ubicación**. Los nombres de los recursos no deben guardar relación con su ubicación.
  - **Transparencia a la replicación**. Se proporciona replicación de servicios, de manera que servicios repetidos aparecen como uno solo, y en caso de fallo de una de las copias, la réplica entra en funcionamiento de forma transparente.
  - **Transparencia a la concurrencia**. Se requiere un control sobre el acceso a los recursos, de manera que el aspecto exterior sea el de que no hay accesos simultáneos.
  - **Transparencia al paralelismo**. Los problemas deben ser descompuestos en múltiples tareas que se puedan ejecutar de forma paralela. Así, el programador debe ser consciente de cuántos procesadores dispone para resolver su problema y de las técnicas de descomposición del problema.

# 4. Consideraciones de Diseño de los SS.DD

- **Flexibilidad**
  - Existen dos escuelas sobre la forma en que deben escribirse los sistemas operativos.
    - Por un lado están los defensores del **kernel monolítico**, en el que el núcleo del sistema operativo contiene todos los servicios del mismo.
    - Por otro lado, los defensores del **microkernel** abogan por un núcleo en el que sólo se incluyen los servicios más fundamentales, quedando el resto como tareas de usuario.
  - Las **ventajas de los sistemas monolíticos es su mayor rapidez**, ya que al estar todos los servicios en el núcleo, su ejecución es inmediata, sin necesidad de paso de mensajes entre tareas. **La principal ventaja de los sistemas microkernel es su flexibilidad**, ya que la configuración de las tareas es mucho más sencilla que la configuración del núcleo. Es más fácil, por ejemplo, cambiar un manejador de dispositivo por otro más moderno.

# 4. Consideraciones de Diseño de los SS.DD

- **Fiabilidad**: Está relacionada con dos conceptos distintos: **disponibilidad y seguridad**.
  - En el sentido de la **disponibilidad**, la fiabilidad se entiende mediante la **replicación** de servicios en máquinas diferentes, de manera que en caso de fallo de una máquina, el servicio que presta no deja de estar disponible ya que también se presta en otra de las máquinas del sistema.
  - En el sentido de **seguridad**, los sistemas distribuidos generan un alto grado de comunicación de datos sensibles por las redes, lo cual representa un grave problema de seguridad, ya que facilita las **escuchas y suplantaciones**. Los sistemas distribuidos deben ser especialmente cuidadosos con este problema.

# 4. Consideraciones de Diseño de los SS.DD

- **Prestaciones**
  - La división de los problemas en tareas paralelas y concurrentes puede mejorar el rendimiento, pero un uso indebido puede hacer todo lo contrario, debido a que **el tráfico por la red** necesario para el protocolo de comunicación entre las tareas supone una pérdida considerable de tiempo.
  - Hay que tener en cuenta el concepto de **granularidad** en el paralelismo. La **granularidad fina** tiene el inconveniente de que es posible que el **coste** de dividir el problema y enviar los fragmentos a las distintas máquinas puede ser mayor que el beneficio logrado por la ejecución paralela. Sin embargo, una **granularidad excesivamente gruesa** tampoco es adecuada. Llegar a un grado de granularidad adecuado es un problema complejo que debe ser estudiado **en función de cada problema** considerando las necesidades particulares de comunicación y cálculo.

# 4. Consideraciones de Diseño de los SS.DD

- **Escalabilidad**. Los algoritmos que valen para un sistema distribuido de un cierto número de máquinas puede no ser válido para un número mucho mayor.
- Por el momento podemos afirmar que las siguientes consideraciones no deben faltar si queremos un buen sistema distribuido:
  - Ninguna máquina debe tener la información completa del sistema (la información está distribuida).
  - Las máquinas toman decisiones basadas en su información local (no hay un nodo que centralice y redistribuya la información).
  - El fallo de una máquina no interrumpe el funcionamiento global.
  - No es necesario un reloj común (porque la sincronización de relojes en entornos de redes de área extensa es complicado y costoso).

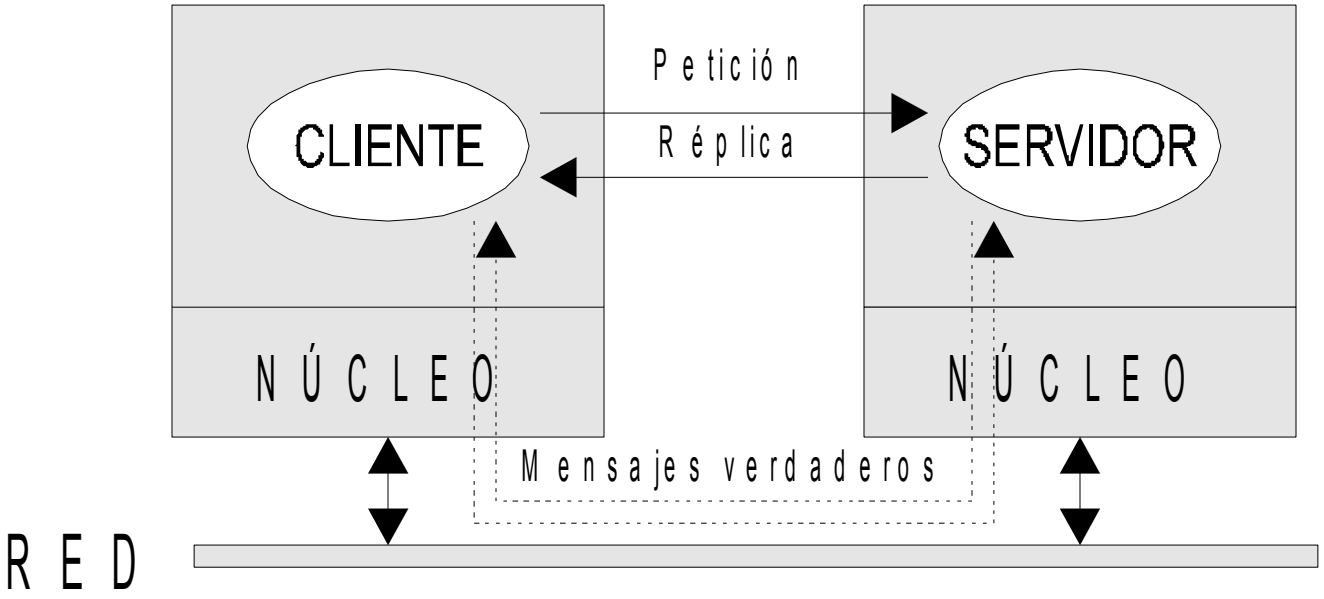
# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.1. Modelo Cliente-Servidor

- El modelo cliente servidor es un protocolo de comunicación no orientado a conexión, en el que existen dos tipos de entidades:
  - Servidores, que son procesos que prestan servicios
  - Clientes, que son procesos que utilizan estos servicios.
- Los clientes envían un mensaje al servidor solicitando un servicio, y los servidores responden con otro mensaje que proporciona el servicio o un error.
- El núcleo de cada máquina es quien se encarga realmente de la distribución de estos mensajes

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.1. Modelo Cliente-Servidor




El paso de mensajes se realiza mediante dos primitivas, una para enviar mensajes (que denominaremos send) y otra para recibir (receive).

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.1. Modelo Cliente-Servidor

- **Problema**: Identificación de los procesos clientes y servidores. Necesitan una **identificación** global dentro del sistema.
- Una primera solución consiste en dividir la identificación en dos partes.
  - Por un lado tenemos la identificación de la **máquina** en la que reside el proceso, que está siempre identificada dentro del sistema a través de una **dirección de red**.
  - Por otro lado, asignamos un identificador (que llamamos **puerto**) al **servicio** proporcionado por el proceso dentro de cada máquina.
- El problema de este método es que no es transparente, ya que el usuario necesita conocer la máquina en la que se presta el servicio.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.1. Modelo Cliente-Servidor

- **Problema**: **Identificación** de los procesos clientes y servidores. Necesitan una **identificación global dentro del sistema.**
- Una solución es unificar la dirección de red y el puerto, y utilizar un **único identificador para cada servicio** prestado en el sistema.
- El problema es que el sistema debe averiguar **en qué máquina** se presta cada servicio a través del identificador.
- Una posible solución a este problema es **difundir el mensaje** con el identificador y que el servidor correspondiente responda indicando su dirección física, con el inconveniente de que se añade una carga adicional al sistema.
- Otra posible solución acceder a los servicios por un **nombre**. En este caso necesitamos un servidor de nombres que proporcione la ubicación física de los servicios, que tiene el inconveniente de que es un servicio centralizado que no escala correctamente.

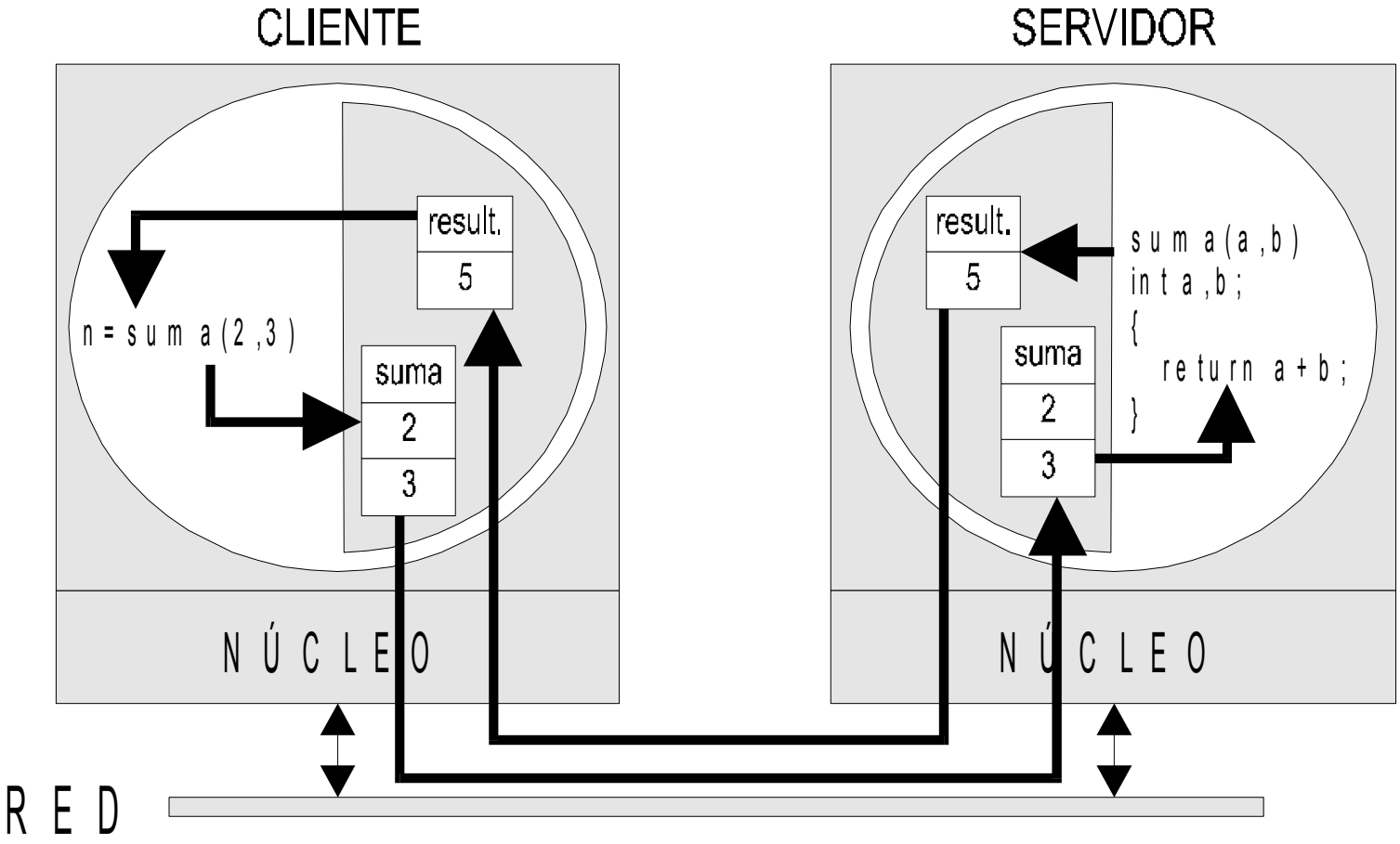
# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.2. Llamada a procedimiento remoto (RPC)

- Es un mecanismo que engloba el modelo cliente servidor y la programación convencional basada en las llamadas a procedimientos y funciones.
- El cliente envía sus peticiones al servidor mediante una llamada ordinaria a un procedimiento, pero ese procedimiento se ejecuta en otro proceso (y posiblemente en otra máquina).
- Internamente, y de forma transparente, se produce un paso de mensajes entre el cliente y el servidor, para comunicarse el servicio solicitado y los parámetros pasados al procedimiento, así como el resultado del servicio, que es devuelto como resultado de la función.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.2. Llamada a procedimiento remoto (RPC)



# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.2. Llamada a procedimiento remoto (RPC)

- Tanto en el servidor como en el cliente existen dos porciones de código que se encargan de hacer transparente la construcción e interpretación del mensaje subyacente.
- Este código habitualmente se genera de forma automática mediante **compiladores especiales**, y se denominan **cabos** (stubs).
- El programador proporciona el *interface* de los procedimientos del cliente y las funciones asociadas en el servidor y el compilador genera el código que compone, envía, recibe e interpreta los mensajes.
- Sólo queda programar las rutinas de servicio (el servidor), el programa cliente, y compilarlos junto a los cabos.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.2. Llamada a procedimiento remoto (RPC)

### El Problema del paso de parámetros

- Plataformas hardware de distintas características pueden utilizar distintas **representaciones de la información**. Por ejemplo en el número de bits para representar los números.
- Tenemos entonces dos posibilidades:
  - Convertir los parámetros a un **formato unificado** ("canónico").
  - Añadir al mensaje información sobre el hardware de la máquina de la que procede el mensaje, y realizar las **conversiones** oportunas en el servidor.
- Otro problema es el paso de **punteros y parámetros por variable**. Puesto que la memoria no es compartida por el servidor y el cliente
- Otras estructuras más complejas, como listas enlazadas o árboles, no pueden ser utilizadas en RPC.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.2. Llamada a procedimiento remoto (RPC)

### Enlace dinámico

- Ya hemos visto que no es conveniente que un servicio esté ligado a una máquina concreta. Los cabos de RPC deben tener esto en cuenta.
- En Sun RPC (implementación de RPC diseñada por Sun Microsystems), existe lo que se denomina enlace dinámico.
- Al especificar el interfaz de los cabos, debe suministrarse un **nombre de servidor y una versión**. Posteriormente, al utilizar los servicios RPC, los cabos tratarán de encontrar la máquina servidora a través del nombre y la versión , y no por una dirección concreta.
- Para que esto sea posible, es necesario que exista un servicio de enlace dinámico, es decir, un servidor (**servidor de nombres**) que proporcione las direcciones físicas de los procesos que prestan el servicio.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.2. Llamada a procedimiento remoto (RPC)

### Enlace dinámico

- El servidor de nombres mantiene información sobre los servidores RPC disponibles.
- Para ello, cuando un servidor comienza a ejecutarse debe registrarse en el servidor de nombres, proporcionándole el nombre y versión.
- Hay varios detalles importantes:
  - Varios programadores pueden darle el mismo **nombre** a sus servidores, incluso la misma versión. Por eso, además de nombre y versión los servidores se identifican con un número aleatorio de 32 bits.
  - Las distintas **versiones** de un servidor pueden proporcionar nuevos servicios, eliminar otros y modificar los existentes. Por tanto, una versión distinta de un servidor es realmente un servidor distinto.
  - Debemos **evitar la centralización del servicio de nombres**, por lo que puede ser conveniente replicarlo o distribuirlo, con el consiguiente problema de coherencia y actualización.

## 5.2. Llamada a procedimiento remoto (RPC)

### Control de Errores

- Un procedimiento remoto puede presentar cualquiera de estos **problemas**:
  - El cliente no puede localizar al servidor.
  - El mensaje del cliente se pierde en la red.
  - La respuesta del servidor se pierde en la red.
  - El servidor cae mientras procesa el mensaje.
  - El cliente cae antes de recibir la respuesta.
- El problema es cómo se entera el cliente de que la llamada ha fracasado.
- En este tipo de situaciones, los programas suelen **repetir las peticiones** o respuestas, mediante un temporizador. Pero no siempre es posible actuar así.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.2. Llamada a procedimiento remoto (RPC)
## Detalles de Implementación

- La implementación de RPC debe ser **eficiente y eficaz**. Si no lo es, no sirve. Se debe prestar atención a los siguientes aspectos:
  - El **protocolo** sobre el que se apoya RPC proporcionará mayor o menor velocidad, a costa de la seguridad. Si RPC se basa en **TCP**, el servicio será fiable pero lento, mientras que si se basa en **UDP** será más rápido y menos seguro, y por lo tanto será necesario implementar algún mecanismo de reconocimientos.
  - Los **reconocimientos** pueden sobrecargar la red si no se planifican correctamente. Podemos reconocer cada paquete ("parada y espera"), o reconocer varios paquetes con una única confirmación ("por ráfagas").
  - Por último hay que tener en cuenta que la **copia de parámetros** se produce varias veces entre distintos espacios de direccionamiento. Como mínimo hay que copiar entre el espacio del usuario y el espacio de los cabos, y desde ahí hacia el buffer del adaptador. Se trata de minimizar el número de copias para evitar cuellos de botella.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.3. Modelo de objetos distribuidos

- **Objetivo**: Extender el paradigma de orientación a objetos al desarrollo de sistemas distribuidos
- **Finalidad**: Diseñar e implementar sistemas distribuidos que se estructuren como colecciones de componentes modulares (objetos) que puedan ser manejados fácilmente y organizados en capas para ocultar la complejidad del diseño.
  - Posibilidad de invocar y pasar como argumento diferentes ”comportamientos”.
  - Posibilidad de aplicar patrones de diseño orientados a objetos en la implementación del sistema distribuido.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.3. Modelo de objetos distribuidos

- El mecanismo de invocación de objetos es denominado **Invocación de métodos remotos** (Remote Method Invocation o RMI)
- Ejemplos de sistemas basados en objetos distribuidos
  - CORBA (Common Object Request Broquer Architecture)
  - Java RMI
  - Microsoft .Net Remoting

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.3. Modelo de objetos distribuidos

### Interfaces

- En general, la interfaz de un módulo de programa especifica los procedimientos y las variables del módulo que son accesibles desde otros módulos
- En programas distribuidos
  - Los módulos pueden ejecutarse en procesos que están en distintas máquinas. Como consecuencia, un módulo no puede acceder a las variables de un módulo remoto.
  - Por este motivo, las interfaces usadas tanto en RPC como RMI no permiten el acceso a directo a las variables del módulo.
  - Tampoco se permiten punteros en los argumentos y valores de retorno de las operaciones.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.3. Modelo de objetos distribuidos

### Interfaces

- En el modelo cliente-servidor con RPC:
  - Las interfaces especifican los servicios que ofrece un servidor.
  - Suele existir una relación uno a uno entre el servidor y la interfaz.
- En el modelo de objetos distribuidos:
  - Las interfaces especifican las operaciones que se pueden invocar de un objeto.
  - Un objeto es una entidad que implementa una interfaz.
  - No tiene por qué existir una relación uno a uno entre objetos y servidores.

## 5.3. Modelo de objetos distribuidos

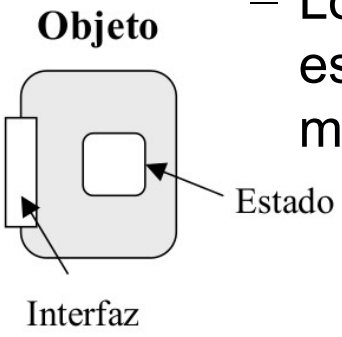
### Arquitectura

- Los sistemas basados en objetos distribuidos presentan una arquitectura similar, aunque algunos elementos son denominados de forma diferente dependiendo del sistema.
- El objetivo fundamental que se persigue es:
  - Llevar a cabo la invocación de un método de un objeto distribuido.

## 5.3. Modelo de objetos distribuidos

### Arquitectura

- Cuando un objeto no es distribuido
  - A partir de una referencia del objeto, el programa cliente invoca los métodos del objeto usando un mecanismo de invocación de procedimientos
  - La referencia al objeto
    - Permite acceder al código de los métodos
    - Los métodos acceden al estado del objeto, que suele estar compuesto por una estructura almacenada en el montículo del espacio de direcciones

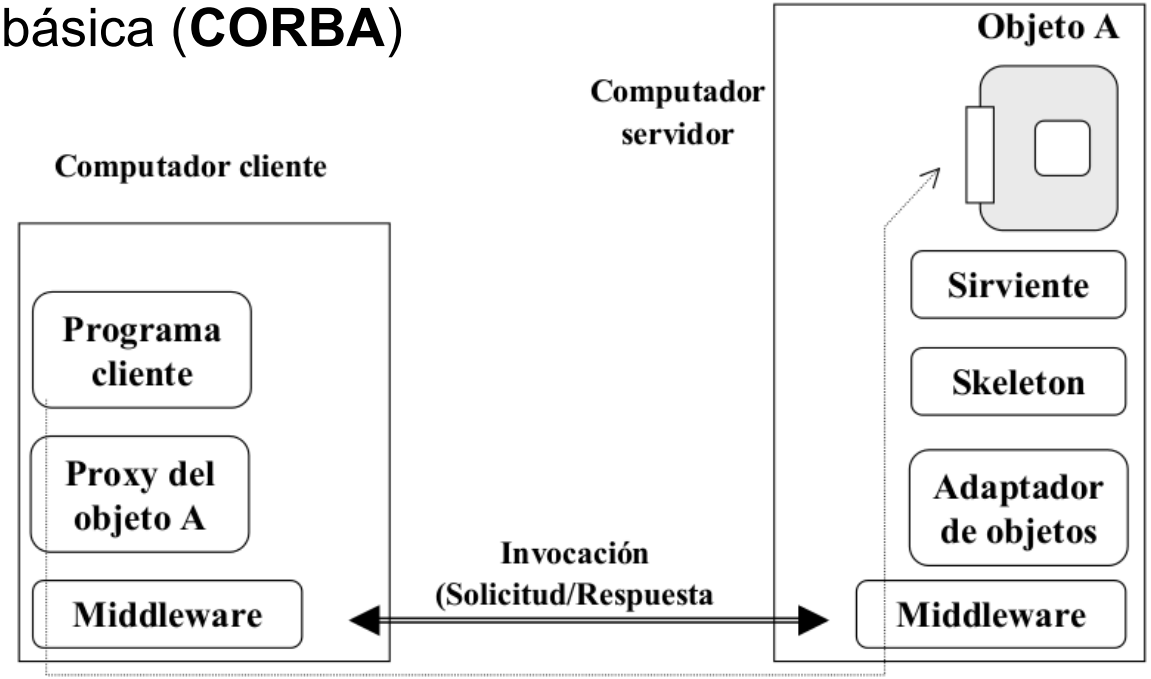


# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.3. Modelo de objetos distribuidos

### Arquitectura

- Cuando un objeto es distribuido
  - La referencia al objeto debe permitir acceder al objeto distribuido.
- Arquitectura básica (**CORBA**)

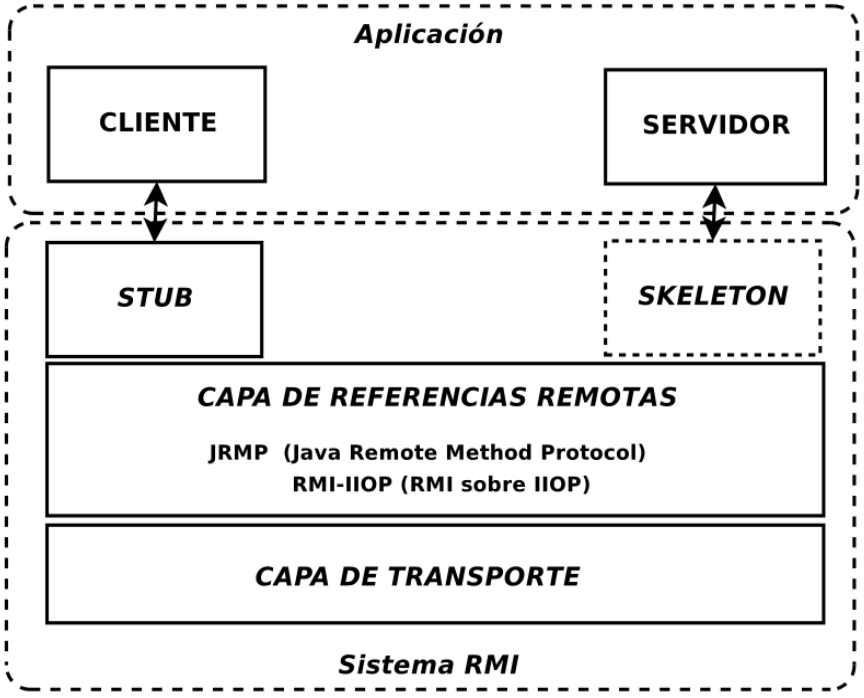


# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.3. Modelo de objetos distribuidos

### Arquitectura

- Cuando un objeto es distribuido
  - La referencia al objeto debe permitir acceder al objeto distribuido.
- Arquitectura básica (**JAVA RMI**)



# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.3. Modelo de objetos distribuidos

### Componentes

- **Proxy**: el proxy de un objeto hace las funciones de un **stub** de un cliente en RPC.
- Es accedido por la referencia que tiene el cliente del objeto.
- Su misión es
  - Empaquetar los argumentos de la operación invocada por el cliente.
  - Invocar al *middleware* para transmitir la operación al objeto.
  - Desempaquetar los resultados de la operación y devolvérselos al cliente.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.3. Modelo de objetos distribuidos

### Componentes

- **Skeleton**: Hace las mismas funciones que el **stub** o **skeleton** de un servidor en RPC. Su misión es
  - Desempaquetar los argumentos de la operación invocada por el cliente.
  - Contactar con el sirviente del objeto.
  - Empaquetar los resultados de la operación y devolvérselos al cliente.
- **Sirviente** (servant): Es una entidad que implementa uno o varios objetos.
  - Existe en el contexto de un servidor.

## 5.3. Modelo de objetos distribuidos

### Componentes

- **Adaptador de objetos** (CORBA). Constituye el medio de enlazar los *sirvientes* con el *middleware*. Es una entidad que adapta la interfaz de un objeto a una interfaz diferente, que es la que espera el cliente.
- Hay que tener en cuenta que el cliente y el objeto pueden estar escritos en lenguajes diferentes.
- Mediante herencia o delegación permite que un cliente invoque servicios de un objeto sin conocer cuál es la interfaz real del objeto.
- Permite establece políticas de activación
  - Un sirviente representa todos los objetos.
  - Se crea un sirviente por objeto, bajo demanda.
  - Vinculación estática entre objeto y sirviente.

## 5.3. Modelo de objetos distribuidos

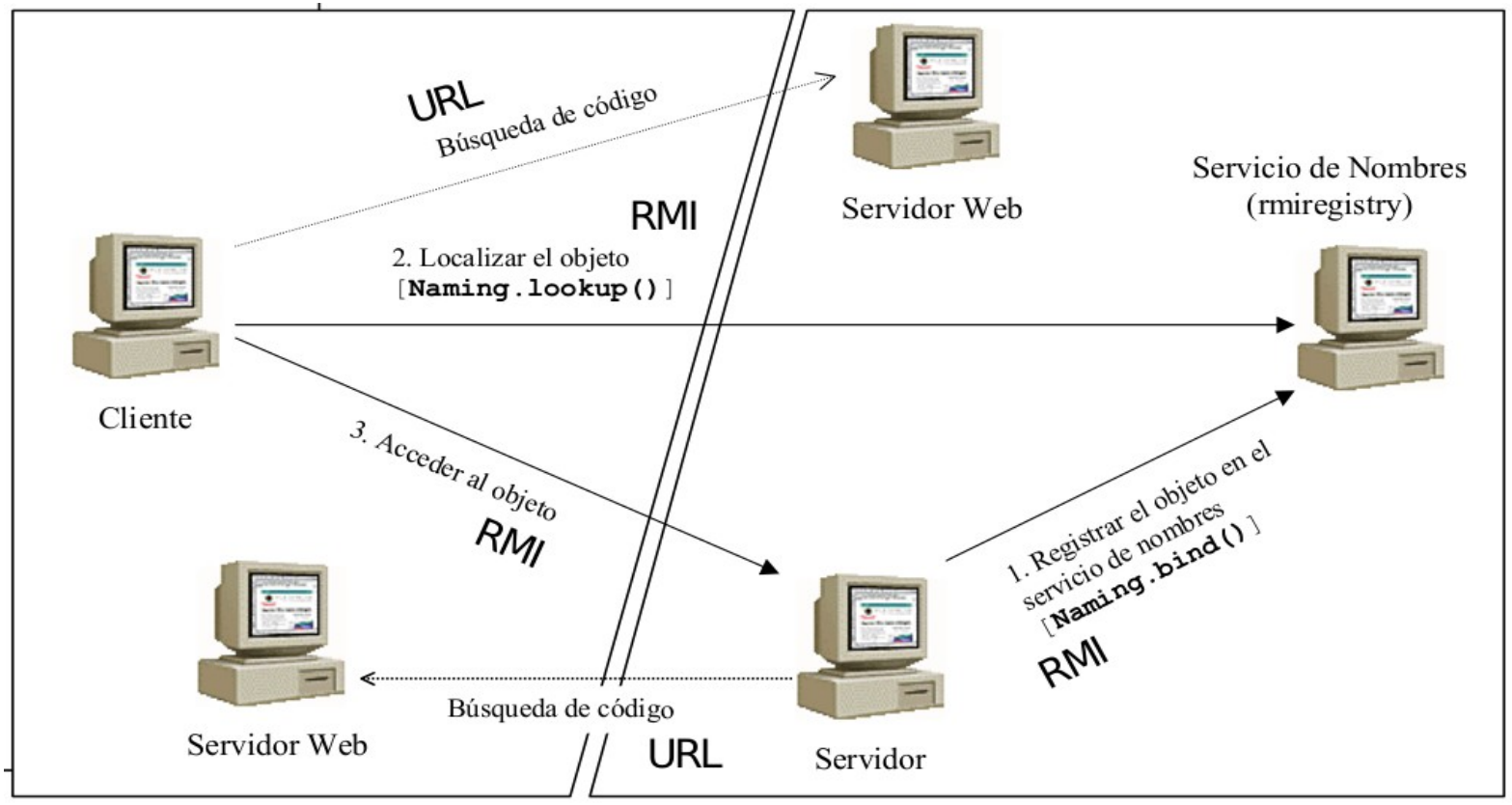
### Vinculación entre objetos

- Se suele usar siempre un servidor de nombres.
- Pasos a seguir:
  - Creación del objeto.
  - Registrar el objeto en un servidor de nombres bajo un nombre concreto.
  - Para usar el objeto, se accede al servidor de nombres.
- Habitualmente
  - El registro de un objeto lleva consigo una referencia al objeto.
  - Esta referencia se obtiene del servidor de nombres, de forma que otro cliente puede usar el objeto.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.3. Modelo de objetos distribuidos

### Ejemplo: Java RMI



# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.4. Comunicación de grupos

- La idea básica es muy sencilla: un mensaje enviado a un grupo llega a todos los miembros del grupo. Sin embargo no es tan sencillo.
- Un grupo es un conjunto de procesos, que pueden estar en cualesquiera máquinas del sistema distribuido. Los grupos son dinámicos, es decir, que pueden crearse, destruirse, incorporar nuevos componentes, o excluir procesos pertenecientes. Un proceso puede además pertenecer a varios grupos.
- Se pretende lograr un mecanismo de comunicación transparente. No vale, por lo tanto, utilizar múltiples llamadas a send/receive o múltiples llamadas RPC.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.4. Comunicación de grupos

- Algunas **redes** proporcionan servicios básicos de comunicación múltiple de dos tipos:
  - **Multicasting**, que consiste en que la red proporciona un conjunto de direcciones que pueden ser utilizadas por varios procesos simultáneamente para escuchar por ellas.
  - **Broadcasting** o difusión, que consiste en que la red proporciona una dirección de red especial que hace que todos los mensajes enviados a esa dirección se entreguen en todas las máquinas.
- En el caso de que no exista ninguno de estos servicios, se pueden simular desde los núcleos de cada máquina, enviando copias de los mensajes a cada miembro del grupo.
- Para mantener la información sobre los grupos y los procesos que los integran se necesita un **proceso servidor de grupos**. Los procesos se registran en el servidor cuando entran en un grupo, y deben notificar su salida cuando dejan de pertenecer.

# 5. Comunicación de procesos en Sistemas Distribuidos

## 5.4. Comunicación de grupos

- Existen dos características que deben darse en la comunicación de grupos para que ésta sea útil y fácil de manejar:
  - **Atomicidad**: Un mensaje enviado a un grupo debe entregarse, bien a todos los miembros, o bien a ninguno.
  - **Ordenación de mensajes**: Es necesario que los mensajes difundidos en un determinado orden sean recibidos por todos los miembros del grupo en ese mismo orden **(orden temporal global)**.
- En redes que soporten difusión, el orden temporal global está garantizado. Si no disponemos de difusión en la propia red, el mantenimiento del orden temporal global es complicado.
- A veces se utiliza un **orden temporal consistente**, en el cual, si dos procesos hacen difusiones casi simultáneas de forma independiente, tal vez no sea posible decidir cuál de los dos lo hizo primero, y, probablemente no importe, por lo que se elige uno cualquiera de los dos como primero, y se hace cumplir el **orden temporal global**.

# 6. Sincronización en Sistemas Distribuidos

## 6.1. Sincronización de relojes

- Los algoritmos distribuidos deben siempre evitar la centralización de servicios. En muchas ocasiones los programas necesitan conocer el tiempo.
- Por lo tanto es necesaria la utilización de algún mecanismo que permita conocer el tiempo de forma global en los sistemas distribuidos, pero evitando un servicio centralizado.
- El problema surge porque los relojes locales de los computadores no son exactos, y se adelantan y retrasan unos respecto a los otros.
- Podemos tener dos niveles de necesidades diferentes en cuanto al tiempo:
  - Conocer la hora real.
  - Simplemente conocer si dos sucesos han ocurrido en un cierto orden. En el primer caso necesitamos relojes físicos, mientras que en el segundo nos bastan los denominados relojes lógicos.

# 6. Sincronización en Sistemas Distribuidos

## 6.1. Sincronización de relojes. Relojes lógicos.

- Existen multitud de casos, en los que basta con conocer el orden de los sucesos, no su hora exacta.
- El algoritmo de ***Lamport*** proporciona un **reloj lógico** que permite establecer un orden entre sucesos relacionados. Se basa en la definición de una relación `a→b` ("`a` ocurre antes que `b`"), siendo `a` y `b` eventos.
  - Si `a` y `b` ocurren en el mismo proceso, y `a` ocurre antes que `b`, entonces `a→b`.
  - Si `a` consiste en el envío de un mensaje en un proceso y `b` en la recepción del mismo en otro proceso, entonces `a→b`.
  - Si `a→b` y `b→c`, entonces `a→c` (propiedad transitiva).
- Esta es una relación de orden, pero no total, ya que no hay forma de ordenar dos eventos independientes en procesos independientes.

# 6. Sincronización en Sistemas Distribuidos

## 6.1. Sincronización de relojes. Relojes lógicos.

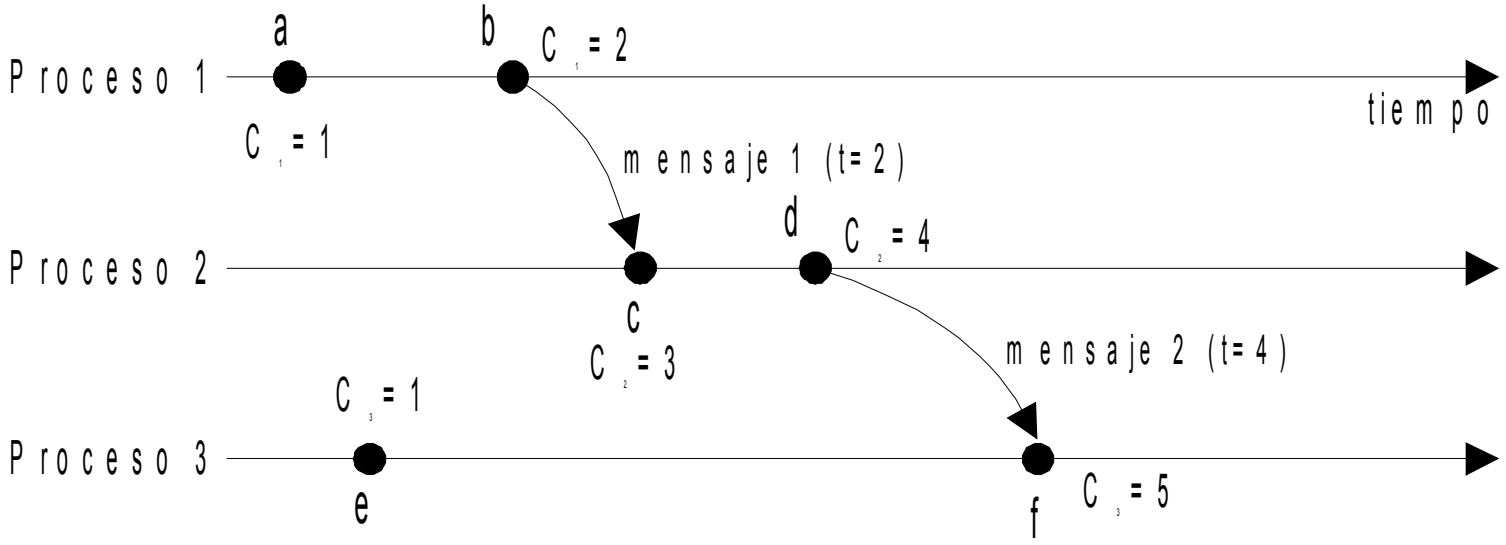
- A partir de esta relación se crea un reloj lógico mediante cualquier función `C(x)` que verifique que si `a→b`, entonces `C(a) →C(b)`. Cada proceso `p` guarda su propio reloj lógico `Cp`, y cada evento que ocurre en el proceso se marca con el valor del reloj (lo que se denomina *timestamp* o etiqueta temporal). Un ejemplo:

  1. En un proceso `p`, antes de que se produzca un evento, `Cp` es incrementado, de modo que `Cp = Cp+1`. Cuando se produzca el siguiente evento se le etiqueta con el valor de `Cp`.

  2. Cuando un proceso `p` envía un mensaje `m`, el mensaje transporta un valor `t` que es el valor del reloj lógico `Cp` del proceso. Cuando el mensaje es recibido por un proceso `q`, entonces `q` calcula el máximo `Cq = máx(Cq, t)` y aplica el paso anterior antes de etiquetar al evento de recibir el mensaje `m`.

# 6. Sincronización en Sistemas Distribuidos

## 6.1. Sincronización de relojes. Relojes lógicos.



# 6. Sincronización en Sistemas Distribuidos

## 6.1. Sincronización de relojes. Relojes físicos.

- A veces las aplicaciones necesitan conocer el tiempo exacto. Pero no es fácil conocer el tiempo exacto.
- Un día solar es el tiempo que transcurre desde que el sol alcanza su punto más alto en el horizonte hasta que vuelve a alcanzarlo. Un día tiene 24 horas, y entonces un segundo es 1/86400 días solares. Pero en 1940 se descubrió que la duración de los días no es constantes. Es más, hace 300 millones de años, los días eran tan cortos que el año tenía 400 días. Las turbulencias del núcleo de hierro de la tierra hacen que, por otro lado, la duración del día tenga variaciones esporádicas.
- Por lo tanto, lo que conocemos como segundo solar medio es la duración de un segundo tomando como referencia el valor medio de una muestra significativa de días.

# 6. Sincronización en Sistemas Distribuidos

## 6.1. Sincronización de relojes. Relojes físicos.

- En 1948 se inventó el reloj atómico, que define el segundo como el tiempo que tarda en isótopo 133 de cesio en realizar 9192631770 transiciones (tiempo equivalente a un segundo solar medio el día que se introdujo esta medición). El tiempo atómico es más preciso que el solar, pero sigue teniendo derivas, por lo que se hace la media de 50 laboratorios que disponen de relojes atómicos y se obtiene lo que se denomina TAI (Tiempo Atómico Internacional).
- Un día solar equivale a 86400,003 segundos TAI (los segundos TAI son más cortos que los solares), por lo que se produce un desajuste entre la hora TAI y la hora solar. Para solucionarlo, cuando la deriva es de 0,8 segundos, se adelanta la hora TAI para hacerla coincidir con la hora solar. Este tiempo, que contiene una cierta irregularidad, se conoce como UTC (Tiempo Universal Coordinado), y es el que se utiliza actualmente como base del registro de tiempo (sustituyendo al antiguo estándar solar, GMT, Greenwich Meridian Time).

# 6. Sincronización en Sistemas Distribuidos

## 6.1. Sincronización de relojes. Relojes físicos.

- El tiempo UTC se proporciona desde algunos organismos por diversos mecanismos (satélite, radio de onda corta, teléfono, etc.) con una precisión de entre 10 y 0,5 milisegundos, pero a un precio que oscila entre 100.000 y varios millones de pesetas.
- En conclusión, puede conocerse el tiempo más o menos exacto, pero a un precio muy alto.

# 6. Sincronización en Sistemas Distribuidos

## 6.1. Sincronización de relojes. Sincronización de Relojes.

- La medida del tiempo en los computadores se hace mediante un oscilador de cristal. Los osciladores de cristal son poco precisos, y hacen que los computadores cuenten el tiempo más rápido o más despacio de lo normal.
- Cada computador tiene su propia deriva, por lo que dos computadores que tengan la misma medida de tiempo (es decir, que estén sincronizados), no la seguirán teniendo transcurrido un tiempo.
- Lo que hay que hacer es volver a sincronizar esos computadores.
- Lo más fácil es conectarlos a una emisora que proporcione el tiempo correcto y poner los relojes al valor adecuado. Pero esta es una medida muy cara. La solución está en poner de acuerdo a los computadores para que se sincronicen entre ellos.

# 6. Sincronización en Sistemas Distribuidos

## 6.1. Sincronización de relojes. Sincronización de Relojes.

- En el algoritmo de **Cristian** se dispone de una fuente de hora UTC que proporciona la hora correcta a una de las máquinas del sistema. Ésta la distribuye periódicamente entre todas las máquinas a través de la red. Pero hay dos problemas:
  - Los relojes nunca pueden ser retrasados porque muchas aplicaciones funcionan basándose en que el tiempo no retrocede (p.e. make). Para solucionar este problema, podemos hacer que los ticks que incrementan el valor del reloj sean más lentos hasta converger con la hora correcta, o incluso omitir algún tick.
  - Otro problema es el retraso que puede sufrir por la red el mensaje que se envía con la hora correcta. La solución es calcular el tiempo que tarda el mensaje por la red a partir del tiempo que tarda en llegar al servidor la réplica del mensaje distribuido. Este tiempo, divido entre 2, es una estimación de lo que pudo tardar el mensaje en llegar a su destino, y se puede añadir a la hora distribuida (la próxima vez) para obtener un tiempo más exacto.

# 6. Sincronización en Sistemas Distribuidos

## 6.1. Sincronización de relojes. Sincronización de Relojes.

- Si no se dispone de la hora real, podemos adoptar la solución del algoritmo de **Berkeley**, utilizando en UNIX BSD:
  - Un servidor realiza un escrutinio periódico de la hora entre todas las máquinas, calcula la hora media y la redistribuye. Podemos mejorar este algoritmo eliminando la centralización, y distribuyendo la tarea entre todas las máquinas de manera que cada computador realice su propio escrutinio

# 6. Sincronización en Sistemas Distribuidos

## 6.2. Exclusión mutua.

- En los sistemas multiproceso varios procesos pueden intentar acceder al mismo recurso, y necesitamos mecanismos de exclusión mutua. En los sistemas tradicionales empleamos semáforos y cerrojos para reservar los recursos, pero ambos mecanismos se basan en la compartición de memoria. En los sistemas distribuidos no existe memoria común, luego necesitamos otro mecanismo.
- La primera idea que nos surge es la creación de un servidor de semáforos, es decir un proceso que se pueda comunicar con los procesos interesados y que coordine el acceso al recurso. Este servidor puede fácilmente proporcionar todas las propiedades que estén presentes en los semáforos tradicionales, por lo que es perfectamente válido. Sin embargo no es adecuado porque es un método centralizado.

# 6. Sincronización en Sistemas Distribuidos

## 6.2. Exclusión mutua.

- El algoritmo de **Ricart y Agrawala** soluciona el problema de la exclusión mutua de forma distribuida, aunque es más lento, complicado y débil. Su funcionamiento exige las siguientes condiciones:

  1. Los procesos se comunican mediante mensajes de capacidad no nula.
  2. Los mensajes pueden llegar a un proceso que esté dentro o fuera de la región crítica.
  3. La comunicación es fiable y los mensajes llegan en orden.
  4. Es necesaria una relación de orden total entre los eventos del sistema.

# 6. Sincronización en Sistemas Distribuidos

## 6.2. Exclusión mutua.

- Cuando un proceso quiere entrar en una región crítica, solicita permiso a los demás procesos con los que compite mediante la difusión de un mensaje en que se incluye un etiqueta temporal. El proceso puede entrar cuando recibe el permiso de todos los demás procesos.
- Cuando un proceso recibe una petición, toma una de las siguientes tres posturas:
  - Si está fuera de la región crítica y no quiere entrar, envía inmediatamente el permiso de entrada.
  - Si está en la región crítica, espera a terminar, y después envía el permiso.
  - Si está fuera de la región crítica pero quiere entrar (y ha enviado la petición correspondiente), envía el permiso sólo si la marca de tiempo de su petición es posterior a la de la petición ajena. En caso de tener la petición anterior, enviará el permiso inmediatamente después de salir de la región.

# 6. Sincronización en Sistemas Distribuidos

## 6.2. Exclusión mutua.

- Cada proceso tiene una cola de peticiones no respondidas, la cual vacía (enviando los permisos) cada vez que sale de la región crítica.
- El problema de este algoritmo es que, para `n` procesos, necesita `2(n-1)` mensajes por cada entrada en la sección crítica, y tiene `n` puntos de fallo, ya que si cualquiera de los procesos falla, ninguno de los otros puede entrar en la región crítica.

# 6. Sincronización en Sistemas Distribuidos

## 6.2. Exclusión mutua.

- Otra posible solución distribuida es el **algoritmo en anillo**, en el que los procesos que compiten se van pasando de forma cíclica un testigo que otorga el privilegio de entrada.
- Cuando un proceso sale de la sección crítica o no quiere entrar en ella, traspasa inmediatamente el testigo al siguiente proceso.
- Este sistema tiene la **desventaja** de que si nadie quiere entrar en la sección crítica, se envían mensajes innecesarios.

# 6. Sincronización en Sistemas Distribuidos

## 6.3. Algoritmos de Elección.

- Muchos algoritmos se basan en la utilización de un proceso coordinador.
- Para evitar que éste sea un punto de fallo, es necesario que los procesos puedan elegir nuevos coordinadores.

1. El **algoritmo del matón** soluciona este problema. Cada vez que un proceso `P` se integra en el grupo (por ejemplo cuando se recupera después de haber caído), o se da cuenta de que el coordinador ha caído, inicia el proceso de elección.

# 6. Sincronización en Sistemas Distribuidos

## 6.3. Algoritmos de Elección.

- **Algoritmo del matón.** Proceso de elección:

  1. `P` envía un mensaje a los procesos con identificador mayor que `P`.
  2. Si no responde ninguno, `P` es el nuevo coordinador; si responde alguno, `P` no puede ser el coordinador.
  3. Si un proceso `Q` recibe un mensaje, lo responde, y además envía mensajes a todos los procesos con identificador mayor que `Q`. Si no responde nadie, `Q` es el nuevo coordinador.
  4. El nuevo coordinador se lo comunica a los demás.

# 6. Sincronización en Sistemas Distribuidos

## 6.4. Transacciones atómicas.

- Las transacciones atómicas pretenden solucionar el problema causado por operaciones que fracasan antes de terminar y que dejan el sistema en un estado inestable.
- El ejemplo habitual de la operación bancaria que extrae dinero de una cuenta y lo ingresa en otra, ilustra sobradamente cuál es el problema que se pretende solucionar.
- Más formalmente, las transacciones atómicas persiguen dos objetivos:
  - Enmascarar los accesos concurrentes a los datos.
  - Evitar los fallos que dejen el sistema en un estado inconsistente.

# 6. Sincronización en Sistemas Distribuidos

## 6.4. Transacciones atómicas.

- Así, en un servicio transaccional, podemos suponer que las operaciones realizadas están protegidas del efecto de otras acciones concurrentes (otras operaciones sobre las mismas cuentas) y que, o bien se completan con éxito, o bien no se efectúan en absoluto (nunca quedan a medias).
- Para ello es necesario establecer explícitamente el comienzo y finalización de una transacción. Siguiendo con el ejemplo del banco, si disponemos de un servicio de transacciones, podemos asegurar que las siguientes operaciones se realizarán correctamente y sin interferir con otras transacciones, o bien no se realizará ninguna de ellas:

```
IniciarTransacción
   Retirar(Cuenta1, Cantidad1)
   Ingresar(Cuenta2, Cantidad1)
   Retirar(Cuenta3, Cantidad2)
   Ingresar(Cuenta4, Cantidad2)
FinalizarTransacción
```

# 6. Sincronización en Sistemas Distribuidos

## 6.4. Transacciones atómicas.

- Además de contar con primitivas de inicio y finalización de transacción, necesitamos otra que nos permita abortar una transacción, que se utilizará en caso de que la propia transacción detecte un problema que impida la correcta finalización.
- Cuando el servidor o el cliente que ejecutan la transacción fallan, las transacciones que no han terminado son abortadas y se restauran los valores originales.
- Es posible establecer plazos de recuperación, finalizados los cuales, se inicia el proceso de restauración.

# 6. Sincronización en Sistemas Distribuidos

## 6.4. Transacciones atómicas.

- En resumen, necesitamos 3 **primitivas**:
  - **`IniciarTransacción`**, que inicia la transacción y devuelve un identificador.
  - **`FinalizarTransacción`**, que termina una determinada transacción e indica si terminó con éxito (comprometida, cometida, *committed*) o fue abortada por un fallo.
  - **`AbortarTransacción`**, que la aborta una determinada transacción.

```
IniciarTransacción --> ID_Transacción
FinalizarTransacción (ID_Transaccion) --> (Cometido, Abortado)
AbrtarTransacción (ID_Transaccion)
```

# 6. Sincronización en Sistemas Distribuidos

## 6.4. Transacciones atómicas.

- Las transacciones atómicas deben cumplir las siguientes **propiedades**:
  - **Atomicidad**. La transacción se ejecuta completamente o no se realiza en absoluto. No es posible por lo tanto acceder a estados intermedios.
  - **Consistencia**. Esta propiedad asegura que los datos no van a quedar en un estado inconsistente como resultado de una transacción fallida.
  - **Aislamiento o serializabilidad**. Una transacción debe ejecutarse de forma independiente de otras transacciones concurrentes que accedan a los mismos recursos, sin que se produzcan interferencias. Es decir, que el resultado debe ser el mismo que si las transacciones tuvieran lugar en serie.
  - **Durabilidad**. La finalización exitosa de una transacción asegura que los cambios realizados son ya permanentes, y no es posible volver al estado anterior.

# 6. Sincronización en Sistemas Distribuidos

## 6.4. Transacciones atómicas.

### Recuperación de transacciones

- La forma de recuperar los valores antiguos cuando se aborta (rollback) una transacción depende de la implementación.
  - Una de las formas más sencillas es mediante el uso de un **registro** (log) de las operaciones que se deben ir realizando. Así, las operaciones de una transacción no se ejecuta realmente, sino que se anotan en el registro. Si la transacción finaliza con éxito, se escriben el resultado sobre los datos originales y se anota la finalización exitosa; si fracasa, se recorre el registro hacia atrás, deshaciendo las operaciones efectuadas y anotadas. Para que esto sea posible, es necesario anotar en el registro, para cada operación, los valores anteriores y posteriores de los datos afectados por la operación.
  - Otra forma de lograr la recuperación es trabajar con **copias de los datos,** de manera que las operaciones de una transacción se hacen sobre una copia de los datos, y los datos originales sólo se modifican cuando termina la transacción con éxito.

# 6. Sincronización en Sistemas Distribuidos

## 6.4. Transacciones atómicas.

### Transacciones Distribuidas

- Cuando una transacción se ejecuta distribuida entre varias máquinas, es necesario un protocolo que permita saber si terminó bien en todas ellas y por tanto finalizó correctamente, o si falló en alguna, y por tanto debe ser abortada.
- Además, en cada máquina es necesario recuperar los datos antiguos en caso de fallo global, o establecer los nuevos datos en caso de éxito global.

# 6. Sincronización en Sistemas Distribuidos

## 6.4. Transacciones atómicas.

### Transacciones Distribuidas

- Para utilizar transacciones distribuidas es necesario que un proceso coordine a los procesos que ejecutan la transacción (trabajadores), por lo que se necesitan dos nuevas primitivas: **`AñadirServidor`** y **`NuevoServidor.`**
- Con **`AñadirServidor`**, un cliente que quiere ejecutar una transacción, informa a un trabajador de que está implicado en una transacción y le indica quién es su coordinador.
- Con **`NuevoServidor`**, un trabajador recién incorporado indica a su coordinador que trabaja en una determinada transacción.

# 6. Sincronización en Sistemas Distribuidos

## 6.4. Transacciones atómicas.

### Transacciones Distribuidas

- Cuando una transacción va a terminar (se invoca `FinalizarTransacción`) comienza el **protocolo en dos fases**:
  - El coordinador envía a cada trabajador un mensaje indicando que la transacción ha terminado, y anota esta situación en el registro ("Preparado"). Cada trabajador debe responder y anotar en su registro si todo ha ido bien ("Preparado") y se puede terminar, o si hay algún error y debe abortarse ("Abortar").
  - El coordinador recibe todas las respuestas. Si todas son favorables, la transacción puede finalizarse, pero si hay algún respuesta de aborto, la transacción debe abortarse. Cualquiera que sea la decisión, el coordinador se la comunica a todos los trabajadores y al cliente de la transacción, y lo anota en el registro ("Cometido" o "Abortado").
  - Cada trabajador responde al mensaje que han terminado o abortado con éxito ("Terminado").

# 6. Sincronización en Sistemas Distribuidos

## 6.4. Transacciones atómicas.

### Transacciones Distribuidas

- Los fallos en los trabajadores y en el coordinador durante el protocolo se pueden recuperar gracias al registro.
- Si el coordinador cae después de haber escrito "preparado", vuelve a comenzar el escrutinio
- Si cae después de haber difundido la conclusión, vuelve a difundirla.
- Si es un trabajador el que cae, el servidor reenviará los mensajes hasta que responda.