CONCURRENCIA DE SISTEMAS DISTRIBUIDOS

SEGUNDO PARCIAL

TEMA 6 – SISTEMAS DE TIEMPO REAL

INTRODUCCIÓN

Un sistema de tiempo real (STR) es un sistema informático que interacciona repetidamente con su entrono físico, respondiendo a los estímulos dentro de un plazo de tiempo determinado. Para que el funcionamiento del sistema sea correcto no basta con que las acciones sean correctas, sino que deben ejecutarse dentro el intervalo de tiempo especificado.

Tiene como requisitos:

- Realizar actividades periódicas solicitadas a intervalos periódicos, que deben completar un trabajo antes de un plazo.
- Un sistema con varios computadores, hardware E/S y software con cierto propósito. Este interacciona con el entorno de manera que debe reaccionar con cierto tiempo a la vez a todos los estímulos que reciba. Como resultado, se imponen requisitos temporales sobre el software, de naturaleza concurrente.

Un STR debe responder a los estímulos de su entorno dentro de un intervalo de tiempo determinado por las propiedades dinámicas del entorno.

El intervalo de respuesta se caracteriza por un plazo *(Deadline)*. Según lo que pase, si se supera este deadline, tendremos las siguientes tres opciones:

- **Estricto o crítico (hard):** una respuesta fuera de plazo es inadmisible.
- **Firme (firm):** una respuesta fuera de plazo no tiene utilidad.
- **Flexible (soft):** una respuesta fuera de plazo tiene una utilidad reducida.

CARACTERÍSTICAS DE LOS STR

- **Concurrencia:** Los componentes del sistema controlado simultáneamente, por lo que el STR debe atenderlo y generar las acciones de control simultáneamente.
- **Interacción con dispositivos:** Los STR interaccionan con su entorno mediante diversos tipos de dispositivos no convencionales, como convertidores A/D y D/A, o sensores. Por otra parte, los componentes del software que controlan el funcionamiento de estos dispositivos son, generalmente, dependientes del sistema concreto.
- **Fiabilidad y seguridad:** un fallo en un sistema de control puede hacer que el sistema controlado se comporte de forma peligrosa o antieconómica. Hay que asegurar que, si el sistema de control falla, lo haga de forma que el sistema controlado quede en un estado seguro.
- **Determinismo temporal:** acciones en intervalos de tiempo determinados. Es fundamental que el comportamiento temporal de los STR sea determinista. No es lo mismo que sea eficiente, solo debe responder correctamente en todas las situaciones. Hay que prever el comportamiento en el peor caso.
- **Predictibilidad:** la predictibilidad es uno de los objetivos principales en sistemas de tiempo real. Se debe realizar un análisis de la planificabilidad de las tareas de un sistema de tiempo real para predecir si las tareas cumplirán sus requisitos temporales.

PLANIFICACIÓN DE TAREAS EN STR

El objetivo es planificar el uso de los recursos del sistema (en particular el procesador), para poder garantizar los requisitos temporales de las tareas.

Un paradigma de planificación proporciona dos elementos:

- Un algoritmo de planificación que determina el orden de acceso de las tareas a los recursos del sistema.
- Un método de análisis que permite calcular el comportamiento temporal del sistema.

Debemos conocer el algoritmo de planificación por prioridades fijas expulsivas. Aquí, la prioridad, además de un mecanismo, es un atributo de las tareas normalmente ligado a su importancia relativa en el conjunto de tareas. En las STR, esta planificación es la más popular, debido a:

- El comportamiento temporal es más fácil de entender y predecir.
- El tratamiento ante sobrecargas es fácil de prever.
- Existen técnicas analíticas completas.
- Es requerido por estándares de sistemas operativos y lenguajes concurrentes.

Consideraremos inicialmente un modelo de tareas simple:

- Conjunto de tareas estático, $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$
- Todas las tareas son periódicas, con periodo T_i
- Las tareas no cooperan entre sí, es decir, son independientes.
- Se conoce el tiempo máximo de ejecución de cada tarea C_i
- Cada tarea tiene un plazo $D_i \leq T_i$

Respecto a la ejecución de las tareas, asumimos que:

- La política de planificación es expulsiva por prioridades.
- La máquina está dedicada a la aplicación.
- Los cambios de contexto tienen coste cero.
- Las tareas no pueden suspenderse voluntariamente. Una vez una tarea está preparada, no puede demorar su ejecución.

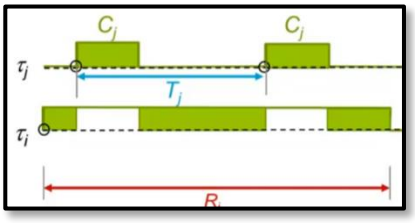
Principios Básicos

Dos conceptos ayudan a construir la situación de peor caso, en el caso de tareas periódicas independientes:

- **Instante crítico:** el tiempo de respuesta de peor caso de todas las tareas se obtiene si las activamos todas a la vez.
- **Basta comprobar el primer plazo:** luego de un instante crítico, si una tarea cumple su primer plazo, cumplirá todos los plazos posteriores.

Basándose en estos conceptos, se obtiene el test de planificabilidad.

El tiempo de respuesta de una tarea es la suma de su tiempo de cómputo (C_i) más la interferencia que sufre por la ejecución de tareas más prioritarias (I_i): $R_i = C_i + I_i$



- El número de activaciones de la tarea j en R_i es: $\left\lceil \frac{R_i}{T_j} \right\rceil$
  - $\lceil x \rceil$: techo (x), es el menor número entero mayor o igual que x.
- Para una tarea de prioridad superior, usamos: $I_i^j = \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j$
- Para todas las tareas de prioridad superior: $I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j$

Test de planificabilidad: en un sistema de tiempo real, con n tareas independientes, planificadas mediante un esquema expulsivo de prioridades fijas, y para una asignación concreta de prioridades a las tareas, se cumplirán todos los plazos de respuesta si, y solo si:

- $\forall i, 1 \leq i \leq n, R_i \leq D_i$, donde $R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j$

El tiempo de respuesta (R_i) es la solución mínima de la ecuación:

$$w = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w}{T_j} \right\rceil \times C_j$$

Se puede resolver mediante la relación de recurrencia:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times C_j$$

Un valor inicial aceptable es:

$$w_i^0 = C_i + \sum_{j \in hp(i)} C_j$$

El cálculo finaliza, cuando:

- $w^{n+1} = w^n$
- $w^{n+1} > D_i$ (No se cumple el plazo)

La Sincronización entre Tareas

Una de las restricciones más importantes del test de planificabilidad anterior es la de suponer que las tareas son independientes. Pero eso no siempre es así, dado que en la mayoría de los sistemas de interés práctico, las tareas necesitarán comunicarse. Cuando esto pasa, hace falta sincronización.

Asumiremos que las tareas solo necesitan exclusión mutua en el acceso a sus secciones críticas, y que las guardan utilizando alguna primitiva de sincronización equivalente a los locks.

Sin embargo, en STRs esta solución no es válida, ya que puede producir una situación denominada "inversión de prioridades", que invalida el test de planificabilidad.

La inversión de prioridad no acotada ocurre cuando una tarea menos prioritaria se ejecuta por delante de una más prioritaria. El tiempo total de bloqueo debido a una tarea menos prioritaria no está acotado, al poder intervenir tareas de prioridad intermedia.

Esta no puede evitarse. Los bloqueos comprometen la planificabilidad del sistema. Se trata de conseguir que el bloqueo esté acotado, se produzca el menor número de veces posible y que sea medible.

Los protocolos de sincronización en tiempo real evitan la inversión de prioridad no acotada. Los más importantes son:

- Herencia de prioridad.
- Techo de prioridad inmediato, o protección por prioridad.

Ambos hacen que la inversión de prioridad, o retraso que una tarea sufre a causa de tareas de prioridad inferior:

- Sea una función de la duración de una o varias secciones críticas.
- No sea función de la duración de tareas completas.

Protocolo del Techo de Prioridad Inmediato

A cada semáforo le asociaremos un techo, equivalente a la mayor prioridad de las tareas que lo pueden usar.

Con este protocolo, una tarea que accede a un semáforo hereda inmediatamente el techo de prioridad del semáforo. La sección crítica se ejecuta con la mayor prioridad de los techos de los semáforos que la guardan. Tiene las siguientes propiedades:

- Cada tarea se puede bloquear una vez como máximo en cada ciclo. Además, si una tarea se bloquea, lo hace al principio del ciclo.
- No puede haber interbloqueos.

El cálculo de los factores de bloqueo máximos se realiza con B_i como factor de bloqueo, el cual es el tiempo máximo de retraso que sufre la tarea i.

Para el protocolo del techo de prioridad inmediata:

- B_i es la duración máxima de todas las secciones críticas, cuyo techo es $\geq$ que la prioridad de la tarea i, que son utilizadas por tareas de prioridad inferior.
- Para la tarea de menor prioridad: $B_n = 0$
- Para el resto de tareas: $B_i = \max(C_{k,s}) \rightarrow \{k, s \mid k \in lp(i) \wedge s \in usa\ (k) \wedge techo(s) \geq prior(i)\}$

El test inicial en base a prioridades fijas, venía de resolver una relación de recurrencia como la siguiente:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j$$

TEMA 7 – SISTEMAS DISTRIBUIDOS

DEFINICIÓN DE SISTEMA DISTRIBUIDO

Es una extensión natural de los sistemas concurrentes, cuando estos se ejecutan en más de un ordenador. Un Sistema Distribuido (SD) es un conjunto de máquinas autónomas (nodos interconectados en red). Es un Sistema Concurrente con:

- Paralelismo real (varios procesadores).
- No comparten reloj interno que los sincronice, lo hacen mediante otros mecanismos como los mensajes.
- No comparten memoria.
- La información se traspasa mediante el intercambio de mensajes.
- Cooperan entre sí.
- Ofrecen imagen de sistema único.

A cada ordenador del sistema distribuido le llamamos nodo.

- Los nodos se conectan unos con otros mediante canales de comunicación.
- El resultado es un grafo, donde los ordenadores son los nodos y las aristas son los canales de comunicación.
- Dependiendo de la forma del grafo, tendremos diferentes topologías.

Tenemos tres puntos de vista:

- **A nivel de hardware:** máquinas autónomas. Entre ellas no se comparte hardware, se ejecutan y fallan de forma independiente unas con otras.
- **A nivel de usuario:** imagen de sistema único. Se crean diferentes tipos de transparencia. Se oculta que el sistema está distribuido entre varios ordenadores.
- **A nivel interno:** algoritmos distribuidos. Cada nodo ejecuta una parte del algoritmo concurrentemente con los otros nodos. Se comunican entre sí y se sincronizan, habitualmente mediante mensajes.

Un recurso debe entenderse en sentido amplio, como una impresora, computadoras o dispositivos de almacenamiento.

Ventajas

- **Acceso a recursos remotos:** desde cualquier nodo.
- **Escalable:** acepta cargas crecientes.
- **Disponible:** servicio siempre activo y fallos solucionables con replicación.

Desventajas

- **Complejidad:** cada nodo un sistema centralizado, y tienen cooperación entre ellos.
- **Seguridad:** múltiples puntos de ataque potenciales.

MIDDLEWARE

Se define como una capa de software por encima del sistema operativo, pero bajo el programa de aplicación que proporciona una abstracción común de programación a lo largo de un sistema distribuido.

Desde el punto de vista del programador de aplicaciones, se trata de una simple API con el que acceder a servicios distribuidos. El uso de la API desencadena un trabajo distribuido entre los diferentes nodos del sistema, que colaborarán para ofrecer el servicio distribuido.

El uso de servicios middleware permite el desarrollo de aplicaciones distribuidas, evitando al programador el desarrollo de parte de su complejidad: *transparencia, disponibilidad, escalabilidad o seguridad.*

Así mismo, al desarrollar un sistema distribuido, será conveniente ofrecer los servicios en forma de middleware, idealmente mediante interfaces.

TIPOS DE SISTEMAS DISTRIBUIDOS SEGÚN MIDDLEWARE

Los SDs son complejos, y hay varios tipos según sean puros o extendidos.

- **Puro:** hardware (nodos + red). Este es variado, complejo y heterogéneo.
- **Extendido:** SD puro + middleware. Estos son mucho más fáciles de programar.

Tenemos los siguientes tipos de middleware:

- **Orientados a objetos:** RMI *(Remote Method Invocation)*. Un objeto remoto es un objeto que no reside en el mismo espacio de direcciones, sino en otro nodo. Permite programar invocaciones de objetos como si fuesen locales.
- **Orientados a comunicaciones:** elementos de comunicación intermedios.
- **Orientados a eventos:** emisores y oyentes de eventos, publicación y suscripción, etc.

JavaRMI – Ejemplo de Middleware

JavaRMI *(Remote Method Invocation)* es un middleware de comunicación orientado a objetos. Permite invocar métodos de objetos java de otra JVM, y pasar objetos Java como argumentos cuando se invocan dichos métodos.

- **Objeto remoto:** para cada objeto remoto, RMI crea dinámicamente un objeto llamado esqueleto.
- **Invocación:** para invocar un objeto remoto desde otro proceso, se utiliza un Proxy: permite localizar al esqueleto del objeto remoto.

El servidor de nombres permite registrar objetos remotos, para que puedan ser localizados. Hace de base de datos de objetos. Puede residir en cualquier nodo, y es accedido desde el cliente o el servidor usando la interfaz local llamada "*registry*". En las distribuciones Oracle de Java, el servidor de nombres se lanza usando la orden "`rmiregistry`".

Transparencia

Logrando transparencia, se ofrecen servicios sencillos al usuario, dado que se le oculta la complejidad de los algoritmos que se ejecutan, los fallos que pueden suceder, el número de ordenadores involucrados, su ubicación y todas aquellas características que sean irrelevantes para el usuario final.

Todo esto se agrupa en la "transparencia de distribución". Para lograrla, se deben ocultar diferentes aspectos específicos llamados "Ejes de la transparencia de distribución". Estos son los tipos de transparencia:

- **Transparencia de ubicación:** se oculta al usuario la ubicación de los recursos, oculta donde se guardan las cosas.
  - **Nombres simbólicos:** para lograr transparencia de ubicación, los recursos suelen estar identificados con nombres simbólicos únicos. Cuando los usuarios deseen acceder a los recursos, utilizarán estos nombres simbólicos, luego el sistema los traducirá en sus nombres transparentes para llegar a su verdadera ubicación.
  - **Servicios de nombres:** servicios de localización y búsqueda de recursos, servicios de directorio, etc.

- **Transparencia de fallos:** Se oculta el hecho de que los componentes de un sistema distribuido fallen, se consigue que ocurra un fallo y no lo sepas. Cuantos más ordenadores forman parte de un sistema distribuido, más alta es la probabilidad de que falle alguno de ellos.
  - **Detectores de fallos:** los nodos están siendo monitorizados continuamente.
  - **Replicación:** Todos los recursos están replicados en más de un nodo. Si se detecta el fallo de un nodo, otro nodo será encargado de continuar dando el mismo servicio sobre la copia del recurso.
  - **Algoritmos tolerantes a fallos:** Todos los algoritmos que se ejecutan en el sistema deben reaccionar adecuadamente al fallo de un nodo, ofreciendo el mismo servicio antes y después del fallo.

- **Transparencia de replicación:** Se oculta el hecho que los recursos están replicados en más de un nodo, que hay un fichero con varias copias, varios servidores que hacen exactamente lo mismo. Los sistemas se replican para lograr sistemas disponibles (tolerancia a fallos) y para aumentar la escalabilidad.
  - **Mapeo:** mapeo de los nombres simbólicos de recursos a la ubicación de diferentes réplicas.
  - **Algoritmos de balanceo de carga:** algoritmos de balanceo de carga, para elegir qué réplica sirve cada petición sobre cada recurso.
  - **Algoritmos de replicación:** algoritmos de replicación que ofrezcan consistencia entre las réplicas.

Tenemos otros tipos de transparencia, menos importantes, que son:

- **Transparencia de persistencia:** se oculta el hecho de que los recursos están almacenados en disco.
- **Transparencia de concurrencia:** se oculta el hecho de que el sistema está siendo utilizado por múltiples usuarios al mismo tiempo.
- **Transparencia de migración:** se oculta el hecho de que los recursos pueden moverse.
- **Transparencia de acceso.**
- **Transparencia de transacción.**
- **Transparencia de reubicación.**
- **...**

Ofrecer transparencia tiene un coste elevado, en mayor número de ordenadores, en ordenadores y redes de mayores prestaciones, en mayor coste de desarrollo y mantenimiento se los sistemas y en mayor coste algorítmico.

A veces, no interesa transparencia total en alguno de los ejes. La convivencia depende de cada sistema en particular. Por otra parte, de vez en cuando la transparencia total es imposible de lograr, o tendría demasiado coste para determinado sistema.

Disponibilidad

Es la probabilidad de que un determinado sistema ofrezca sus servicios a los usuarios.

- **Sistemas altamente disponibles:** cuando la probabilidad es mayor que el 99.999.
- **Sistemas mucho más disponibles:** sistemas críticos en los que las probabilidades son mucho mayores.

Hay tres factores que afectan a la disponibilidad: fallos, tareas de mantenimiento y ataques maliciosos.

- **Fallos:** pueden fallar tanto los nodos como las redes que los comunican. Se deben diseñar sistemas que continúen ofreciendo servicio en presencia de fallos.
- **Tareas de mantenimiento:** todo sistema requiere mantenimiento. Se deben diseñar sistemas que permitan el mantenimiento al mismo tiempo que los usuarios acceden a los servicios.
- **Ataques maliciosos:** todo sistema es objeto potencial de ataques maliciosos. Se deben diseñar sistemas que sean resistentes a los ataques de forma que no se interrumpa el servicio ofrecido.

El mecanismo básico para lograr la tolerancia a fallos en los SDs es la replicación. Es un servicio que se configura como un conjunto de nodos, llamados réplicas, de tal forma que el fallo de una réplica no impida al servicio seguir funcionando.

La replicación introduce el problema de la consistencia: el grado de similitud/diferencia entre las diferentes réplicas. Se pueden entender diferentes tipos:

- **Consistencia fuerte:** idealmente, todas las réplicas son iguales entre sí en todo momento. Este objetivo es imposible de lograr, pero se pueden lograr sistemas con consistencia bastante fuerte, donde los usuarios del servicio obtienen la misma respuesta independientemente de la réplica que les atienda.
- **Consistencia débil:** las réplicas pueden divergir, de forma que cada una puede dar respuestas diferentes en cierto instante.

Tenemos a su vez distintos tipos de fallos:

- **Número de nodos:** los fallos pueden ser simples, afectando a un único nodo o a un único canal de comunicación, o compuestos, afectando a varios nodos y canales simultáneamente.
- **Detectabilidad:** los fallos pueden ser detectables si otro nodo es capaz de observar el fallo, o indetectables en caso de que no pueda ser detectado por otro nodo.

Fallos Simples Detectables

- **Fallo de parada:** el nodo falla deteniéndose. Otro nodo lo puede detectar mediante monitorización continua, o “pings” periódicos.
- **Fallo de temporización:** el nodo falla, tardando demasiado en responder. Otro nodo lo puede detectar con temporizadores asociados a cada petición, también mediante pings periódicos.
- **Fallo de respuesta detectable:** el nodo falla, proporcionando una respuesta equivocada, detectable como tal. Otro nodo lo puede detectar, aceptando únicamente rangos válidos de respuesta.

En general, todos los fallos simples detectables se pueden tratar de igual forma:

1. Se emplea replicación.
2. Si se detecta el fallo de una réplica, se expulsa a la réplica que ha fallado y el resto de las réplicas continúan ofreciendo el servicio.

Fallos Simples Indetectables – Fallos Bizantinos

Un nodo falla, exhibiendo un comportamiento arbitrario o proporcionando una respuesta que no puede detectarse como fallo. Puede deberse a las siguientes causas:

- Errores en el software.
- Errores en el hardware.
- Ataques maliciosos.

Se tratan de forma diferente a los fallos detectables. Se puede emplear replicación también, pero de forma diferente al caso de fallos detectables: cada petición se envía a todas las réplicas, y todas contestan. Se elige la respuesta mayoritaria. Cuando se detecta, se genera una alarma de que falta unanimidad.

Fallos Compuestos

En la mayoría de casos, se tratan de igual forma a la aparición de varios fallos simples de forma consecutiva.

- **Varios fallos de parada:** se trata cada uno de forma independiente y consecutiva.
- **Varios fallos bizantinos:** se tratan suponiendo que habrá mayoría de nodos no afectados por fallos, y la mayoría correcta podrá seguir ofreciendo el servicio.

Las particiones, un tipo de fallo compuesto, merecen especial atención. Se producen varios fallos en nodos o canales de comunicación que dejan al sistema dividido en 2 o más subgrupos.

Teorema CAP (Consistency, Availability, Partitions)

Es imposible lograr un Sistema que ofrezca al mismo tiempo todo: Consistencia Fuerte (C), Disponibilidad Elevada (A) y puedan ocurrir Particiones (P). Algunos sistemas cumplen dos, pero no las tres: CA, CP, AP.

Suponiendo solo fallos simples detectables, que no causen particiones, tenemos: *detectores de fallos, servicio de pertenecencia al grupo y replicación.*

Detectores de Fallos

La detección de fallos suele integrarse en un módulo de detección de fallos incorporado a cada nodo. Este módulo se encarga de monitorizar a otro nodo o a varios nodos y emitir "sospechas de fallo".

Más que detectar un fallo, se suele indicar que el nodo sospecha del fallo de otro nodo. Puede pasar que hay un canal con fallo entre ambos nodos, y por tanto ambos se sospecharán mutuamente.

En casos de sospechar un fallo, el módulo de detección de fallos lo notifica al servicio de pertenecencia al grupo.

Servicio de Pertenecencia al Grupo

Servicio encargado de establecer un acuerdo entre los nodos vivos, sobre qué nodos han fallado. Ante la sospecha de un fallo, o varias sospechas, este servicio inicia una fase de acuerdo, para determinar qué nodo o nodos han fallado.

Si se acuerda el fallo de un nodo, el servicio de pertenencia lo expulsará y notificará a todos los nodos que permanecen "vivos" informándoles del fallo. Estos ignoran los mensajes de los nodos que han fallado y funcionan sin ellos.

Nótese que es posible que, ante cierta sospecha, se determine que el nodo que falla es el que emitió la sospecha, pues los demás nodos acuerdan que la sospecha es infundada.

Todos los fallos simples detectables se convierten a fallos de parada. Ocurra el fallo que ocurra, si lo detectamos y se acuerda el fallo, se expulsa a algún nodo y los demás nodos siguen funcionando, ignorando al nodo que fue expulsado.

Cuando un nodo reciba la notificación del fallo de otro nodo, se reconfigurará para seguir trabajando sin el nodo expulsado.

Replicación

Cada servicio se configura con más de una réplica, de forma que ante fallos, las réplicas que permanecen "vivas" se reconfiguran y continúan ofreciendo el servicio. Esto proporciona transparencia de replicación, solo se percibe mayor o menor disponibilidad.

- **Replicación pasiva:** una única réplica será la primaria, el resto serán secundarias. La replica primaria es la única activa.
- **Replicación activa:** todas las réplicas son iguales. Todas reciben las peticiones, todas procesan las peticiones y todas contestan al cliente.

Esquemas de Replicación Pasiva

Una única réplica será la réplica primaria. El resto son las secundarias. La primaria recibe todas las peticiones por parte de los clientes del servicio, las procesa y responde a los clientes.

Ante cada petición que implique un cambio de estado, la réplica primaria difundirá un mensaje de actualización de estado (checkpoint) a las secundarias.

- Si el mensaje de checkpoint se envía a las secundarias, esperando la correspondiente confirmación, antes de responder al cliente, se tendrá un sistema replicado con consistencia fuerte.
- Si el mensaje de checkpoint se envía a las secundarias más tarde, tendremos un sistema replicado con consistencia débil.

En caso de fallos, se debe reconfigurar de la siguiente manera:

- **Replica secundaria:** la primaria la ignora y manda un checkpoint menos.
- **Replica primaria:** costoso, dado que se compone de varios pasos:
    1. Elegir una réplica secundaria para que asuma el rol de réplica primaria.
    2. Asegurarse que la réplica elegida tiene el estado más reciente posible.
    3. Asegurarse que los clientes pueden encontrar adecuadamente al nuevo primario cuando detecten que el antiguo primario no responde.
    4. Durante la reconfiguración, es muy posible que el servicio no esté disponible.

Este sistema plantea diversas ventajas sobre la replicación activa:

1. Sólo una réplica trabaja, esto es más eficiente durante el tiempo sin fallos.
2. Se pueden replicar servicios cuya implementación no sea determinista.
3. Si fuese necesario acceder a datos externos a las réplicas, solo lo haría la réplica primaria, por lo que no es necesario utilizar exclusión mutua distribuida.

Esquemas de Replicación Activa

Todas las réplicas son activas, pues todas procesan. Este modelo de ejecución es muy restrictivo, e implica en la práctica que las réplicas no tengan concurrencia interna.

- **Utiliza algoritmos de difusión:** implica el uso de algoritmos costosos que proporcionen la misma secuencia de mensajes a todas las réplicas.
- **Reconfiguración en caso de fallos:** en caso de fallo se una réplica, no es necesario mucho trabajo, con tan solo eliminar las referencias a dicha réplica de los clientes que traten de acceder al servicio sirve.

Sobre la replicación pasiva, plantea varias ventajas:

1. Reconfiguración sencilla en caso de fallos.
2. No es necesario implementar dos tipos de réplicas, dado que todas ejecutan el mismo software.

Disponibilidad en Sistemas a Gran Escala

En sistemas a gran escala, las particiones ocurren. Por el teorema CAP sabemos que se debe sacrificar disponibilidad o consistencia. Muchos sistemas de hoy en día a gran escala se diseñan para proporcionar alta disponibilidad reduciendo la consistencia. La disponibilidad resulta imprescindible para la mayoría de sistemas.

Antes o después, cuando la partición desaparezca, las diferentes réplicas convergerán. Se deberán ejecutar algoritmos de convergencia de estado en las réplicas al desaparecer la partición.

Escalabilidad

Los sistemas distribuidos deben diseñarse para que sigan proporcionando servicio conforme crezcan. Decimos que un sistema es escalable si el servicio que ofrece no sufre alteraciones de rendimiento y disponibilidad desde el punto de vista del usuario al aumentar el número de: *usuarios, recursos, nodos, peticiones simultáneas, etc.* Hay que evitar los cuellos de botella.

Cada sistema puede tener limitaciones plausibles a la escalabilidad, que dependen de cada sistema.

Un sistema altamente escalable es un sistema con objetivos de crecimiento a escala global.

TÉCNICAS

Las más importantes para aumentar la escalabilidad implican aumentar la distribución eliminando la centralización.

- **Distribuir la carga:** distribuir el procesamiento que realiza el servicio a diferentes nodos.
- **Distribuir los datos:** distribuir los recursos en diferentes nodos, de forma que cada nodo sirva una parte de los recursos del sistema.
- **Replicación:** replicar los recursos para permitir que cada réplica atienda parte del total de peticiones sobre el mismo recurso. Se emplean balanceadores de carga para repartir la carga entre las diferentes réplicas.
- **Caching:** caso particular de la replicación, donde hay una copia del recurso en el propio cliente.

REPLICACIÓN

Es esencial para aumentar la escalabilidad y muy eficaz en sistemas donde la mayoría de las peticiones son de sólo lectura. En sistemas donde hay operaciones de modificación de los datos, se suele sacrificar la consistencia debido a que se pretende una alta disponibilidad y pueden existir particiones.

En sistemas donde hay mayoría de operaciones de escritura y se pretende consistencia fuerte, la replicación no escala bien. Se deben mantener mutuamente consistentes las réplicas y resulta necesario emplear algoritmos que bloquean el acceso al recurso replicado, mientras se realizan los cambios.

En sistemas que demanden consistencia fuerte, se suele optar por un minucioso particionado de los datos.

CACHING

Es un caso particular de la replicación, donde el mismo cliente mantiene una réplica. Consisten en recordar las últimas versiones de la información accedida en cada componente de una aplicación distribuida, simulando a una caché. Para accesos repetidos sobre un mismo elemento, se obtiene el valor de la copia guardada localmente, sin necesidad de acceder al servicio remoto.

Esta réplica tendrá consistencia débil con respecto al servicio y, por tanto, será adecuado para aquellos servicios que no requieran consistencia fuerte y para recursos de solo lectura.

SEGURIDAD

Todo SD debe ofrecer un servicio disponible y correcto a los usuarios, garantizando:

- **Autenticación:** los usuarios y las peticiones que estos realizan deben identificarse.
- **Integridad:** debe mantener los datos sin modificaciones maliciosas o no autorizadas.
- **Confidencialidad:** solo los usuarios autenticados y autorizados pueden acceder a determinados recursos.
- **Disponibilidad:** el servicio no debe sufrir interrupciones en todo el programa o en parte de él.

TEMA 8 – COMUNICACIÓN

CARACTERÍSTICAS DE LOS MECANISMOS DE COMUNICACIÓN

Permiten la comunicación entre procesos que se ejecutan en ordenadores distintos y son ofrecidos por los middlewares de comunicación. Sus principales características a destacar son: *utilización, estructura y contenido de los mensajes, direccionamiento, sincronía y persistencia.*

Estas características son generalmente útiles para categorizar y comprender los diferentes mecanismos de comunicación existentes. No obstante, la realidad es compleja, y en algunos casos hay mecanismos de comunicación en los que una caracterización precisa no es posible.

UTILIZACIÓN

Con primitivas básicas de comunicación tenemos operaciones de envío y recepción. Un ejemplo: *sockets y colas de mensajes.*

Mediante construcciones del lenguaje de programación, conseguimos un mayor nivel de abstracción y un envío y recepción de mensajes transparente al programar.

Estructura y Contenido de los Mensajes

La estructura de los mensajes puede tener tres formas:

- **No estructurados:** solo contenido que estará en un formato libre (Sockets).
- **Estructurados en cabecera + contenido:** la cabecera es un conjunto de campos, generalmente extensible. Contenido en formato libre (Colas de mensajes).
- **Estructura transparente al programador:** determinada por el middleware de comunicaciones (RPC/ROI).

Por otra parte, en el contenido de los mensajes tenemos:

- **Bytes:**
  - **Ventajas:** eficiente e información compacta.
  - **Inconvenientes:** difícil de procesar, la representación en binario no es igual entre arquitecturas y lenguajes de programación.
- **Texto:**
  - **Ventajas:** independiente de la arquitectura y lenguaje de programación.
  - **Inconvenientes:** menos eficiente.

Normalmente se emplea algún lenguaje con amplia disponibilidad de bibliotecas para su procesamiento, como por ejemplo: *Extensible Markup Language (XML), JavaScript Object Notation (JSON).*

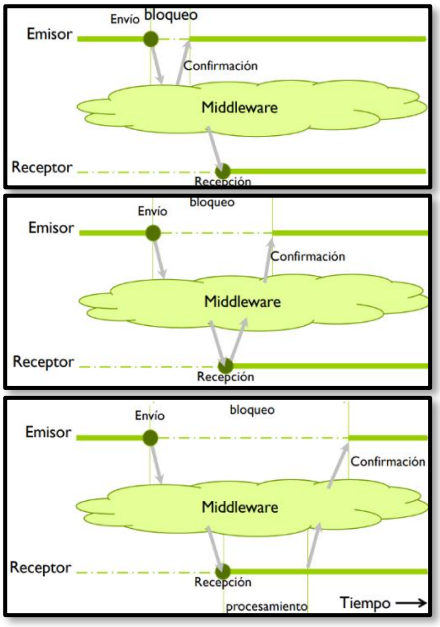
Direccionamiento

- **Direccionamiento directo:** el ordenador emisor envía los mensajes directamente al ordenador receptor (Sockets, servicios web, RPC/ROI).
- **Direccionamiento indirecto:** el ordenador emisor envía los mensajes a un intermediario (bróker), que los hace llegar al receptor (Colas de mensajes).

Sincronización

Tenemos tres tipos de sincronización:

- **Comunicación asíncrona:** el middleware responde al emisor con la confirmación, tras almacenarlo en sus buffers. (UDP, Colas de mensajes)
- **Comunicación síncrona (entrega):** el middleware responde cuando el receptor ha confirmado la entrega correcta del mensaje. (TCP, Colas de mensajes)
- **Comunicación síncrona (respuesta):** el middleware responde al emisor tras recibir el aviso del receptor de haber procesado el mensaje. (RPC/ROI)



Persistencia

Tenemos dos tipos de comunicación según el tipo de persistencia:

- **Comunicación persistente:** el middleware puede guardar los mensajes pendientes de entrega. El receptor puede no estar activo cuando se envía el mensaje, y el emisor puede terminar su ejecución antes de que el mensaje se entregue.
- **Comunicación no persistente:** el middleware no es capaz de mantener los mensajes que deben transmitirse. Se requieren el emisor y el receptor activos para que los mensajes lleguen a transmitirse (Sockets, RPC/ROI).

Invocación a Objeto Remoto (ROI)

Un objeto remoto es aquel que puede ser invocado desde otros espacios de direcciones. Se instancia en un nodo servidor y puede responder a la invocación de clientes locales y remotos.

Las aplicaciones se organizan en colecciones dinámicas de objetos que pueden estar en diferentes nodos, de forma que las aplicaciones y los objetos pueden invocar métodos de otros objetos de forma remota.

VENTAJAS DE LOS OBJETOS REMOTOS

- **Aprovechar la expresividad:** capacidad de abstracción y flexibilidad del paradigma orientado a objetos.
- **Transparencia de ubicación:** la sintaxis de invocación de los métodos de un objeto no depende del espacio de direcciones en el que reside dicho objeto. Podemos ubicar los objetos de acuerdo a distintos criterios: *localidad de acceso, restricciones administrativas, seguridad, etc.*
- **Aplicaciones heredadas:** podemos reutilizar aplicaciones heredadas (*legacy*) encapsulándolas en objetos remotos, haciendo uso del patrón de diseñó *wrapper.*
- **Escalabilidad:** los objetos pueden distribuirse sobre una red, teniendo en cuenta la demanda actual.

TIPOS DE INVOCACIONES

- **Invocador local:** los objetos invocador e invocado residen en el mismo proceso.
- **Invocación remota (ROI):** los objetos invocador e invocado residen en procesos diferentes, dentro del mismo nodo o en nodos distintos.

ANTECEDENTES (RPC – REMOTE PROCEDURE CALL)

RPC es el antecedente de ROI, y comparten los mismos objetivos. No contempla el concepto de objeto, y el ordenador remoto ofrece un catálogo de procedimientos. Estos procedimientos pueden ser llamados de forma transparente (como si fueran locales) desde ordenadores cliente. El mecanismo es implementado por los *"stub"* cliente y servidor.

El procedimiento es el siguiente:

1. Invocación del procedimiento local.
2. Empaquetado de argumentos de entrada *(marshalling)* en un mensaje.
3. Envío del mensaje al servidor y espera de respuesta.
4. Desempaquetado del mensaje y extracción de argumentos de entrada.
5. Llamada al procedimiento.
6. Ejecución del procedimiento.
7. Retorno de control al stub servidor.
8. Empaquetado de argumentos de salida y resultado en un mensaje.
9. Envío del mensaje de respuesta.
10. Desempaquetado del mensaje y extracción de argumentos de salida y resultado.
11. Retorno de control al código que invocó el procedimiento local.

El procedimiento local es invocado, manda el mensaje con lo que le haga falta al servidor, el cual llama al procedimiento, lo ejecuta y le devuelve la respuesta al procedimiento local. Enmascara invocación a objeto remoto como invocación local. Similar a RPC en cuanto a la transparencia de ubicación, pero aplicado al modelo de objetos.

ELEMENTOS QUE INTERVIENEN EN UNA ROI

- **Proxy:** representa al objeto remoto. Ofrece la misma interfaz que el objeto remoto. Contiene una referencia al objeto remoto y proporciona acceso al objeto remoto y a su interfaz. Se crea en tiempo de ejecución cuando se accede por primera vez al objeto remoto.
- **Esqueleto:** recibe las peticiones de los clientes. Realiza las verdaderas llamadas a los métodos del objeto remoto. Se crea en tiempo de ejecución cuando se crea el objeto remoto.
- **Object Request Broker (ORB):** componente principal de un middleware orientado a objetos. Se encarga de:
    1. Identificar y localizar los objetos.
    2. Realizar las invocaciones remotas de los proxies a los esqueletos.
    3. Gestionar el ciclo de vida de los objetos (creación, registro, activación y eliminación).

PASOS DE UNA ROI

El cliente invoca el método en el proxy, y este se encarga de empaquetar los argumentos. Llama al ORB y gestiona la invocación, el esqueleto invoca al método real, al finalizar libera al esqueleto. Este, llama al ORB empaquetando los argumentos y el proxy hace que le lleguen los resultados al cliente.

1. El proceso cliente invoca el método del proxy local relacionado con el objeto remoto.
2. El proxy empaqueta los argumentos y, utilizando la referencia al objeto, llama al ORB. Este gestiona la invocación, haciendo que el mensaje llegue al esqueleto.
3. El esqueleto desempaqueta los argumentos e invoca al método solicitado, quedando a la espera de que finalice.
4. El método llamado finaliza y se desbloquea al objeto.
5. El esqueleto empaqueta los resultados y llama al ORB, el cual hace llegar el mensaje al proxy.
6. El proxy desempaqueta los resultados y los devuelve al proceso cliente.

Paso de Objetos como Argumentos

Tenemos dos formas de hacerlo:

- **Paso por valor:** el estado del objeto origen se empaqueta, mediante un proceso denominado *serialización.* El objeto serializado se transmite al nodo destino, donde se crea un nuevo objeto copia del original. Ambos objetos evolucionan por separado.
- **Paso por referencia:** se copia la referencia del nodo invocador al nodo invocado. No importa que el objeto pertenezca al nodo invocador, o que pertenezca a un tercer nodo. Todas las modificaciones se hacen sobre el mismo objeto.

Creación de Objetos ROI

La creación de objetos, y su registro en el ORB, puede realizarse con dos procedimientos:

- **Por iniciativa del cliente:** el cliente solicita a una factoría crear el objeto. La factoría crea el objeto solicitado y lo registra en el ORB, obteniendo una referencia al objeto y su esqueleto. Tras esto, devuelve una copia de su referencia. La *factoría* es un objeto servidor que crea objetos de un determinado tipo.
- **Por iniciativa del servidor:** un proceso servidor crea un objeto y lo registra en el ORB, obteniendo una referencia al objeto. El proceso servidor registra la referencia en el servidor de nombres, para que otros clientes puedan buscarlo.

Java RMI (Remote Method Invocation)

Es un middleware de comunicación orientado a objetos que proporciona una solución para un lenguaje orientado a objetos específico (Java) que da soporte a la portabilidad.

No es multilenguaje, pero es multiplataforma. Permite invocar métodos de objetos Java de otra JVM, y pasar objetos Java como argumentos cuando se invocan dichos métodos. Puede hacer lo siguiente:

- **Incorporación:** el componente RMI se incorpora de forma automática a un proceso Java cuando se utiliza su API. Escucha peticiones que llegan en un puerto TCP.
- **Objeto invocable:** un objeto es invocable de forma remota si implementa una interfaz que extiende a "`Remote`".
- **Invocar a un objeto:** para invocar a un objeto remoto desde otro proceso se utiliza un objeto proxy. Su interfaz es idéntica a la del objeto remoto.

Este contiene una referencia al objeto remoto, compuesta por su dirección IP, el puerto y qué objeto permite localizar al esqueleto del objeto remoto.

El Servidor de Nombres de Java RMI

El servidor de nombres almacena, para cada objeto: nombre simbólico + referencia. Puede residir en cualquier nodo y es accedido desde el cliente o el servidor usando la interfaz local llamada "`Registry`". En las distribuciones Oracle de Java, el servidor de nombres se lanza usando la orden `rmiregistry`.

- **Bind:** registrar un objeto con un nombre, si ya existe da una excepción.
- **Rebind:** igual que bind, pero si ya existía solo le cambia el nombre.
- **Unbind:** elimina el registro del objeto asociado al nombre.

Desarrollo de una Aplicación Java RMI

Reglas para Programar Objetos Remotos en Java

Primero, estudiaremos la interfaz de un objeto remoto:

- Debe extender la interfaz `java.rmi.Remote`.
- Los métodos de dicha interfaz deben indicar que puede generarse la excepción `RemoteException`.
- A partir de la definición de la interfaz, el compilador de Java genera proxies y esqueletos.

```
Public interface Hola extends Remote{
          String Saluda() throws RemoteException;
}
```

La clase de los objetos remotos debe implementar la interfaz remota, (En el caso del ejemplo, la interfaz "Hola") y extender `java.rmi.server.UnicastRemoteObject`. Esto permite registrar los objetos en el ORB de Java.

```
class ImplHola extends UnicastRemoteObject implements Hola{
          ImplHola() throws RemoteException {...} //Constructor
          public String Saluda() throws RemoteException{
                    return "Hola a todos";
          }
}
```

Aplicación Servidor y Cliente

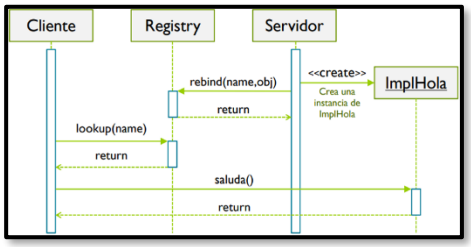
Servidor

```
Registry reg = LocateRegistry.getRegistry(host, port);
reg.rebind("objetoHola", new ImplHola());
System.out.println("Servidor Hola preparado");
```

Cliente

```
Registry reg = LocateRegistry.getRegistry(host, port);
Hola h = (Hola) reg.lookup("objetoHola"); //Casting
System.out.println(h.saluda());
```

Características



Cuando se invoca un método, podemos pasar objetos como argumentos:

- **Implementa la interfaz Remote:** se pasa por referencia. El objeto es compartido por las referencias previas y la nueva.
- **No implementa la interfaz Remote:** se serializa y se pasa por valor. Se crea un objeto copia en la máquina virtual destino totalmente independiente al original.

Los objetos creados y devueltos deben ser del tipo "Remote".

Por otra parte, la comunicación se caracteriza por lo siguiente:

- **Utilización:** las llamadas a métodos remotos hacen transparente al programador el uso de las primitivas básicas de comunicación.
- **Estructura y contenido de los mensajes:** determinados por el compilador de Java, transparente al programador.
- **Direccionamiento:** directo al ordenador donde reside el objeto remoto.
- **Sincronización:** sincrónica en la respuesta. Se espera a que el método remoto termine.
- **Persistencia:** no persistente. El objeto remoto debe estar activo.

Servicios Web

Son sistemas software diseñador para proporcionar interacciones ordenador-ordenador utilizando la red (W3C). Normalmente están basados en una arquitectura cliente-servidor implementada sobre el protocolo HTTP.

Tenemos las siguientes variantes:

- **Servicios web basados en SOAP y WSDL:** son generalmente conocidos simplemente como "servicios web". SOAP (Simple Object Access Protocol) es una especificación XML de la información que se intercambia, y WSDL (Web Services Description Language) es especificación XML que describe la funcionalidad ofrecida por un servicio web.
- **Servicios web RESTful:** es una alternativa más simple y flexible, con más aceptación en la actualidad. Destacamos JSON (JavaScript Object Notation) por ser el formato más habitual para el intercambio de información, ya que no requiere ningún lenguaje de descripción de funcionalidad. No obstante, empiezan a emplearse, siendo OAS (Open API Specification) el más utilizado actualmente.

Como alternativa, podemos utilizar servicios web mediante la construcción y el procesado directo del contenido de los mensajes HTTP. Esto permite la generación automática de código, proporcionada por frameworks de desarrollo y despliegue de servicios web, tanto para el lado cliente como el lado servidor, a partir de la definición del servicio.

Dado el mecanismo de petición y respuesta, y la generación automática de código, los servicios web podrían considerarse como una forma de RPC.

SERVICIO WEB RESTFUL

Son muy importantes para el desarrollo de aplicaciones web. Están disponibles en la inmensa mayoría de lenguajes de programación y frameworks de desarrollo. Además, son sencillos.

Tiene convenios de uso, los cuales no son unos estándares, sino más bien un conjunto de convenciones de uso:

- **Datos y funcionalidad:** se consideran recursos, se accede a ellos mediante URIs.
- **Actuación sobre recursos:** se actúa sobre recursos utilizando operaciones simples y bien definidas (HTTP).
- **Cliente-servidor:** se usa la arquitectura cliente-servidor.
- **Sin estado:** diseñado para utilizar un protocolo de comunicación sin estado. Están basados en Caching: un cliente puede almacenar información para no tener que consultar constantemente un servidor.
- **REST + Servicios Web:** se les llama servicios web RESTful.

La identificación de recursos se realiza a través de las URIs, dado que proporcionan un espacio de dirección global para el descubrimiento de recursos y servicios.

CARACTERÍSTICAS DE LA COMUNICACIÓN EN REST

REST es un mecanismo de difícil categorización en función de la manera en que se analice:

- Desde el punto de vista del mecanismo subyacente, sus características serían aquellas asociadas a la comunicación basada en el protocolo HTTP y en el uso de sockets TCP.
- En cambio, desde el punto de vista del uso, es básicamente un mecanismo de petición-respuesta, similar en muchos aspectos a RPC y ROI.
- **Utilización:**
  - **Enfoque 1:** primitivas básicas de envío y recepción (API de Sockets).
  - **Enfoque 2:** API de alto nivel proporcionada por el proveedor del servicio.
- **Estructura y contenido de los mensajes:**
  - **Enfoque 1:** contenido codificado con HTTP, XML o JSON.
  - **Enfoque 2:** contenido oculto por la API del proveedor.
- **Direccionamiento:** directo mediante peticiones al ordenador que alberga el servicio.
- **Sincronización:**
  - **Enfoque 1:** sincrónica en la entrega.
  - **Enfoque 2:** sincrónica en la respuesta.
- **Persistencia:** no persistente. El servidor tiene que estar activo cuando hagamos la petición.

MIDDLEWARES ORIENTADOS A LA MENSAJERÍA

Son middlewares que ofrecen comunicación basada en mensajes. Los emisores envían mensajes no directamente al receptor, sino a un elemento intermedio denominado generalmente cola. (Direccionamiento indirecto).

Una vez depositado el mensaje en la cola, el emisor prosigue con su ejecución, sin esperar a que el receptor recoja el mensaje. Este recoge el mensaje en cualquier momento. La comunicación es asíncrona.

La mayoría de estos sistemas están basados en un bróker de comunicaciones, donde un proceso gestiona todas las colas.

- Crea y borra los elementos de esta atendiendo a los envíos y las peticiones de los emisores y receptores.
- Mantiene los mensajes en las colas pendientes de entrega, aunque emisores y receptores no estén en ejecución, incluso ni siquiera el propio bróker.

Presenta las siguientes ventajas e inconvenientes:

Ventajas	Inconvenientes
• Permiten componentes del sistema distribuido altamente desacoplados. • Pueden ofrecer un alto grado de disponibilidad mediante réplicas activas de brókeres. • Es posible conseguir un alto grado de seguridad al ser sistemas centralizados.	• Menor rendimiento al introducir el bróker como intermediario para comunicarse. • Difícil escalabilidad por la existencia del bróker. • Falta de estandarización, dado que existen protocolos abiertos, pero muchas de las implementaciones más utilizadas son propietarias.

JAVA MESSAGE SERVICE 2.0 (JMS)

Es una API Java que permite a las aplicaciones enviar y recibir mensajes. Con direccionamiento a través de proveedor, sincronización asíncrona y persistente.

Ofrece una comunicación débilmente acoplada:

- El emisor envía mensajes a un intermediario "destino", y el receptor recibe mensajes de dicho "destino".
- Emisor y receptor no necesitan conocerse entre sí, solo deben estar de acuerdo con el formato del contenido de los mensajes.

Se debe utilizar JMS cuando:

- No se quiere que los componentes de una aplicación dependan de conocer las interfaces de otros componentes.
- No es necesario que todos los componentes estén simultáneamente en ejecución.
- Los componentes, tras enviar un mensaje, no necesitan recibir una respuesta inmediata para poder continuar operando con normalidad.

COMPONENTES DE JMS

- **Proveedores JMS:** sistema de mensajería que implementa las interfaces de JMS y proporciona herramientas administrativas y de control. Recibe y envía mensajes de los clientes.

- **Clientes JMS:** programa o componente escrito en Java que produce o consume mensajes.

- **Mensajes:** objetos que mandan información entre clientes, con tipo y escritura definida.
  - **Cabecera:** campos fijos, definidos por JMS.
  - **Propiedades:** extensión de la cabecera. Campos definidos por el programa.
  - **Cuerpo tipo:** vacío, texto, bytes, objeto serializado, diccionario, etc.

- **Objetivos administrativos:** factorías de conexiones y destinos (Colas y Temas).
  - **Factorías de conexiones:** creadas mediante las herramientas administrativas del proveedor JMS, y se utilizan para crear las conexiones de los clientes al sistema de mensajería.
  - **Destinos:** Creados mediante las herramientas administrativas del proveedor JMS.
    - **Colas:** entrega a un solo cliente.
    - **Temas:** entrega a múltiples clientes, difusión.

- **ZeroMQ:** no requiere la existencia del bróker.

JMS COMO MODELO DE PROGRAMACIÓN

Tenemos los siguientes elementos en el modelo de programación JMS:

- **`interface ConnectionFactory:`** los objetos que la implementan vinculan la aplicación con un objeto y crean conexiones con el proveedor JMS.

- **`interface JMSContext:`** los objetos que la implementan:
  - Mantienen una conexión con el proveedor JMS y son utilizables por un solo hilo.
  - Procesan envíos y recepciones de forma secuencial.
  - Una aplicación puede usar varios objetos JMSContext si requiere concurrencia.

- **`interface JMSProducer:`** los objetos que la implementan permiten enviar mensajes a colas y temas..

- **`interface JMSConsumer:`** los objetos que la implementan permiten recibir mensajes de colas y temas.

- **`interface Destination:`** los objetos que la implementan vinculan la aplicación con un objeto administrado. Encapsulan una dirección específica del proveedor JMS.
  - **Subinterfaces:** `Queue, Topic.`

- **`interface Message:`** los objetos que la implementan son los mensajes que se envían y reciben en JMS
  - **Subinterfaces:** `TextMessage, BytesMessage, ObjectMessage, etc.`

TEMA 9 – ALGORITMOS DISTRIBUIDOS

CONCEPTOS

Son independientes de tecnologías particulares, y las características de los algoritmos distribuidos descentralizados son:

- Ningún nodo posee toda la información completa del sistema.
- Se ejecutan en procesadores independientes. El fallo de un nodo no impide que el algoritmo progrese.
- Toman decisiones basadas en información local.
- No hay una fuente precisa de tiempo global.

ESTADOS GLOBALES Y TIEMPO

Hay una necesidad de sincronización de relojes físicos. Cada nodo "i" dispone de un reloj local "Ci" que representa un valor de tiempo universal coordinado (ETC). Dado cualquier instante real "t", el objetivo es que en todos los nodos Ci (t) = t. Todos los relojes locales tengan la misma hora.

El problema radica en los chips de reloj de los relojes Ci no son absolutamente exactos. Dependiendo del tipo, tienen una precisión de entre 10^{-6} y 10^{-13}.

ALGORITMO DE SINCRONIZACIÓN DE RELOJES FÍSICOS – ALGORITMO DE CRISTIAN

Hay un nodo que tiene una hora la cual consideramos la exacta. El objetivo es que todos los nodos se sincronicen con esa hora. Los relojes nunca deben retroceder, pero se pueden parar para sincronizarse. Los pasos son los siguientes:

1. Cliente pide valor del reloj al servidor en el instante T0 (según Cc).
2. Servidor contesta con valor de su reloj Cs.
3. Respuesta llega a cliente en T12 (Según Cc).
4. Cliente calcula $C = C_s + \frac{T_1 - T_0}{2}$
5. Cliente actualiza su reloj según:
    a. Si C > Cc, se fija el valor del reloj del cliente a C (Cc = C).
    b. Si C < Cc, se detiene Cc las siguientes Cc – C unidades de tiempo.

Se almacena LAG = Cc – C, y se descartarán ticks del reloj del cliente hasta transcurrido el tiempo LAG. Se evita que retroceda el reloj.

Existe un inconveniente obvio, el cual de debe a la posibilidad de que caiga el nodo central. Depende del medio por el que se envían los mensajes, puede que no todos estén sincronizados.

ALGORITMO DE SINCRONIZACIÓN DE RELOJES FÍSICOS – ALGORITMO DE BERKELEY

Aquí no tenemos un reloj muy preciso, por lo que se opta por otra opción. Se debe sincronizar todos los relojes entre sí, de manera local.

Un nodo manda a todos su hora actual y deja que los demás le contesten con su hora. De esa hora, calcula las diferencias con la suya, le hace la media y le reenvía a cada uno cuánto se tiene que desviar para que todos vayan a la par. Las operaciones se hacen teniendo en cuenta el tiempo que tarda en ir y volver el mensaje.

Si una hora es muy distinta, se descarta. Esto se hace de manera periódica para que las desviaciones que pueden tener los relojes no alteren el funcionamiento óptimo del sistema.

- **Ventajas:** si el servidor falla, se inicia el algoritmo en otro servidor. Se inicia un algoritmo de elección de líder. En principio, cualquiera puede ser el líder, mientras que en el algoritmo de Cristian solo uno puede serlo.
- **Desventajas:** no se tiene sincronización con el exterior, solo a nivel local de los dispositivos que estén conectados.

RELOJES LÓGICOS

Indican el orden en que suceden ciertos eventos, no el instante real en que suceden. Son útiles para muchos tipos de aplicaciones distribuidas que solo requieren saber si un evento ha sucedido antes que otro. A diferencia de los relojes físicos, su sincronización es perfecta, pero tienen ciertas limitaciones.

- Si dos nodos no interactúan, es decir, si no intercambian mensajes, no es necesario que tengan el mismo valor de reloj.
- Es importante el orden global en que ocurren los eventos, y no el tiempo real.

Para sincronizar los relojes lógicos, se define la relación "ocurre-antes". Tiene las siguientes características:

- **Observación:** se puede observar de forma directa.
  - Si a y b son eventos del mismo nodo, y a ocurre antes que b: a → b.
  - Si a es el evento de envío de un mismo mensaje por parte de un nodo, y b el evento de recepción de ese mensaje por parte de otro nodo, a → b.
- **Es transitiva:** es una relación transitiva. Si a → b, y b → c, entonces a → c.
- **Eventos simultáneos:** dos eventos "x" e "y" son concurrentes si no se puede decir cuál de ellos ocurre antes (x || y).

Relojes Lógicos de Lamport

Es necesaria una forma de medir la relación ocurre-antes: usando el reloj lógico de Lamport.

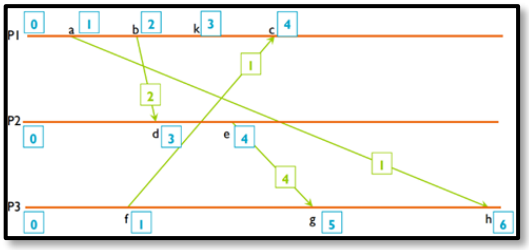
Los relojes lógicos deben marcar el instante en que ocurren los eventos, de forma que asocien un valor a cada evento. Llamamos C(a) al valor del reloj lógico del evento a.

Los relojes lógicos deben satisfacer: Si a → b, entonces C(a) < C(b).

Algoritmo de Lamport

Cada nodo dispone de un contador (reloj lógico) CP inicializado a 0.

1. Cada ejecución de un evento (envío de mensaje o evento interno) en un nodo p, incrementa el valor de su contador CP en 1.
2. Cada mensaje m enviado por un nodo p es etiquetado (Cm) con el valor de su contador CP, es decir, Cm = CP.
3. Cuando un nodo p recibe un mensaje, actualiza su reloj: CP = max(CP, Cm) + 1.



Es como decir que los relojes cuentan eventos en vez de segundos, preguntando el número de eventos tienes el instante en el que está.

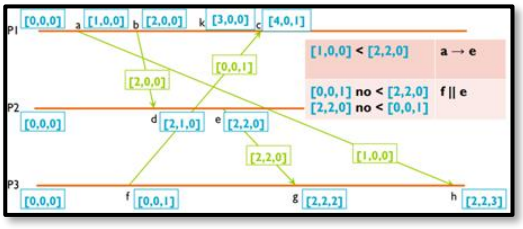
Establece un orden parcial entre los eventos. Los eventos concurrentes no guardan orden entre sí. Este orden puede convertirse en total añadiendo el identificador del nodo como sufijo.

- Si a → b, el algoritmo garantiza que C(a) < C(b).
- Si C(a) < C(b), no se puede decir si a → b, o a || b.

Relojes Vectoriales

No se puede saber cuando 2 eventos son concurrentes. Pero, puede ser que nos haga falta. Se puede solucionar haciendo uso de los relojes vectoriales.

Los relojes vectoriales nos permitirán saber si un evento ocurre antes que otro, o si los dos eventos son concurrentes. Estos asocian un valor vectorial V(x) a cada evento x.



Cada nodo tiene un vector con N espacios, siendo N el número de nodos del sistema.

- Vp[p] es el número de eventos que han ocurrido en p.
- Si Vp[q] = k, entonces p sabe que, al menos, han ocurrido k eventos en el nodo q.

Cada nodo p dispone de un reloj vectorial Vp, inicializado con todos sus elementos a 0.

1. El nodo p incrementa Vp[p] en 1 cada vez que envía un mensaje o ejecuta un evento interno.
2. Un mensaje m enviado por un nodo p lleva asociado su reloj vectorial. Es decir, Vm = Vp.
3. Al recibir p un mensaje m, incrementa Vp[p] en 1, y además actualiza su reloj seleccionando el máximo entre el valor local y el valor del reloj del mensaje, para cada una de sus componentes, siendo una componente por cada nodo distribuido:

$$\forall i, 1 \leq i \leq N, \quad Vp[i] = \max(Vp[i], Vm[i])$$

V(a) será menor que V(b) si:

- Cada una de las componentes de V(a) es menor o igual que las de V(b).
- Además, debe haber al menos una de ellas que sea estrictamente menor.

Estado Global

El estado global de un sistema distribuido es el conjunto de estados de cada nodo (se presta atención solo en las variables de interés), y los mensajes enviados y todavía no entregados. Ejemplos de uso:

- **Recolector de objetos remotos no utilizados:** se averigua si algún nodo mantiene alguna referencia al objeto remoto, y en si en algún mensaje en tránsito hay alguna referencia a dicho objeto. Si no se encuentra, el objeto se elimina.
- **Detección de deadlock distribuido:** detecta si el sistema está en un estado de bloqueo mutuo.
- **Detección de terminación distribuida:** detecta si ha terminado el algoritmo distribuido. Todos los procesos pueden estar detenidos, pero puede haber un mensaje en camino.

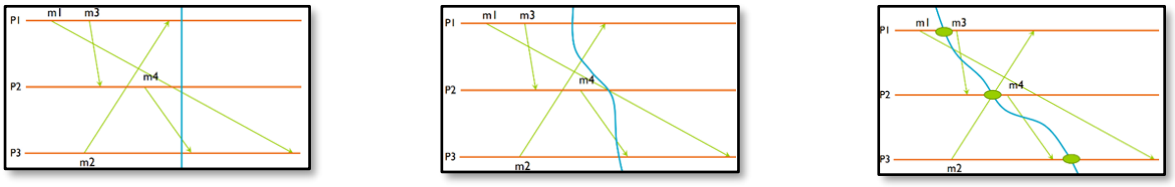
La instantánea distribuida refleja el estado que pudo haberse alcanzado en el sistema. Las instantáneas deben reflejar sólo estados consistentes.

Instantáneas

Tenemos tres tipos:

- **Instantánea precisa:** la instantánea precisa no es factible, haría falta que los nodos tuviesen sus relojes perfectamente sincronizados, y eso no es posible.
- **Instantánea consistente:** no se ha reflejado ninguna entrega de mensaje que no esté precedida por su respectivo envío. No es perfecta, pero por lo menos tiene sentido.
- **Instantánea inconsistente:** se ha registrado la entrega de algunos mensajes, pero no su envío.

Precisa **Consistente** **Inconsistente**



Algoritmo de Chandy-Lamport

Crea una instantánea consistente del estado global del sistema distribuido. Se asume:

- Sistema distribuido formado por varios nodos.
- Red con topología completa. Existe un canal entre todo par de nodos.
- Los canales son fiables, transmiten sus mensajes en orden FIFO y son unidireccionales, es decir, que entre dos nodos "p" y "q", existen dos canales: (p,q) y (q,p).

Su procedimiento es el siguiente:

1. El nodo iniciador p guarda su estado local, y envía un mensaje MARCA al resto de nodos.
2. Cuando un nodo p recibe el mensaje MARCA por el canal c:
    a. **Si aún no ha registrado su estado:** registra su estado local y anota como vacío el canal c. Luego, pero antes de enviar cualquier otro mensaje, envía MARCA al resto de nodos y comienza a registrar los mensajes que lleguen por otros canales distintos a c.
    b. **Si ya había registrado su estado:** anota todos los mensajes que ha recibido por el canal c (puede no haber ninguno), y deja de registrar la actividad en dicho canal.
3. Cuando un nodo p recibe MARCA por todos sus canales de entrada, envía su estado previamente guardado y el de sus canales de entrada al nodo iniciador y finaliza su participación en el algoritmo.

Los mensajes de MARCA tienen que tener el identificador del nodo que lo ha empezado todo.

Exclusión Mutua Distribuida

Tenemos el mismo problema de siempre, pero ahora con varios procesos, acceso a la sección crítica de un recurso compartido por parte de distintos procesos en nodos diferentes. Para solucionar este problema, se han de evitar inconsistencias, y las soluciones necesitan asegurar el acceso con exclusión mutua de los procesos.

Se tienen que dar estas condiciones para garantizar la exclusión mutua:

- **Seguridad:** como mucho, un solo proceso puede ejecutarse dentro de la sección crítica en todo momento.
- **Viveza:** todo proceso que quiera entrar en la sección crítica lo conseguirá en algún momento:
    - **Progreso:** si la sección crítica está libre, y hay procesos que desean entrar, se selecciona en tiempo finito a uno de ellos.
    - **Espera limitada:** si un proceso quiere entrar en sección crítica, solo debe esperar un número finito de veces a que otros entren antes que él.

Tenemos tres soluciones para garantizar la exclusión mutua garantizada:

Algoritmo Centralizado

Se elige un nodo como coordinador. Este líder gobierna el acceso a la sección crítica, y:

- Cuando un nodo quiere entrar en la sección crítica, envía un mensaje SOLICITAR al líder, pidiendo permiso.
- Si ningún otro nodo está en la sección crítica, el líder responde CONCEDER. El líder anota que la sección crítica está ahora ocupada mientas el nodo solicitante la usa.
- Si la sección crítica está ocupada, y otro nodo solicita al líder utilizarla, El líder anota la identidad del solicitante y no le contesta. El nodo solicitante permanece suspendido, esperando la respuesta.
- Cuando un nodo sale de su sección crítica, avisa al líder con un mensaje LIBERAR.
    - Si existe algún nodo suspendido esperando permiso para entrar en la sección crítica, el líder le envía CONCEDER.
    - En caso de haber varios esperando se elige uno según algún criterio (FIFO).
    - Si no existe ningún nodo suspendido el líder anota vuelve a estar libre.

Algoritmo Distribuido

Suponemos que todos los eventos están ordenados. Se puede hacer por ejemplo usando relojes lógicos de Lamport junto al número de nodo. En este caso, no hay un nodo líder.

- Cuando un nodo quiere entrar en la sección crítica, difunde un mensaje TRY a todos.
- Cuando un nodo recibe un mensaje TRY:
    - Si no está en su sección crítica, ni esperaba entrar, responde OK.
    - Si está en su sección crítica, no contesta y encola el mensaje.
    - Si no está en su sección crítica, pero quiere entrar, compara el número de evento del mensaje entrante con el que él mismo envió al resto. Vence el número más bajo: Si el mensaje entrante es más bajo, responde OK. Si es más alto, no responde y encola el mensaje.
- Un nodo entra en la sección crítica, cuando recibe OK de todos los demás nodos.
- Cuando un nodo abandona la sección crítica, envía OK a todos los nodos que enviaron los mensajes que este ha metido en su cola, o sea los que estaban esperando.

Algoritmo para Anillos

No existe ningún nodo coordinador. La coordinación se resuelve mediante comunicación en anillo, y el uso de un token que circula por dicho anillo. Requiere comunicación fiable. Si un nodo cae, hay que reconstruir el token.

Aquí solo puede entrar en la sección crítica el nodo que tiene el token.

Al inicio, la sección crítica está libre y el token se encuentra en alguno de los nodos. El algoritmo es el siguiente:

1. Si el nodo no espera entrar en su sección crítica, pasa el token al siguiente nodo.
2. Un nodo espera para entrar en su sección crítica si no tiene el token.
3. Un nodo entra en su sección crítica cuando consigue el token. No lo pasa al siguiente nodo hasta que finaliza su sección crítica.

Elección de Líder

En muchos algoritmos distribuidos resulta ventajoso que un nodo actúe como líder o coordinador ya que los simplifica y reduce el número de mensajes necesario para acordar algo.

La elección se produce al inicio o cuando se detecta que el líder deja de responder. Se asume que los nodos conocen los identificadores del resto de nodos del sistema distribuido y que la elección de líder es un caso particular de consenso.

Algoritmo Bully

Sirve para que, a iniciativa de uno de los nodos, se elija un nuevo líder. El nuevo líder será el nodo activo con el identificador más alto. Tras ser elegido, el líder lo notificará al resto.

1. Cuando un nodo quiere comenzar una elección, envía ELECCIÓN a todos los nodos con identificador mayor al suyo.
2. Cuando un nodo recibe ELECCIÓN, envía OK a quien se lo envió para avisarle que participa y que ganará.
3. Cuando un nodo recibe al menos un mensaje OK, deja de participar.
4. Los pasos anteriores se repiten hasta que un nodo no obtiene respuesta, él es el nuevo líder.
5. El nuevo líder difundirá un mensaje COORDINADOR para comunicarlo al resto.

Algoritmo para Anillos

Los nodos están dispuestos en un anillo lógico y envían sus mensajes a través de los canales de dicho anillo. Este algoritmo sirve para que, a iniciativa de uno de los nodos, se elija un nuevo líder y, tras ser elegido, se propagará la identidad del líder por el anillo.

1. Cuando un nodo quiere comenzar una elección de líder construye un mensaje ELECCIÓN con dos campos: iniciador y lista de nodos participantes. Asigna su identificador a ambos campos y envía el mensaje al siguiente nodo del anillo.
2. Cuando un nodo recibe un mensaje ELECCIÓN, si no es el iniciador responde al nodo emisor con un mensaje OK. Se le incluye en la lista de nodos participantes y envía el mensaje al siguiente nodo. Si no llega confirmación (OK), lo envía al siguiente del nodo que no responde.
3. Cuando un nodo recibe un mensaje ELECCIÓN, y es el iniciador, selecciona como líder el nodo de la lista de participantes con mayor identificador, construye un mensaje COORDINADOR. Nuevo líder y propaga dicho mensaje por el anillo.

Los mensajes de COORDINADOR son opcionales. Estos mensajes pueden mandarse a través de los anillos, o en modo *broadcast* a todos.

Consenso

El acuerdo que deben alcanzar los nodos del valor de una variable. Este valor debe de ser propuesto por un nodo de los activos, no valen redondeos, aproximaciones, medias, etc.

Las condiciones de corrección de consenso son:

- **Terminación (Viveza):** todo nodo correcto, tarde o temprano, decide algún valor.
- **Integridad uniforme (Seguridad):** todo nodo decide como máximo una vez.
- **Acuerdo (Seguridad):** ningún par de nodos correctos decide de manera diferente.
- **Validez uniforme (Seguridad):** si se decide v, entonces v fue propuesto por algún nodo.

Algoritmos de Consenso – Algoritmos sin Fallos

Suponemos que la red es totalmente conexa, de los N nodos algunos están apagados/han fallado. No pueden fallar ni reactivarse durante el algoritmo. En caso de que sea un fallo se interpretarán como fallos de parada, los bizantinos no se consideran.

La detección de fallos se hace mediante TimeOuts, se mandan mensajes y si no se contesta al cierto tiempo se considera fallado.

Algoritmos en Presencia de Fallos

Suponemos que la red es totalmente conexa, de los N nodos algunos están apagados/han fallado. Sí pueden fallar o reactivarse durante el algoritmo. En caso de que sea un fallo se interpretarán como fallos de parada.

La detección de fallos se hace mediante TimeOuts, se mandan mensajes y si no se contesta al cierto tiempo se considera fallado. Estos se supone que están bien ajustados Un nodo valido detectará a otro nodo valido en algún momento.

Los nodos están numerados y ordenados, del 0 al N-1. Estos pueden estar funcionando o estar "apagados", o fallar durante el algoritmo. Se toleran $\left\lfloor \frac{(N-1)}{2} \right\rfloor$ nodos fallidos (Redondeo hacia abajo). Solo funciona si hay como mínimo $\left\lceil \frac{(N+1)}{2} \right\rceil$ (Redondeo hacia arriba). Si fallan más nodos, el algoritmo no funciona. Los nodos coordinadores se bloquean.

FUNCIONAMIENTO GENERAL

Se basa en que los nodos vayan ejecutando rondas hasta que cada uno emita "decisión(V)".

En cada ronda se elige un coordinador de esta manera: Si es ronda 0 el 0 es el coordinador, si es ronda 1 el 1 es coordinador... En una ronda solo hay 1 coordinador, pero puede ser que no cambien de ronda exactamente al mismo tiempo, por lo que en un momento dado puede haber varios coordinadores. Pero solo uno por ronda.

Debe tener las siguientes variables:

- Ronda actual (r).
- Coordinador Actual (NC).
- Valor que el nodo propone a la variable (lastEstimate).
- Ronda más reciente que me hizo cambiar la propuesta (lastR).

ALGORITMO

Todos los nodos inician r = 0, NC = 0, lastR = 0, lastEstimate = (valor que proponga el nodo).

Por cada ronda "r", se debe seguir estos pasos:

1. Todos los nodos envían al coordinador de la ronda NC: "estimate(r, lastEstimate, lastR). Esperan como respuesta un mensaje "propose(V)". Si excede el tiempo de respuesta máximo, su detector de fallos creerá que el coordinador ha fallado.
2. El coordinador espera a recibir $\left\lceil \frac{(N+1)}{2} \right\rceil$ mensajes "estimate". No usa TimeOuts. Si no llegan al menos ese número de mensajes, se quedará esperando "bloqueado".
3. Los nodos ordinarios esperan a recibir "propose(r, proposeR)" del coordinador, o les vence el TimeOut:
    a. Si reciben el mensaje, contestan con "ACK(r)" a NC y actualizan lastEstimate = proposeR, lastR = r.
    b. Si vence su TimeOut, contesta con NACK(r) a NC.
4. El coordinador espera respuestas ACK o NACK de los nodos ordinarios. Espera $\left\lceil \frac{(N+1)}{2} \right\rceil$ mensajes. El coordinador también se manda a sí mismo, por lo que se cuenta a sí mismo también. Si recibe esa cantidad de mensajes ACK, difunde el "decide(lastEstimate)", y genera "decision(lastEstimate)". Este será el valor final.

Debemos conocer los siguientes conceptos:

- **Nodo pasivo:** todos los nodos que han generado "decision" siguen participando en el algoritmo como nodos pasivos. Solo responden a los propose si el valor es el mismo que el suyo.
- **Cambios de ronda:**
    - **Coordinador:** tras la fase 4, si no recibe el número suficiente de ACKs, incrementa la ronda y ya no será coordinador. Puede recibir tanto NACK como ACK, cuando llegue al mínimo de mensajes. Si no todos son ACKs, entonces cambia de ronda.
    - **Ordinario:** tras la fase 3, los nodos ordinarios incrementan la ronda y se convierten en el nuevo coordinador si les corresponde.
- **Finalización eventual:** el algoritmo termina cuando todos los nodos correctos están en modo pasivo. Todo nodo correcto habrá generado la misma decisión.