RESUMEN TEORÍA SISTEMAS DISTRIBUIDOS Y CONCURRENTES

ÍNDICE DE CONTENIDOS:

[] TEMA 1: INTRODUCCIÓN

} 1.1 ¿QUÉ ES UN SISTEMA DISTRIBUIDO?

Sistema Distribuido: Conjunto de sistemas o nodos autónomos (dispositivos hardware o procesos software) capaces de ejecutarse independientemente actuando de forma totalmente transparente como un único sistema para el usuario.
Idea principal: Compartir servicios y recursos (datos, hardware, aplicaciones software, servicios de localización y robots), eliminar cuellos de botella y reducir los puntos centrales de error del sistema.

} 1.2 EJEMPLOS DE SISTEMAS DISTRIBUIDOS

Ejemplos: Redes de telefonía (antenas, torres de comunicación, cables, satélites, conmutadores y orquestación), Internet (www, red de pares P2P), aplicaciones (Gmail, Google Cloud), GPS (usado en infinidad de dispositivos, cuyo objetivo es obtener la geolocalización mediante 32 satélites con un reloj atómico orbitando a 20000 km de altura a una velocidad de 14000 km/h), OctaPi (cluster de computación basado en Raspberry Pi), Robocup (algoritmos distribuidos donde cada robot puede tener programados varios nodos), Smart Cities (IoT y 5G, los sistemas distribuidos engloban comunicaciones entre satélites, aviones, coches y dispositivos WiFi).

} 1.3 VENTAJAS DE LOS SISTEMAS DISTRIBUIDOS

Ventajas: Económicas (es más barato comprar muchos ordenadores pequeños que uno grande), fiabilidad (el sistema puede seguir funcionando aunque algunos nodos fallen), crecimiento incremental (se pueden añadir/actualizar recursos bajo demanda), distribución inherente (aplicaciones dependientes de varias máquinas separadas por sus propias características).

} 1.4 TRANSPARENCIA (DEFINICIÓN Y TIPOS)

Transparencia de Distribución: Esconde el hecho de que el sistema está formado por distintos componentes. Se consigue una transparencia total si se obtiene la ilusión de tener un único sistema (single-system image).
Transparencia de Acceso: Esconde los detalles sobre diferencias en la representación de los datos y sus mecanismos de acceso (orden de bytes, codificación de texto plano y acceso a una interfaz homogénea de datos sin dependencia de su procedencia).
Transparencia de Localización: Esconde los detalles sobre la localización de los recursos mediante la asignación de nombres lógicos (números de teléfono, paths y enlaces).
Transparencia de Migración/Relocalización: Esconde al usuario si el recurso se mueve de un componente a otro. Si esto ocurre mientras el recurso está en uso se le llama relocalización (sistemas de ficheros en montajes de volumen, satélites GPS de visión directa en localización y reasignación de la torre de comunicaciones al hablar por teléfono).
Transparencia de Replicación: Esconde que haya réplicas del mismo recurso, por lo que se requiere transparencia de localización (al actualizar una tabla de bases de datos se actualizan N réplicas, donde las peticiones van a la réplica del servicio que esté más cercana a la red).
Transparencia de Concurrencia: Esconde los componentes del sistema que deben compartir ciertos objetos y cooperar para proporcionar el recurso conservando un estado coherente (no es necesario reintentar la operación si la base de datos está siendo usada por otro componente en ese momento).
Transparencia de Fallo: Esconde si ciertos componentes del sistema han fallado, aunque enmascarar los fallos es complejo y no siempre es posible ni apropiado (redirige la petición a otro servidor en caso de fallo, aunque el cliente recibe el resultado sin retornar error).
Grado de transparencia deseable: Compromiso entre transparencia y eficiencia mediante restricciones físicas, donde no siempre conviene ofrecer un tipo de transparencia (transparencia de localización y de fallo).

} 1.5 SISTEMA ABIERTO

Sistema Abierto: Los recursos sirven de una forma estándar para proporcionar interoperabilidad, portabilidad y extensibilidad (descripción de la interfaz del recurso, formato y semántica de los mensajes, convenios del sistema y separación de políticas y mecanismos).

} 1.6 ESCALABILIDAD (DEFINICIÓN Y TIPOS)

Escalabilidad: Soporte de crecimiento, distribución geográfica y administración; replicación (servicio centralizado que tiende a convertirse en un cuello de botella que reduce la escalabilidad del sistema); distribución (servicio centralizado que estará lejos de algunos clientes si éstos están dispersos geográficamente); un servicio centralizado es más sencillo de implementar sin subestimar la simplicidad del sistema mientras que un servicio distribuido es más difícil de asegurar.
Algoritmo centralizado: Los nodos tienen que recolectar todos los datos sobre el estado del sistema para ejecutar el algoritmo (esto puede no ser asumible).
Algoritmo distribuido: Ningún nodo tiene la información completa del sistema; pueden tomar decisiones en base a su estado; el fallo de uno de ellos no arruina el algoritmo; y no asumen un reloj común exactamente sincronizado (consenso, commit atómico de transacciones, ordenación de eventos, elección de líder, exclusión mutua distribuida).
Escalabilidad geográfica: El problema principal es la latencia, cuyo límite es fijo entre nodos lejanos (300000 km/s), las comunicaciones son menos fiables a larga distancia.
Técnicas de escalabilidad: Comunicación asíncrona en servicios no interactivos; batching (agrupación de operaciones); protocolos con pocos round-trips; preprocesado en el cliente; caching (coherencia de cache).

} 1.7 EDGE COMPUTING Y FOG COMPUTING

Edge: Propone que los datos sean analizados por los propios sensores o por componentes que se encuentran cerca de los propios sensores evitando el paso por la nube y proporcionando la ventaja de analizar datos en tiempo real (dispositivos de funcionamiento pasivo en los que los datos recogidos se envíen por largos recorridos para llegar a centros de datos y nubes de computación).
Edge Computing: Basado en cómo los procesos computacionales se realizan en los dispositivos de IoT que generan la información.
Fog Computing: Estructura de red descentralizada en la que recursos, datos y aplicaciones se sitúan en algún lugar lógico entre el Cloud y la fuente que genera los datos, donde la inteligencia se lleva al nivel de red del área local de la arquitectura de red.

} 1.8 MIDDLEWARE DISTRIBUIDO

Middleware Distribuido: Sistema software que permite la comunicación entre procesos abstrayendo toda la complejidad de comunicaciones, localización, hardware, etc (ONC RPC Sun, CORBA, Java RMI, ICE ZeroC, DDS).

[] TEMA 2: COMUNICACIÓN ENTRE PROCESOS DISTRIBUIDOS

} 2.1 COMUNICACIÓN ENTRE PROCESOS

Los procesos que se ejecutan dentro de la misma máquina pueden interactuar entre ellos mediante espacios de memoria compartidos (variables, buffers y ficheros) y paso de mensajes (pipes y colas), aunque la comunicación entre sus nodos debe ser distribuida, algo que complica la interacción entre los procesos.
Marshaling: Proceso de transformación de la representación de datos a un formato adecuado para su almacenamiento o transmisión usado en comunicaciones distribuidas (procesos de diferentes arquitecturas con diferentes representaciones de datos) que ofrecen una transparencia de acceso.
Sockets: Permiten la comunicación entre procesos distribuidos a bajo nivel. Se necesita conocer los detalles de la red y tener el control completo de las comunicaciones, aunque tienen difícil extender el sistema si se quieren soportar nuevas arquitecturas.
RPC (Remote Procedure Call): Permite realizar llamadas a funciones localizadas en otra máquina; encapsula toda la programación de red y comunicaciones; ofrece transparencia de acceso; todas sus complejidades se encapsulan en stubs; permiten modular la lógica del sistema distribuido con una escalabilidad excelente.
RMI (Remote Method Invocation): Paradigma de orientación a objetos aplicado a comunicación distribuida; permite modificar el estado de objetos remotos; separa entre interfaces de comunicación y objetos; está compuesto por un proxy (stub cliente) y un skeleton (stub servidor); y ofrece modularización y escalabilidad.

} 2.2 PROCESO, THREAD Y FORK

Proceso/Tarea: Abstracción del SO para ejecutar un conjunto de código que no comparte memoria física (siempre tienen un ID -> PID).
Thread/Hilo/Subproceso: Subflujo de ejecución que puede ser manejado de manera independiente por un planificador/scheduler (normalmente son parte de un proceso y suelen compartir espacios de direcciones y memoria).
fork(): Genera un proceso hijo con una copia exacta del padre, donde cada uno ejecuta independientemente en un espacio diferente de memoria.

} 2.3 ARQUITECTURAS DE COMPUTACIÓN

Cliente/Servidor: Dos roles en la interacción, el servidor hace de cuello de botella, y utilizan comunicación síncrona.
Editor/Suscriptor: Maneja eventos/topics generados por editores y recibidos por suscriptores, y utilizan comunicación asíncrona.
Peer-To-Peer: Poseen una arquitectura descentralizada y redes sin una tipología definida.

} 2.4 SINCRONIZACIÓN ENTRE EMISOR Y RECEPTOR

Comunicación síncrona: Se realiza una comunicación y se espera activamente hasta obtener el resultado esperado (fork()).
Comunicación asíncrona: Se realiza una comunicación y se atienden otras tareas hasta que la respuesta está lista (low battery).

} 2.5 PASO DE MENSAJES

Paso de mensajes: Paradigma de programación/Capa de protocolo de transporte que se usa para invocar de forma abstracta un comportamiento concreto por parte de otro actor, todo ello desde un proceso.
Características: Alto nivel de encapsulamiento y distribución, mantienen coherencia y sincronización entre los procesos distribuidos, y la comunicación se realiza mediante primitivas de comunicación (send y receive).

} 2.6 PASO DE MENSAJES (COMUNICACIÓN)

Comunicación directa: Cada proceso que desea comunicarse debe nombrar explícitamente el destinatario o el remitente de la comunicación, donde los dos procesos tienen que conocerse previamente y solo existe un único enlace de comunicación (send/receive(proceso,mensaje)).
Comunicación indirecta: Los mensajes se envían y reciben de buzones, permiten todo tipo de esquemas (muchos a uno -> puertos), y dos o más procesos pueden utilizar el buzón (send/receive(buzón,mensaje)).
Comunicación simétrica: Tanto emisor como receptor necesitan nombrarse entre sí para comunicarse (send/receive(proceso/mensaje)).
Comunicación asimétrica: Sólo el emisor nombra al destinatario resolviendo el problema en aplicaciones cliente/servidor (send/receive(proceso/ID,mensaje)).
Llamadas bloqueantes: La comunicación se bloquea hasta que el paso de mensaje se haya realizado (utilizadas en comunicación síncrona).
Llamadas no bloqueantes: La comunicación no bloquea al componente que la solicita. En el envío, el emisor delega el envío a un subcomponente y reanuda su ejecución, mientras que en la recepción, si existe un dato disponible, el receptor lo lee, y en caso contrario se notifica que no hay datos.

} 2.7 PASO DE MENSAJES (CANAL, MENSAJE Y ERRORES)

Canal: Medio de comunicación por el cual emisor y receptor pueden interaccionar, puede ser unidireccional (televisión, radio) o bidireccional (conversación telefónica, chat).
Características: Flujo de datos con enlaces unidireccionales o bidireccionales; gran rango de capacidad; los mensajes pueden ser de tamaño fijo o variable; canales con o sin tipo; y pueden tener paso por copia o por referencia.
Mensaje: Tanto emisor como receptor deben conocer la codificación correcta (tamaño, ordenación y formato).
Errores: Son más probables en transmisiones entre procesos distribuidos; se pierden mensajes en procesos que sólo se comunican utilizando un canal; los ruidos de la transmisión pueden alterar el mensaje original; bloqueo del emisor o del receptor; y todo esto se engloba en el ámbito de Sistemas Tolerantes a Fallos.

[] TEMA 3: SOCKETS

} 3.1 INTRODUCCIÓN A SOCKETS

Socket: Mecanismo que nos permite realizar comunicación entre procesos a través de una red proporcionando una interfaz entre aplicación y red. En UNIX pueden ser entendidos como descriptores de fichero, siempre se comunican en pares y son usados para comunicar procesos entre sí tanto local como remotamente.

} 3.2 TIPOS DE SOCKETS

SOCK_STREAM (características): TCP, entrega confiable, orden garantizado, orientado a conexión y bidireccional.
SOCK_STREAM (comunicación): Se crean sockets tanto en cliente como en servidor, el servidor prepara el socket y escucha, el cliente genera el suyo y se conecta, el servidor acepta la conexión y ambos intercambian mensajes, y finalmente ambos cierran los sockets.
SOCK_DGRAM (características): UDP, entrega no segura, no se garantiza el orden, no es orientado a conexión y bidireccional.
SOCK_DGRAM (comunicación): Se crean sockets tanto en el cliente como en el servidor, el servidor prepara el socket, ambos intercambian mensajes, y finalmente ambos cierran los sockets.

} 3.3 ESTRUCTURAS DE DATOS

struct in_addr {}: Define en IPV4 la dirección IP representada en 4 bytes (big-endian en la red y little-endian en el host).
struct sockaddr_in {}: Define detalles de la conexión como el tipo de familia del socket, el puerto de la conexión y su dirección IP.
INADDR_ANY: Se usa para escuchar en cualquier interfaz del servidor.

} 3.4 CREACIÓN DE UN SOCKET

int socket (int d, int t, int p): Crea un canal de comunicación especificando dominio d, tipo t y protocolo de comunicación p.
int bind (int s, const struct sockaddr *a, socklen_t al): Asigna una dirección específica al socket, especificándole un descriptor s, la estructura con información de la IP y puerto a, y la longitud al de la estructura anterior.
int listen (int s, int b): Convierte un socket definido por un descriptor en un socket pasivo y puede ser utilizado para aceptar conexiones entrantes, especificándole un descriptor de socket s y un tamaño máximo de conexiones encoladas por el kernel b.
int accept (int s, struct sockaddr *a, socklen_t *al): Extrae la primera petición de conexión de la cola de conexiones pendientes y crea un nuevo socket para el cliente, especificándole un descriptor s, la estructura con información de la IP y puerto a, y la longitud al de la estructura anterior.
int close (int s): Marca el socket como cerrado (el descriptor s no puede ser usado por el proceso).
ssize_t recv (int s, void *b, size_t l, int f): Recibe datos de un socket, especificándole un descriptor s, un buffer b donde se guarda la información, la longitud l de la estructura anterior, y el número de flags f que extienden su funcionalidad. Esta función es bloqueante, por lo que se queda esperando indefinidamente hasta que el socket tenga información que leer (utilizar MSG_DONTWAIT).
ssize_t recvfrom (int s, void *b, size_t l, int f, struct sockaddr *sa, socklen_t *al): Recibe datos de un socket con una dirección en concreto, especificándole un descriptor s, un buffer b donde se guarda la información, la longitud l de la estructura anterior, el número de flags f que extienden su funcionalidad, la estructura sa con la información de la dirección origen, y la longitud al de la estructura anterior.
ssize_t send (int s, void *b, size_t l, int f): Envía datos por un socket, especificándole un descriptor s, un buffer b donde se guarda la información, la longitud l de la estructura anterior, y el número de flags f que extienden su funcionalidad.
ssize_t sendto (int s, void *b, size_t l, int f, struct sockaddr *sa, socklen_t *al): Manda datos por un socket a una dirección en concreto, especificándole un descriptor s, un buffer b donde se guarda la información, la longitud l de la estructura anterior, el número de flags f que extienden su funcionalidad, la estructura sa con la información de la dirección origen, y la longitud al de la estructura anterior.

} 3.5 LLAMADAS, MÚLTIPLES CONEXIONES Y BUFFERING

select(): Monitoriza descriptores de ficheros hasta que estén listos para realizar ciertas operaciones de I/O.
Manejo de múltiples conexiones: Inicia el servidor (bind/listen/accept). Cuando se acepta la conexión, se crea un thread para atender, interactuar y comunicarse con cada cliente.
setbuf (stdout, NULL): Deshabilita el buffering a la hora de imprimir mensajes.

} 3.6 LIBRERÍA PTHREAD

#include <pthread.h>: Librería que implementa el standard POSIX.
pthread_create (): Crea el thread y lo pone en ejecución.
pthread_exit (): Finaliza el thread, puede pasar parámetros opcionales recogidos en pthread_join().
pthread_join (): Realiza una espera hasta que un thread determinado ha finalizado.

[] TEMA 4: SINCRONIZACIÓN Y RELOJES

} 4.1 RELOJES Y SINCRONIZACIÓN EN ROBÓTICA

Relojes en Sistemas Distribuidos: No existe un único reloj hardware común, ya que cada nodo tiene su propio reloj local. Es necesaria una sincronización para desarrollar aplicaciones en tiempo real y ordenar de forma natural eventos distribuidos.
Tipos de sincronización: Relativa (nodos del sistema) y absoluta (realidad).
Ejemplos de sincronización en Robótica: Distributed Data Logging, SLAM, láser de un robot, conducción autónoma.

} 4.2 CONCEPTO DE TIEMPO

Tiempo (concepto): Antiguamente se basaban en la observación y posición de las estrellas y los planetas para contar el tiempo.
Calendarios a lo largo de la historia: Monumento monolítico de Escocia (8000 aC), división en 12 ciclos lunares de los sumerios (3000 aC), división en 12 meses de 30 días de Egipto (1000 aC), calendario Juliano de Julio César (46 aC), calendario gregoriano del papa Gregorio XIII (1582).
Periodo de rotación (1900): Un segundo equivalía a 1/86400 del día solar medio.
Tiempo de Efemérides (TE): Basado en el movimiento orbital de los astros y de los planetas sin tener en cuenta la rotación de la Tierra.
GMT (Greenwich Mean Time): Tiempo solar medio en el observatorio de Greenwich.
Reloj atómico (1948): Alimenta su contador utilizando una frecuencia de resonancia atómica normal (reloj de Cesio 133, donde 1 segundo equivale al tiempo en el que el Cesio 133 realizaba 9192631770 transiciones).

} 4.3 UTC

UTC (Coordinated Universal System): Principal estándar de tiempo que regula relojes y el tiempo actualmente. Se obtiene a partir del TAI (Tiempo Atómico Internacional) además de que se actualiza con el tiempo medio de Greenwich (se añade o quita 1 segundo a finales de Julio y de Diciembre). Este sistema de tiempo es utilizado por muchos estándares de Internet y la World Wide Web.
UTC (problemas): En el tiempo de UNIX/POSIX se guarda el tiempo como un número de segundos a partir de un tiempo de referencia (1/1/1970), no tiene noción de los segundos intercalares.

} 4.4 SINCRONIZACIÓN

Resolución: Mínima magnitud que un sistema puede medir.
Precisión: Dispersión del conjunto de valores obtenidos de mediciones en nuestro sistema.
Exactitud: Cuánto de cerca del valor real se encuentran las medidas del sistema.
Sincronización (problemas): No se puede garantizar que un conjunto de relojes esté siempre sincronizado, ya que no todos avanzan al mismo ritmo por razones técnicas y/o físicas, donde aparecen el offset (diferencia con el reloj de referencia) y el drift (velocidades y frecuencias diferentes en cada reloj).
Sincronización de relojes en Sistemas Distribuidos: Garantiza que los procesos se ejecuten de forma cronológica, y a la vez respetando el orden de los eventos dentro del sistema.

} 4.5 RELOJES FÍSICOS

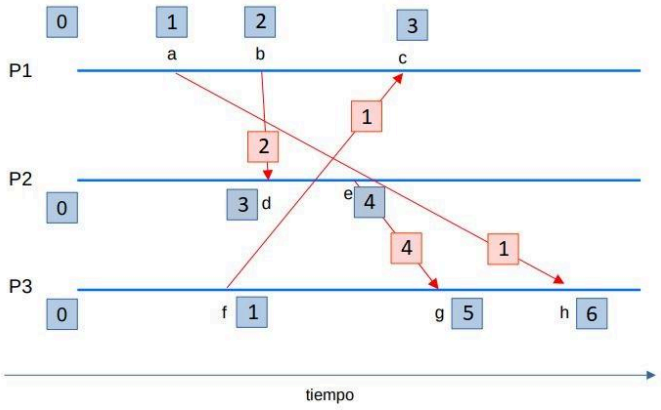
Relojes físicos: Hardware que tiene noción del tiempo real.
NTP (Network Time Protocol): Sistema síncrono que depende de relojes o unidades máster, donde existe una jerarquía de relojes (Cesio, rubidio de raíz, etc).
NTP (funcionamiento): El cliente se conecta al servidor; el servidor contesta con una marca de tiempo NTP; los mensajes tardan tiempo en enviarse y llegar; por lo que el cliente debe compensar el RTT (round-trip time), donde RTT = T.petición + T.espera (Tc) + T.respuesta.
systemd-timesyncd: Demonio Linux encargado de actualizar la hora del sistema mediante NTP.
GPS: Sistema de satélites con un reloj atómico que orbitan la Tierra a 20000 km de altura a 14000 km/h.
Teoría de la relatividad espacial: Al observar desde la Tierra, el tiempo medido por los satélites parece que transcurre más lento (atrasos de 7 microsegundos al día).
Teoría de la relatividad general: La curvatura espacio-tiempo depende del campo gravitacional al que es sometido (adelantos de 45 microsegundos al día).

} 4.6 RELOJES LÓGICOS

Relojes lógicos: Indican el orden en que suceden ciertos eventos, no el instante real en el que suceden. Son útiles en sistemas distribuidos donde la noción del orden de los eventos es importante. Su sincronización es perfecta pero posee limitaciones.
Relojes lógicos (aplicaciones): Mensajes periódicos de sincronización, ordenación de eventos y detección de violaciones de causalidad.

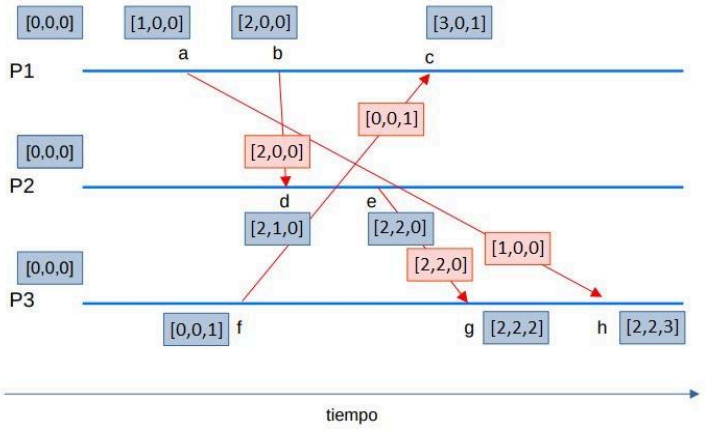
} 4.7 RELOJES DE LAMPORT

Relojes de Lamport (1978): Basados en la relación transitiva “ocurre antes” a -> b, donde a y b son eventos del mismo nodo donde a ocurre antes que b, y también si a es el elemento de envío del proceso P1 y b es el elemento de recepción del proceso P2.
x || y: Dos eventos x e y son concurrentes si no se puede decir con seguridad qué evento ocurre antes, estableciendo un orden parcial entre los elementos de un sistema distribuido, ya que no todos los eventos están relacionados entre sí.
Proceso P: Posee una variable LCp inicializada a 0 que se actualiza (+1) cuando el proceso genera un envío, y que incluye el valor del reloj una vez se ha enviado el mensaje.
Proceso Q: Cuando recibe un mensaje con un tiempo t, actualiza su reloj en base a LCq = max (LCq, t) + 1.
a -> b: El algoritmo garantiza que C(a) < C(b).
C(a) < C(b): El algoritmo no garantiza que a -> b y a || b.



} 4.8 RELOJES VECTORIALES

Relojes vectoriales: Asocian un valor vectorial a cada evento (envío y/o recepción de mensajes) que se produce en un sistema distribuido. En un sistema de N nodos, los relojes vectoriales guardan un vector de N relojes lógicos en cada nodo (solucionan algunas limitaciones de los relojes de Lamport).
Nodo P: Se inicializa a 0 (Vp = [0,0,0,...,N]) e incrementa en 1 antes de que ocurra un evento, donde cada mensaje enviado lleva asociado un reloj vectorial Vm [p] ($\forall i, 1 \leqslant i \leqslant N$ Vp[i] = max(Vp[i],Vm[i])).
V(a) < V(b): Todas las componentes (una estrictamente) de V(a) son menores o iguales que las de V(b).
a -> b: El algoritmo garantiza que V(a) < V(b).
V(a) < V(b): El algoritmo garantiza que a -> b.
a || b: El algoritmo no garantiza que V(a) < V(b) ni V(b) < V(a).
Relojes vectoriales (problemas): Presentan algunas limitaciones, el tamaño de los mensajes se incrementa debido al vector de relojes, y se pueden optimizar únicamente mandando los valores que han cambiado.



[] TEMA 5: CONCURRENCIA AVANZADA

} 5.1 ESPERA ACTIVA

Espera activa: Técnica donde un proceso está repetidamente comprobando una condición, ya sea una pulsación de teclado o una habilitación de sección crítica, entre otros. Es una estrategia válida para sincronizar procesos y sistemas con múltiples procesadores (SMP), por lo que esta técnica debe evitarse siempre que sea posible para evitar un uso innecesario de la CPU.

} 5.2 MUTEX Y SECCIÓN CRÍTICA

Mutex: Mecanismos usados en programación concurrente para evitar que entre más de un proceso a la vez en la sección crítica.
Sección crítica: Fragmento de código donde puede modificarse un recurso compartido, algo que si se realiza concurrentemente sin protección puede conllevar resultados indeterminados (operaciones lock() y unlock()).

} 5.3 SEMÁFOROS

Semáforos: Se utilizan para solucionar problemas de sincronización sin espera activa, ya que los procesos pueden bloquearse o ejecutar estando condicionados únicamente por el valor que tiene una variable entera.
init(): Establece el valor inicial del contador del semáforo.
wait(P): Decrementa (lock) el contador del semáforo; si éste valor es positivo, decrementa el contador y finaliza; y si el valor del contador es 0, la llamada se bloquea (wait, acquire).
signal(V): Incrementa (unlock) el contador del semáforo, y si después de actualizar el contador hay algún proceso que está bloqueado en el wait(), éste será despertado para realizar el lock() sobre el semáforo (signal, post, release).
Semáforo compartido: pshared=0 (threads del mismo proceso), pshared=1 (threads de diferentes procesos).
Exclusión mutua: Se reduce la implementación a la inicialización del semáforo con el valor 1. Si entra un proceso, se decrementará el contador y se ejecutará la sección crítica mediante wait(), y después se ejecutará signal(). Y si entra un segundo proceso, el contador valdrá 0 al ejecutar el wait() y se bloqueará hasta que el otro proceso ejecute signal().
Sincronización: El valor del contador del semáforo equivale al número de recursos que utilizará el sistema, es decir, si se utilizan N procesadores, se limita el acceso a la sección crítica a N procesos.
Uso inadecuado de semáforos: Puede dejar varios procesos/hilos bloqueados, y si el único proceso/hilo que puede despertar a los demás se queda bloqueado, podría no despertar nunca al resto.
Interbloqueo: Situación en la que dos hilos se quedan bloqueados esperando a que otro lo despierte.

} 5.4 VARIABLES DE CONDICIÓN

Variables de condición: Ofrecen una extensión sobre el comportamiento de los mutex ofreciendo respuesta ante determinados eventos, además de que éstos nos permiten bloquear el hilo que se está ejecutando y liberar el mutex para que otro proceso pueda continuar (son una buena solución para evitar realizar espera activa y deben ser llamadas con el lock del mutex asociado).
Deadlock: Ocurre cuando la espera activa genera un bloqueo mutuo.
cond_wait() / cond_signal(): Libera el mutex y deja bloqueada la ejecución esperando el signal de la variable condición (si hay muchos threads bloqueados esperando a la variable condición, cada cond_signal que se ejecute desbloqueará un único thread dependiendo de la política del planificador).
cond_broadcast(): Desbloquea todos los threads bloqueados de la variable condición.

} 5.5 PRODUCTORES / CONSUMIDORES

Enunciado: Es un ejemplo clásico de problema de sincronización de multiprocesos, donde el productor genera y escribe información, mientras que el consumidor recibe y lee información, ambos de un mismo buffer de tamaño finito que comparten.
Problema: El productor no puede añadir más elementos que la capacidad del buffer y el consumidor no puede leer elementos de un buffer que está vacío.
Posibles soluciones: Utilizando mutex + variable de condición; o utilizando dos variables de condición para modelar las acciones de “buffer no lleno” (cond_wait) y “buffer no vacío” (cond_signal), las cuales deben realizar su uso dentro de la región crítica, una vez adquirido el mutex (entre el lock y el unlock).

} 5.6 LECTORES / ESCRITORES

Enunciado: Problema de exclusión mutua donde muchos procesos compiten por entrar en la sección crítica.
Proceso lector: Puede compartir la sección crítica con otros procesos lectores, pero nunca con escritores.
Proceso escritor: Necesita ejecutar con acceso exclusivo y de uno en uno, ya que realizarán operaciones de escritura.
Diferentes enfoques: Prioridad para los lectores o para los escritores.

} 5.7 MONITORES

Monitores: Primitiva con estructura cuya responsabilidad de funcionamiento recae en los módulos del programa.
Características: Proporcionan un mecanismo de abstracción; contienen variables accesibles desde los métodos/funciones, que no pueden ejecutarse si otro método/función se está ejecutando; además, deben asegurar la exclusión mutua de la ejecución de sus métodos/funciones.
Idea: Separar variables y métodos/funciones utilizando cerrojos para la exclusión mutua y variables de condición para restringir la ejecución (librería thread_safe).

} 5.8 BARRERAS Y FUTEX

Barrera: Elemento de sincronización que nos permite esperar a que un número determinado de tareas finalicen, y son útiles para sincronizar el comienzo de sub-tareas dentro del mismo thread implementados en la librería pthread.
pthread_barrier_init (&barrier, …, count): Inicializa la barrera con el número de tareas a esperar.
pthread_barrier_wait (&barrier): Cada tarea espera a que el resto haya terminado.
Futex (Fast Userspace Mutex): Se utiliza en contextos de memoria compartida y en la implementación de pthread_mutex, además de que su implementación es eficiente y rápida en espacio de usuario.

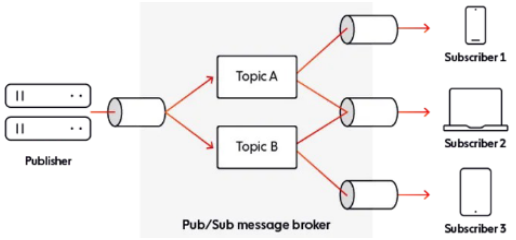
[] TEMA 6: PUBLICADOR/SUSCRIPTOR

} 6.1 INTRODUCCIÓN

Tipos de comunicaciones: Locales (pipes, ficheros, memoria compartida) y globales (sockets).
Tipos de conexiones según agentes: Uno-Uno, Uno-Muchos, Muchos-Uno, Muchos-Muchos.
Problema: Se tiene un sistema masivo de recolección de información de sensores, encargado de sensorizar la temperatura de un edificio grande.
Desventajas: El servidor es el único punto de fallo; tanto el servidor como los sensores necesitan conocer sus endpoints; y por último, ni escala correctamente ni es flexible.
Características interesantes: Mínimo ancho de banda usado; latencia baja (real-time); es fiable incluso en redes no fiables como UDP; y es capaz de desacoplar software y hardware.

} 6.2 PUBLICADOR/SUSCRIPTOR

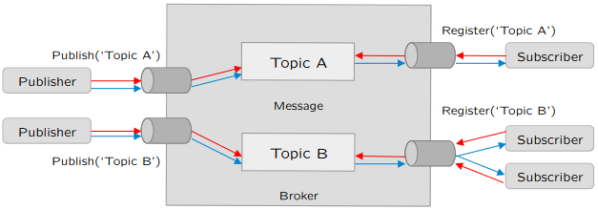
Publicador/Suscriptor: Patrón de diseño utilizado para distribuir mensajes generados por un publicador a todos los suscriptores, comunicándose de forma asíncrona. No hay conexión directa entre publicadores y suscriptores, por lo que ninguno de ellos sabe el número de nodos en el sistema.
Publicador: Genera mensajes con información y eventos asociados a un topic que envía al broker.
Suscriptor: Se conecta al broker y solicita la recepción de mensajes de un topic específico.
Topics: Permiten filtrar mensajes.



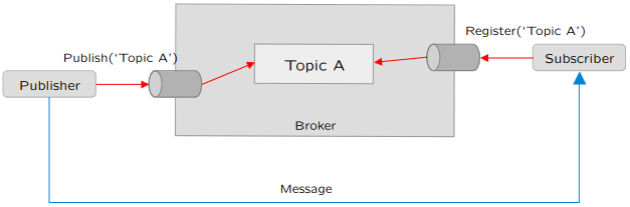
Ventajas: Bajo acoplamiento entre nodos del sistema; gran escalabilidad en el sistema; habilita las topologías dinámicas en la red; es flexible y de diseño limpio.
Desventajas: Versiones centralizadas con un único punto de fallo; broker con cuello de botella; los errores de entrega de mensajes no se notifican ni a publicadores ni a suscriptores.
Usos: Aplicaciones de chat, notificación de eventos y fuentes de datos cuyo origen no es importante.

} 6.3 TIPOS DE PUBLICADOR/SUSCRIPTOR

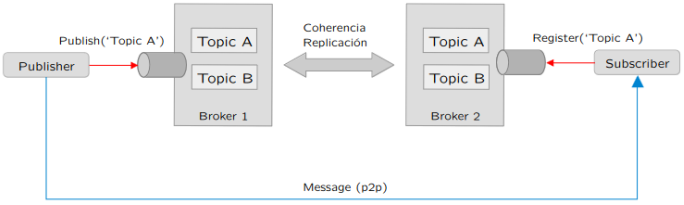
Publicador/Suscriptor Centralizado: Todo el flujo pasa por el único broker del sistema.



Publicador/Suscriptor Centralizado Parcialmente: Sólo el registro y la petición del topic pasan por el broker del sistema, donde los mensajes son directamente intercambiados entre publicador y suscriptor (p2p).

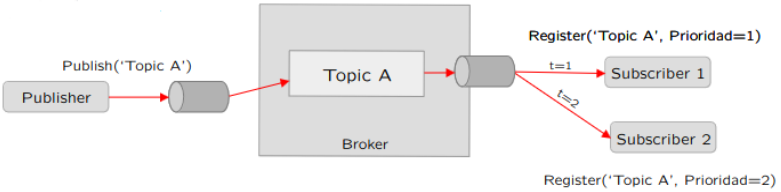


Publicador/Suscriptor Distribuido: El servicio de registro y el descubrimiento de servicios está distribuido, donde los mensajes son directamente intercambiados entre publicador y suscriptor (p2p).



} 6.4 QUALITY OF SERVICE (QOS)

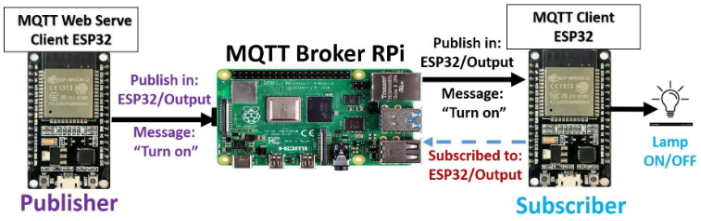
Quality of Service (QoS): Rendimiento del sistema o red desde el punto de vista del usuario final.
Problemas: Congestión de red, fallos en comunicaciones y desconexiones aleatorias que pueden provocar que el mensaje no se entregue correctamente.



Niveles de QoS: Nivel 0 (recibe el mensaje como mucho una vez), Nivel 1 (recibe el mensaje al menos una vez), Nivel 2 (recibe el mensaje exactamente una vez).
Sistemas: Apache Kafka (Uber, Spotify, Shopify), MQTT (IoT), DDS (ROS2 y sistemas industriales), Amazon SNS, Corba, Google Cloud.

} 6.5 MQTT

MQTT: Protocolo de red basado en publicador/suscriptor estándar para comunicaciones IoT (IBM en 1999).
Características: Se ejecuta normalmente sobre TCP/IP; computacionalmente ligero; concepto de broker y clientes; se puede considerar como un publicador/suscriptor centralizado.
Usos: Automoción; fábricas; productos de consumo; transporte, gases y petróleo; IoT y microcontroladores como ESP, Arduino o RaspberryPi (protocolo muy ligero y sin coste computacional).



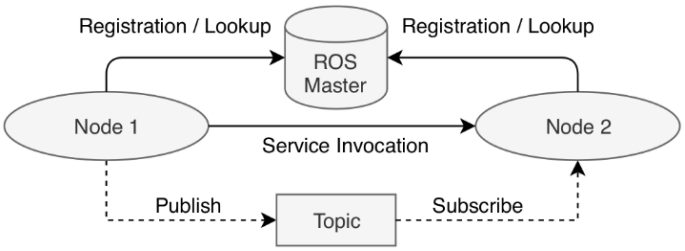
Ventajas: Eficiencia, bajo cómputo y consumo; poco uso de la red para intercambiar mensajes; e implementación sencilla.
Desventajas: Ciclos lentos de transmisiones cuando el sistema está compuesto por más de 200 nodos; poco uso de seguridad y encriptación; escalabilidad.

} 6.6 MOSQUITTO

Mosquitto: Implementación open-source de MQTT que utiliza la arquitectura publicador/suscriptor. Es muy utilizado en sistemas Linux que actúan de broker y pueden comunicarse con nodered.

} 6.7 ROS Y ROS2 (DDS)

ROS: Basado en un publicador/suscriptor centralizado.



Características ROS2 (DDS, Data Distribution Service): Totalmente distribuido; descubrimiento dinámico; real-time; alto rendimiento; seguridad avanzada; y múltiples QoS.

[] TEMA 7: DATA DISTRIBUTION SERVICE (DDS)

} 7.1 INTRODUCCIÓN

Data Distribution Service (DDS): Especificación para un middleware del tipo publicador/suscriptor en computación distribuida capaz de distribuir altos volúmenes de datos.
DDS OMG (Object Management Group): Estándar para tiempo real, confiable y de alto rendimiento para publicación y suscripción. Es el primer estándar internacional middleware que aborda directamente la publicación y suscripción de comunicaciones de tiempo real y sistemas embebidos.

} 7.2 OMG DDS

DDS API: Garantiza la portabilidad del código fuente entre diferentes implementaciones.
DDS Interoperability Wire Protocol: Asegura la interoperabilidad y la intercomunicación entre diferentes implementaciones de DDS.
Aplicaciones: Smart Grids, Smart Cities, Air Traffic Control, High Performance Telemetry, Financial Trading y Large Scale Supervisory System, donde los datos se entregan de forma confiable y predecible con una latencia baja.

} 7.3 DOMINIOS, TOPICS Y TIPOS DE TOPICS

Dominio DDS: Red lógica de aplicaciones que pueden comunicarse entre ellas si pertenecen a dicho dominio, y donde cada dominio se representa mediante un identificador único.
Topic: Representa una unidad de información que puede ser producida o consumida, y que está compuesta por un tipo de datos, un nombre único en GDS y por políticas QoS.
Topic Types: Representan la información que se produce o se consume, normalmente definidos en IDL o xml.
IDL (Interface Description Language): Lenguaje para especificar interfaces utilizados en computación distribuida. Es muy útil para generar proxys/stubs en diferentes lenguajes/arquitecturas partiendo del mismo IDL.

} 7.4 QOS, SEGURIDAD E IMPLEMENTACIONES EN DDS

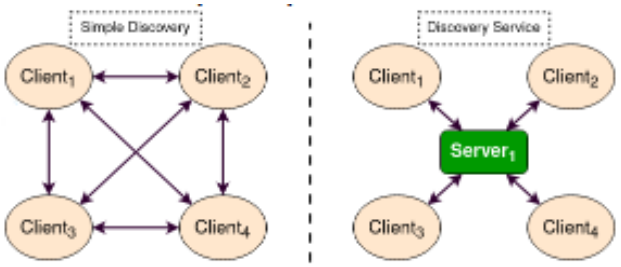
QoS en DDS: Conjunto de parámetros configurables que controlan el comportamiento del sistema DDS, como el gasto de recursos, la tolerancia a fallos o la confiabilidad de la comunicación.
DDS RTPS: Protocolo completo p2p que no requiere broker ni servidores; se puede adaptar por QoS (timeouts, confiabilidad, prioridad); soporta unicast y multicast; es robusto a desconexiones (mantiene sesiones por UDP); encapsula los datos de forma eficiente por Binary XCDR; y es capaz de ser persistente en los datos y de recuperar fallos.
Seguridad en DDS: Entidades autenticadas; control de acceso mediante topics y dominios; integridad en los datos y en la confidencialidad; asegura no ser repudiado; y provee disponibilidad mediante canales confiables.
Implementaciones: Open Source (Fast-DDS y OpenDDS) y propietarias (RTI, MilSoft y Adlink).

} 7.5 USOS

NAV CANADA: Gestiona los 18 millones de km^2 de espacio aéreo civil canadiense y oceánico del Atlántico Norte. Es el segundo proveedor más grande de servicios aéreos, y utilizan DDS para todas sus comunicaciones, proporcionando fiabilidad, escalabilidad y rendimiento, además de reducir costes de desarrollo software.
Planta eólica Siemens: Granja de turbinas con 100 unidades de capacidad con palas de unos 50-100 metros. Controlan ráfagas, requieren una comunicación rápida y dinámica sobre el array de turbinas (filtro selectivo), y gracias a DDS, permite comportamientos inteligentes distribuidos.
Grand Coulee Dam: Está localizada en el río Columbia de Washington, y es la instalación de generación eléctrica que produce la mayor cantidad de energía en EEUU y la mayor estructura de hormigón del país. Está compuesta por 24 turbinas, 12 bombas, y distribuye más de 300.000 valores del sistema.
SCADA (Supervisory Control And Data Acquisition): Realiza un software para ordenadores que permite controlar y supervisar procesos industriales a distancia. La NASA utilizó DDS en el lanzamiento de la nave Orion en 2014, donde combinó más de 300000 nodos a 400000 mensajes por segundo.
Robots: Robotic Drilling Systems utiliza DDS en sus robots, además de sistemas automatizados para perforar el suelo mediante cooperación y flexibilidad.

} 7.6 DDS EN ROS2

Simple Discovery Protocol DDS (inconvenientes): No escala correctamente, ya que el número de mensajes se incrementa con el número de nodos; requiere multicasting, que puede no funcionar correctamente en algunas redes; y FastDDS, utilizado por ROS2, permite un descubrimiento basado en cliente-servidor.



Servicio de descubrimiento de FastDDS: Utiliza el topic utilizado por cada cliente para decidir si 2 nodos querrán comunicarse en el futuro.