Resumen Sistemas Distribuidos

Tomás Felipe Melli

Noviembre 2024

Índice

1 Introducción

Un **Sistema Distribuido** es un conjunto de recursos conectados entre sí que interactúan. Pueden ser varias máquinas conectadas en red, un proce con varias memorias, varios proces que compartan una o más memorias.
Esto tiene varias ventajas

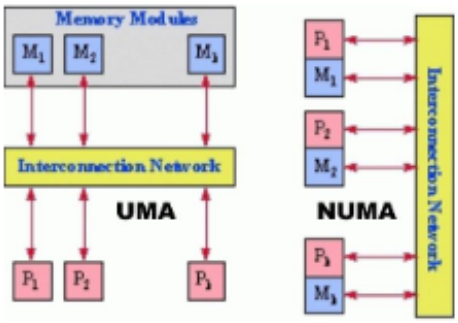
- **Paralelismo** : podremos ejecutar múltiples tareas al mismo tiempo en diferentes nodos.
- **Replicación** : se pueden hacer copias de datos en varios nodos para garantizar **disponibilidad** y **tolerancia a fallos**. Ya que si falla un nodo, otro puede responder sin afectar el funcionamiento del sistema.
- **Descentralización** : no hay un único punto de fallo y se evita la sobrecarga de un único nodo y mejora la escalabilidad.

No todo es color de rosas...

- **Sincronización** : es difícil sincronizar procesos que no comparten ni clock ni memoria y donde la comunicación entre los procesos en diferentes nodos no es instantánea.
- **Coherencia** : mantener copias actualizadas en distintos nodos puede ser complicado debido a los retrasos de propagación de cambios.
- **No comparten clock**
- **Información parcial** : Cada nodo solo tiene acceso a una parte de la información total del sistema.

2 istemas distribuidos de memoria compartida

Harware :



2.1 Si no hay memoria compartida ?

Normalmente son sistemas autónomos. La forma en la que coopera el software se conoce como **arquitectura de software**. A través de la red.

3 TelNet

Es un **protocolo** y un programa que permite conectarse remoto a otro equipo. La idea es que en el equipo en el que yo trabajo corro un programa liviano que sólo se encarga de interactuar con el que me conecto remotamente que absorbe toda la carga de lo que estoy ejecutando.

4 RPC (Remote Procedure Call)

Es un mecanismo que les permite a los programas hacer *procedure call*(invocar una función) de manera remota. Tengo un código local y cuando quiero invocar a esa función remota, está el **stub** que les da a los parámetros un formato estandard (cuando hay diferencias de hw (endianess)) al kernel para que, a través de la red se contacte con el equipo destino, el stub del servidor desempaquete esos parámetros, se ejecute y devuelva la respuesta.

5 Mecanismo Sincrónico

Se invoca a la función y me quedo esperando la respuesta.
Otras versiones

- *RMI* : Java Remote Method Invocation
- *JSON-RPC* : JavaScript Object Notation
- *SOAP* : Simple Object Access Protocol

Lo que tienen en común estos métodos es que **la cooperación** tiene la forma de solicitarle servicios a otros, donde estos últimoss no tienen un rol activo. Este tipo de **arquitectura** se suele llamar **cliente/servidor**. El servidor es un componente que **da servicios** cuando el cliente le pide y entonces, el programa "principal" hace de cliente de los distintos servicios que se van necesitando para completar cierta tarea.

6 Mecanismo Asincrónico

Dijo que no vamos a entrar en detalles del **RPC asincrónico**

- Promises
- Futures
- Windows Asynchronous RPC

Y algo muy utilizado como el **Pasaje de mensajes**(Send/Receive)

- Mailbox
- Pipe
- Message Passing Interface (MPI) para C/C++
- Scala actors : send, receive/react

Que es el mecanismo más general que no supone que haya nada compartido, sólo un **canal de comunicación** como es la red.

6.1 Cómo es el tema de sincronización ?

Como no hay memoria compartida, la sincronización supone otros desafíos.

- Tengo que manejar la **codificación/decodificación** de los datos
- Si hago comunicación asincrónica, tengo que dejar de procesar para atender el traspaso de mensajes.
- La comunicación es lenta
- Eventualmente el canal puede perder mensajes (esto hoy con **TCP/IP** no pasa)
- Podría haber un costo económico por cada mensaje que se transmite

Sin embargo, existen bibliotecas que ayudan a resolver algunos de estos problemas. La más conocida **MPI** que es una API.
Existen otros problemas que exceden el alcance de la materia, pero que se mencionan

- Los nodos pueden morir (en los otros esquemas también pero en este es más probable y hasta de mayor impacto)
- La red se puede partir (partes incomunicadas)

7 Conjetura de Brewer

También conocida como **teorema CAP**, es una conjetura que dice que **en un entorno distribuido no se puede tener a la vez consistencia, disponibilidad y tolerancia a fallas.** Sólo dos de esas tres.

8 Locks en entornos distribuidos

No existe un TestAndSet atómico porque no hay memoria compartida.

8.1 Cómo implememtamos locks ?

Una manera es poner el control de los recursos bajo un único nodo que haga de coordinador. Pensar que dentro de él hay procesos que ofician de representantes *(proxies)* de los procesos remotos y cuando un proceso necesita un recurso, se lo pide a su proxy, que lo "negocia" con el resto utilizando algunos mecanismos ya vistos en sincro.
Este **enfoque centralizado** conlleva una serie de problemas

- Todo depende de un nodo
- Punto único de falla
- Cuello de botella en procesamiento y en capacidad de red
- Se requiere consultar al coordinador (que podría estar lejos, incluso para acceder a recursos cercanos)
- Cada interacción con el coordinador requiere de mensajes que viajan por la red, lo cual es lento

8.2 Existen alternativas ?

Si hay dos nodos que piden un servicio, cómo podemos saber quién pidió primero ? El timestamp de quién tomamos ? Se podría utilizar un protocolo de red como **NTP** y que todos tengan la misma hora, pero transfiere muchos datos a través de la red... en fin, no es buena idea... Afortunadamente, **Leslie Lamport**(inventor de LaTex) propuso

9 Orden parcial no reflexivo - Leslie Lamport

- **Si dentro de un proceso, A sucede antes que B** $A \to B$
- **Si E es el envío de un mensaje y R su recepción** $E \to R$. Aunque E y R sucedan en procesos distintos.
- **Si** $A \to B$ **Y** $B \to C$**, entonces** $A \to C$
- **Si no vale ni** $A \to B$**, ni** $B \to A$**, entonces A y B son concurrentes**

9.1 Cómo se implementa ?

- Cada proce tiene un **reloj**, que puede ser el real, pero alcanza con que sea un valor monótonamente creciente en cada lectura
- Cada mensaje lleva adosada la lectura del reloj
- Como la recepción es siempre posterior al envío, cuando se recibe un mensaje con una marca de tiempo t que es mayor al valor actual del reloj, *se actualiza el reloj interno* $t + 1$

Esto permite un **orden parcial**. Si lo que buscamos es un **orden total**, lo único que queda es *romper un empate* y esta situación sólo se daría frente a eventos concurrentes, entonces se toma un criterio como, el pid.

10 Acuerdo Bizantino

El problema es : un grupo de generales que rodean a una ciudad sólo pueden ganar si todos atacan al mismo tiempo. Se pueden comunicar mediante mensajes, el mensajero puede morir en el camino o alguno de los generales puede ser traidor, deben llegar a un consenso, sino pierden. Cómo logran el acuerdo confiable ?

10.1 Formalizado

Dados:

Fallas	En la comunicación.
Valores	$V = \{0, 1\}$
Inicio	Todo proceso i empieza con $init(i) \in V$.

Se trata de:

Acuerdo	Para todo $i \neq j$, $decide(i) = decide(j)$.
Validez	Existe i, $decide(i) = init(i)$.
Terminación	Todo i decide en un número finito de transiciones (*WAIT-FREEDOM*).

Teorema:

No existe ningún algoritmo para resolver consenso en este escenario.

Dados:

Fallas	Los procesos dejan de funcionar.
Valores	$V = \{0, 1\}$.
Inicio	Todos proceso i empieza con $init(i) \in V$.

Se trata de:

Acuerdo	Para todo $i \neq j$, $decide(i) = decide(j)$.
Validez	Si $\forall i.\ init(i) = v$, entonces $\nexists j.\ decide(j) \neq v$.
Terminación	Todo i que *no falla* decide en un número finito de transiciones.

Teorema:

Si fallan a lo sumo $k < n$ procesos, entonces se puede resolver consenso con $\mathcal{O}((k + 1) \cdot n^2)$ mensajes.

Dados:

Fallas	Los procesos no son confiables.
Valores	$V = \{0, 1\}$
Inicio	Todos proceso i empieza con $init(i) \in V$.

Se trata de:

Acuerdo	$\forall i \neq j$, que *no fallan*, $decide(i) = decide(j) \in V$
Validez	Si $\forall i$, que *no falla*, $init(i) = v$, entonces $\nexists j$, que *no falla*, tal que $decide(j) \neq v$
Terminación	Todo i que *no falla* decide en un número finito de transiciones.

Teorema:

Se puede resolver consenso bizantino para n procesos y k fallas si y sólo si $n > 3 \cdot k$ y la *conectividad* es mayor que $2 \cdot k$

Conectividad: conn(G) = mínimo número de nodos N t.q. $G \setminus N$ no es conexo o es trivial

11 Clusters

Muchos significados

- Conjunto de computadoras conectadas por una red de alta velocidad con un scheduler de trabajos común
- Un conjunto de compus que trabajan cooperativamente desde alguna perspectiva. Usualmente para proveer servicios relacionados.
- **Grids** : conjunto de clusters cada uno con un dominio administrativo distinto
- **Clouds** : clusters donde uno puede alquilar una capacidad fija o bajo demanda

12 Scheduling en Sistemas Distribuidos

Cuando tengo muchos procesos que quiero distribuir la carga (porque son muy complejos). Surgen muchas cuestiones :

12.1 Niveles

- **Local** : dar el proce a un proceso ready
- **Global** : dentro del cluster, el scheduler debe asignar un proceso a un procesador **(mapping)**

12.2 Global : compartir la carga entre los procesadores

- **Estática** : en el momento de la creación del proceso **(affinity)**
- **Dinámica** : la asignación varía durante la ejecución **(migration)**

12.3 Compartir vs Balancear

- Balancear significa repartir equitativamente.
- Hay que evaluar el costo-beneficio

12.4 Migración

Migrar un proceso de un equipo a otro puede ser iniciada por

- el proce sobrecargado **(sender initiated)**
- el proce libre **(receiver initiated / work stealing)**

12.5 Políticas de scheduling

- **Transferencia** : cuándo hay que migrar un proceso.
- **Selección** : qué proceso migrar.
- **Ubicación** : a dónde hay que enviar el proceso.
- **Información** : cómo se difunde el estado.

13 Problemas de Sincronización de Procesos Distribuidos

Existen diversos algoritmos relacionados con los sistemas distribuidos. Ya hablamos de los **proxies** pero ahora vamos a ver los que son **completamente distribuidos**.

13.1 Modelo de Fallas

Un **modelo de fallas** es una caracterización de los tipos de fallas que pueden ocurrir para inducir algoritmos diferentes que mantengan la sincronización.

- **Nadie falla**
- **Los procesos se caen, pero no levantan** (no vuelven a estar operativos)
- **Los procesos caen y pueden levantar**
- **Los procesos caen y pueden levantar pero sólo en determinados momentos**
- **La red se particiona** (ojo con los algortimos de acuerdo)
- **Fallas bizantinas**, los procesos pueden comportarse de manera impredecible

En esta materia vemos **versiones sin fallas**

13.2 Métricas de complejidad

En este tipo de algoritmos, una **métrica** que suele tener mucho sentido es *la cantidad de mensaje que se envían a través de la red.* Aunque no siempre, redes dedicadas de altísima velocidad o algortimos que tienen otros cuellos de botella.
Qué tipos de fallas soportan.
Cuánta información necesitan. Es decir, el tamaño de la red, la cantidad de procesos, cómo ubicar a cada uno de ellos.

13.3 Vamos a abordar problemas vinculados a estas familias...

- **Orden de ocurrencia** de eventos
- **Exclusión mutua**
- **Consenso**

13.4 Exclusión Mutua Distribuida

La forma más sensilla es **token passing**. La idea es *armamos un anillo lógico entre los procesos y hacemos girar un token.* Cuando un proceso quiere entrar en una sección crítica, debe esperar a que le llegue el token.
Dato de color : si no hay fallas, no hay inanición.
La desventaja es que hay mensajes circulando cuando no son necesarios.
Las implementaciones son :

- **FDDI** (Fiber Distributed Data Interface)
- **TDMA** (Time-Division Multiple-Access)
- **TTA** (Timed-Triggered Architecture)

Existe otro enfoque :
Cuando un proceso quiere entrar a la sección crítica **envía a todos una solicitud**(P_i, ts) **siendo** ts **el timestamp**. Cada proceso puede responder inmediatamente o encolar la respuesta. El proceso podrá ingresar **cuando reciba todas las respuestas** y **al salir, responderá a todos los pedidos demorados**. Responderá **"sí"** en caso de que **no le interese ingresar a la sección crítica** o en caso de querer, *si aún no lo hizo y el ts del pedido recibido es* **menor al del que tenga** porque el otro tiene prioridad. Este algoritmo *exige que todos conozcan la existencia de todos los nodos de la red.* No habrá mensajes circulando si no se quiere entrar a la sección crítica.
Se presuponen dos cosas :

- **No se pierden mensajes**
- **Ningún proceso falla**

13.5 Locks Distribuidos

Ya hablamos del nodo coodinador (centralizado), ahora queremos analizar una alternativa totalmente distribuida, **el protocolo de mayoría**. La idea es que queremos obtener un lock sobre un objeto del cual hay copia en $n - lugares$. **Para obtener un lock, debemos pedirlo a por lo menos** $\frac{n}{2} + 1$**sitios**. Cada sitio **responde si puede o no** darlo. **Cada copia del objeto tiene un número de versión.** Si lo escribimos, tomamos el más alto y lo incrementamos en uno. **Puede producir deadlock y en ese caso hay que adaptar los algoritmos de detección**.
Podrían otorgarse dos locks a la vez ?
Para que eso suceda, **dos procesos deberían tener más de la mitad de los locks (imposible)**
Podría un proceso leer una copia desactualizada ?
Para que eso suceda, el proceso que lee, necesita solicitar $k \geq \frac{n}{2} + 1$ locks (para la escritura es análogo). Dicho esto, el timestamp t que lee cumple que es menor a otro existente, que es el "más nuevo" t_j es decir que $t_j > t$. Pero el tema es que el proceso que escribió en ese timestamp t_j, tuvo que replicar el dato actualizado a la mitad + 1 nodos del sistema. Por tanto, no puede suceder.

13.6 Elección de Líder

Este algoritmo consiste en :
Una serie de procesos *debe elegir a uno de ellos como el* ***líder*** para algún tipo de tarea. En una red sin fallas, es sencillo. Se mantiene un **status** que dice que cierto proceso no es líder. Se organizan los procesos en **anillo (algoritmo del anillo (Le Lann, Chang y Roberts))**, y los procesos *hacen circular los ID.* Cuando reciben un mensaje, lo comparan con el ID y hacen circular (retransmiten) el mayor de los dos y el otro lo descartan. Cuando el mensaje dio toda la vuelta, ya saben quién es el líder (el dueño del ID). Se pone un mensaje de notificación para que todos los sepan.
Posibles complicaciones

- Varias elecciones simultáneas
- Procesos que suben y bajan del anillo (procesos que entran y salen del anillo)

Qué complejidad tiene este algoritmo ?

- **Sin fase de stop**: El proceso con el ID más alto es elegido como líder cuando su ID da una vuelta completa al anillo. Sin embargo, algunos procesos pueden no estar seguros de que la elección terminó y podrían seguir iniciando nuevas elecciones. La elección ocurre en $O(n)$, pero pueden seguir existiendo elecciones innecesarias.
- **Con fase de stop** : Después de determinar el líder, se envía un mensaje de notificación por todo el anillo. Este mensaje informa a todos los procesos sobre quién es el líder. Cuando un proceso recibe este mensaje, detiene cualquier otra elección en curso. Como este mensaje debe recorrer todo el anillo nuevamente, el tiempo total es aproximadamente $O(2n)$.

Respecto a la **comunicación** (la cantidad de mensajes que se envían entre procesos), queremos analizarla y en cuanto a ...

- Cada proceso puede iniciar una elección y propagar su ID por el anillo. En el peor caso, cuando todos los procesos inician una elección simultáneamente, cada ID compite con otros y se generan múltiples mensajes. Esto puede llevar a un número total de mensajes del orden de $O(n^2)$, ya que cada mensaje puede propagarse a través de varios procesos.
- Se ha demostrado que la **cota inferior de comunicación** para algortimos de elección líder es $\Omega(n \log n)$. Existen algoritmos más sofisticados como el de **Hirschberg y Sinclair** que se acercan a ella.

13.7 Instantánea Global Consistente

Otro algoritmo para sincronizar procesos en sistemas distribuidos consiste en realizar una **captura del estado de un sistema** de manera que refleje un **estado coherente de todos los procesos y los mensajes circulantes en la red**.
Supongamos que tenemos un estado $E = \sum(E_i)$ donde E_i corresponde al estado de P_i. Lo que **modifica el estado son los mensajes que intercambian los procesos y no eventos externos**. La ides es tener un **snapshot consistente** de E.
Cuando se quiere una instantánea, un proceso se envía a sí mismo un mensaje de **marca**. Cuando P_i recibe un mensaje de **marca** por primera vez, guarda una copia C_i de E_i y *envía un mensaje de marca a todos los procesos*. En ese momento, P_i empieza a registrar todos los mensajes que recibe de cada vecino P_j hasta recibir la marca de todos. Luego **se conforma la secuencia de** $Recibidos_{i,j}$ de todos los mensajes que recibió P_i de P_j antes de que este tome la instantánea. **El estado globla es que cada proceso está en el estado C_i y los mensajes que están en** *Recibidos* **están circulando por la red**.
Este algortimos se puede usar para

- Detección de propiedades estables (una vez que son verdaderas, lo siguen siendo)
- Detección de terminación
- Debbuging distribuido
- Detección de deadlocks

13.8 Protocolo 2PC (Two-Phase-Commit)

El protocolo de compromiso en dos fases (2PC) es un algoritmo utilizado en sistemas distribuidos para garantizar la atomicidad de una transacción distribuida, es decir, que todos los nodos involucrados acuerden confirmar o abortar una transacción de manera coordinada ("acuerdo en dos etapas").
En la **primer fase**(Prepare-Phase), cierto proceso **pregunta a todos** si están de acuerdo en que se haga la transacción. Si se puede ejecutar, los procesos responden que sí(ready), sino no(abort). Si pasado un tiempo máximo no recibe todos los sí, se aborta.
En la **segunda fase**(Commit Phase), si **todos** respondieron que sí, el proceso que demandó la transacción debe mandar un mensaje de confirmación.

Descripción (problema del **COMMIT**):

Valores	$V = \{0\ (abort), 1\ (commit)\}$
Acuerdo	$\nexists i \neq j.\ decide(i) \neq decide(j)$
Validez	❶ $\exists i.\ init(i) = 0 \implies \nexists i.\ decide(i) = 1$
	❷ $\forall i.\ init(i) = 1 \wedge$ no fallas $\implies \nexists i.\ decide(i) = 0$
Term. débil	Si no hay fallas, todo proceso decide.
Term. fuerte	Todo proceso que no falla decide.

Two-phase commit

- Fase 1
  1. $\forall i \neq 1$: i envía $init(i)$ a 1. Si $init(i) = 0$, $decide(i) = 0$.
  2. $i = 1$: Si recibe todos 1, $decide(i) = init(i)$, si no, $decide(i) = 0$.
- Fase 2
  1. $i = 1$: Envía $decide(i)$ a todos.
  2. $\forall i \neq 1$: Si i no decidió, $decide(i)$ es el valor recibido de 1.

Teorema

Two-phase commit resuelve **COMMIT** con terminación débil.

Pero

- Two-phase commit no satisface *terminación fuerte*.
- Solución: three-phase commit (N. Lynch, cap. 7.2 y 7.3).

13.9 Consenso : Tipos de Acuerdo y Aplicaciones

El consenso es un problema fundamental en sistemas distribuidos donde múltiples procesos deben acordar un mismo valor, a pesar de posibles fallas o incertidumbre en la comunicación.
Tipos de Auerdo :

- **K-Agreement** : Generaliza el consenso tradicional, permitiendo hasta k valores distintos en la decisión final. Formalmente: cada proceso i decide un valor $decide(i)$ tal que el conjunto de valores decididos W tiene a lo sumo k elementos. Es útil en escenarios donde no es necesario un acuerdo absoluto, pero sí una reducción en la diversidad de respuestas.
- **Consenso aproximado** : Generaliza el consenso tradicional, permitiendo hasta k valores distintos en la decisión final. Formalmente: cada proceso i decide un valor $decide(i)$ tal que el conjunto de valores decididos W tiene a lo sumo k elementos. Es útil en escenarios donde no es necesario un acuerdo absoluto, pero sí una reducción en la diversidad de respuestas.
- **Consenso probabilístico** : No garantiza un acuerdo determinista, pero la probabilidad de desacuerdo es menor que un umbral ϵ. Formalmente:

  $$Pr[\exists i \neq j,\ decide(i) \neq decide(j)] < \epsilon$$

  Aplicado en sistemas donde un pequeño margen de error es aceptable a cambio de eficiencia, como algoritmos de consenso en redes P2P.

Aplicaciones :

- **Sincronización de relojes** : No garantiza un acuerdo determinista, pero la probabilidad de desacuerdo es menor que un umbral ϵ. Formalmente:

  $$Pr[\exists i \neq j,\ decide(i) \neq decide(j)] < \epsilon$$

  Aplicado en sistemas donde un pequeño margen de error es aceptable a cambio de eficiencia, como algoritmos de consenso en redes P2P.

- **Tolerancia a fallas en sistemas críticos** : En sistemas de control de aviones, plantas nucleares o infraestructuras críticas, se usa consenso para garantizar que todos los nodos tomen decisiones coherentes, incluso si algunos fallan. Protocolos como Byzantine Fault Tolerance (BFT) permiten mantener la operatividad incluso con nodos defectuosos o maliciosos.

14 Otros algoritmos y tecnologías

- Algoritmo Paxos - Lamport 1989 - consenso con tolerancia a ciertas fallas.
- Algoritmo Reliable, Replicated, Redundant, And Fault-Tolerant (RAFT) - mismo objetivo que Paxos, otro enfoque y diseño. Busca simplificar.
- Tecnología Blockchain - registro distribuido que permite la creación y el mantenimiento de un registro digital de transacciones de manera descentralizada y segura.