

Sistemas Distribuidos

Introducción

Diego Fernández Slezak

Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Sistemas Operativos, primer cuatrimestre de 2024

(2) Créditos

Basado fuertemente en trabajo de Sergio Yovine y Fernando Schapachnik.

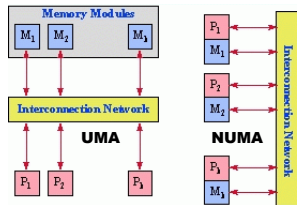
(3) Sistemas distribuidos

- Conjunto de recursos conectados que interactúan.
 - Varias máquinas conectadas en red.
 - Un procesador con varias memorias.
 - Varios procesadores que comparten una (o más) memoria(s).
- Fortalezas:
 - Paralelismo.
 - Replicación.
 - Descentralización.
- Debilidades:
 - Dificultad para la sincronización.
 - Dificultad para mantener coherencia.
 - No suelen compartir clock.
 - Información parcial .

(4) Sistemas distribuidos de memoria compartida

Hardware

- Uniform Memory Access (UMA)
- Non-Uniform Memory Access (NUMA)
- Híbrida



Software

- Estructurada
 - Memoria asociativa: Linda (tuple-spaces), JavaSpaces.
 - Distributed arrays: Fortran, X10, Chapel.
- No estructurada
 - Memoria virtual global.
 - Memoria virtual particionada por localidad.

(5) Y si no hay memoria compartida?

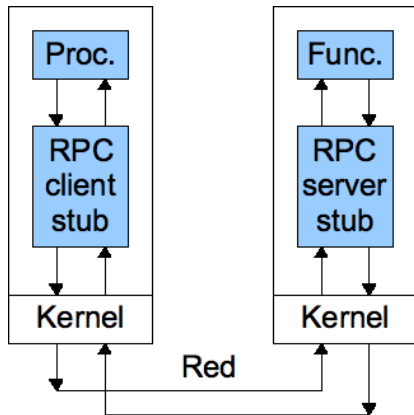
- En ese caso hay algunas alternativas.
- La forma en la que coopera el software, en estos casos, se conoce como *arquitectura de software*.

(6) Conexión remota

- La idea es que los recursos necesarios para cierta parte del procesamiento están en otro equipo.
- Entonces, accedemos a éste como si estuviéramos sentados frente al mismo y hacemos el procesamiento de manera remota.
- Notar que involucra al menos dos equipos, pero sólo uno de ellos hace el trabajo.
- El otro simplemente corre un programa interactivo de comunicaciones, que es bastante liviano.

(7) RPC


- Se trata de un mecanismo que les permite a los programas (C en un principio) hacer *procedure calls* de manera *remota*.
- Involucra una serie de bibliotecas que ocultan del programador los detalles de comunicación y le permiten además enviar los datos de un lugar a otro de la red.



(8) RPC (cont.)

- Notar que es un mecanismo **sincrónico**.
- Algunas versiones:
 - Java Remote Method Invocation (RMI)
 - JavaScript Object Notation (JSON-RPC)
 - Simple Object Access Protocol (SOAP)
- ¿Quién procesa? ¿una o varias máquinas?

(9) En general

- Notar que lo que tienen en común estos métodos es que la cooperación tiene la forma de solicitarle servicios a otros.
- Estos otros no tienen un rol activo.
- Este tipo de arquitecturas se suelen llamar *cliente/servidor*. 
- El servidor es un componente que da servicios cuando el cliente se lo pide.
- Entonces, el programa “principal” hace de cliente de los distintos servicios que va necesitando para completar la tarea.

(10) Mecanismos asincrónicos

Comunicación **asincrónica**:

- RPC asincrónico
 - Promises (B. Liskov, 1988)
 - Futures (Walker, 1990) – Java
 - Windows Asynchronous RPC
- Pasaje de mensajes (*send* / *receive*)
 - Mailbox
 - Pipe
 - Message Passing Interface (MPI) para C/C++
 - Scala actors: *send*, *receive/react*

(11) Callbacks

- Ejemplo:

```
1  import com.twitter.util.Future
2
3  val f: Future[Int] = ???
4
5  f.onSuccess { res: Int =>
6    println("The result is " + res)
7  }
```


(12) Pasaje de mensajes

- De alguna forma, éste es el mecanismo más general.
- Porque no supone que haya nada compartido, excepto un canal de comunicación.
- Notar: si tengo un canal de comunicación muchas veces no voy a necesitar mutexes.
- Problemas que sí vamos a considerar:
 - Tengo que manejar la codificación/decodificación de los datos.
 - Si hago una comunicación asíncrona, tengo que dejar de procesar para atender el traspaso de mensajes.
 - La comunicación es lenta.
 - Eventualmente el canal puede perder mensajes (hoy en día con TCP/IP se puede pensar que el canal es confiable).
 - Eventualmente podría haber un costo económico por cada mensaje que se transite por el canal.
- Existen bibliotecas que ayudan con algunos de estos problemas. La más popular es MPI (que es una API en realidad).

(13) Pasaje de mensajes (cont.)

- Problemas que vamos a ignorar en esta materia:
 - Los nodos pueden morir (también en los otros esquemas, pero acá es más probable/de mayor impacto).
 - La red se puede partir (partes incomunicadas).
- Sólo como curiosidad:
 - Conjetura de Brewer: en un entorno distribuido no se puede tener a la vez consistencia, disponibilidad y tolerancia a fallas. Sólo dos de esas tres.
 - Detalles en “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”, de Seth Gilbert y Nancy Lynch.

(14) Locks en entornos distribuidos

- En entornos distribuidos no hay TestAndSet atómico. 
- Por ende, ¿cómo implementamos locks?
- Hay varias soluciones posibles, empecemos por las más sencillas.
- Una de las más elementales consiste en:
 - Poner el control de los recursos bajo un único nodo, que hace de coordinador.
 - Pensar que dentro de ése nodo hay procesos que ofician de representantes (o *proxies*) de los procesos remotos.
 - Cuando un proceso necesita un recurso se lo pide a su proxy, que lo “negocia” con el resto de los proxies utilizando todos los mecanismos que ya estudiamos.

(15) Enfoque centralizado

- Este enfoque tiene bastantes problemas:
 - Hay un nodo del que todo depende.
 - 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.

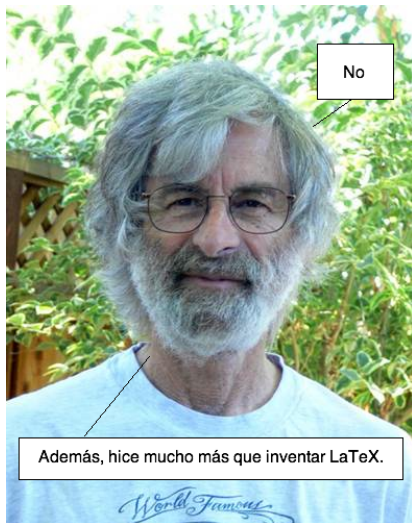
(16) ¿Hay alternativas?

- Sí, hay alternativas.
- Supongamos un “canto guerra pri”.
- En un entorno distribuido, ¿quién cantó “pri”?
 - ¿El que emitió antes el mensaje?
 - ¿El que logró que sus mensajes llegaran primero al resto?
 - ¿Y si hay empate?
 - ¿El que lo hizo cuando el reloj indicaba el timestamp más chico?
 - ¿El reloj de quién?
- Pero... ¿cuándo un evento sucede antes que otro?


(17) Razonando de manera distribuida

- Analicemos qué pasa con el tiempo en los sistemas distribuidos.
- Si la precisión no importa, podemos consultar cada uno nuestro reloj y decidir qué pasó antes de qué.
- El problema aparece cuando necesitamos una precisión de milésimas de segundo.
- Y para los eventos que generan las computadoras, sí la necesitamos.
- Si no, ¿cómo sabemos, por ejemplo, quién pidió primero el lock?
- Podríamos intentar sincronizar relojes...
- Pero hacerlo con mucha precisión, y mantenerlos sincronizados en el tiempo, es caro y difícil de lograr.
- Ahora bien, ¿realmente necesitamos sincronizar relojes?


(18) Leslie opina que no.




(19) Orden parcial entre eventos

- Lamport se dio cuenta de que lo único importante era saber si algo había ocurrido antes o después de otra cosa, pero no exactamente cuándo.
- Su propuesta es definir un *orden parcial no reflexivo* entre los eventos de la siguiente manera: 
 - Si dentro de un proceso, A sucede antes que B , $A \rightarrow B$.
 - Si E es el envío de un mensaje y R su recepción, $E \rightarrow R$. Aunque E y R sucedan en procesos distintos.
 - Si $A \rightarrow B$ y $B \rightarrow C$, entonces $A \rightarrow C$.
 - Si no vale ni $A \rightarrow B$, ni $B \rightarrow A$, entonces A y B son *concurrentes*.

(20) Orden parcial entre eventos (cont.)

- Se implementa de la siguiente forma: 
 - Cada procesador 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 siempre es 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 a $t + 1$.
- Esto da un orden parcial.
- Si lo que queremos es un orden total –y muchas veces lo queremos–, lo único que queda es romper empates.
- Los empates se dan entre eventos concurrentes, entonces se los puede ordenar arbitrariamente.
- Por ejemplo, por el pid.

(21) Acuerdo bizantino

- No se puede saber que él sabe que yo sé que él sabe que yo sé que...
- (Si el medio puede perder mensajes.)
- Problema: 
 - Las distintas divisiones del ejército bizantino rodean una ciudad, desde distintas comarcas. Sólo pueden ganar si atacan todos juntos, así que deben coordinar el ataque.
 - Sólo se pueden comunicar mediante mensajeros que corren de un lugar a otro, pero pueden ser interceptados.
 - Versión más complicada: pueden ser sobornados.
- Imaginemos sólo A y B. A decide hora de ataque y envía mensajero.
- B recibe mensaje. Manda mensajero diciendo que está de acuerdo o proponiendo otra hora.
- Mensajero no llega. ¿Qué hace A?
- Supongamos que sí llega. ¿Cómo sabe B que A sabe?

(22) Acuerdo bizantino, formalización

- Datos:

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.

(23) Acuerdo bizantino, formalización (cont.)

- Datos:

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.

(24) Acuerdo bizantino, formalización (cont.)

- 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:

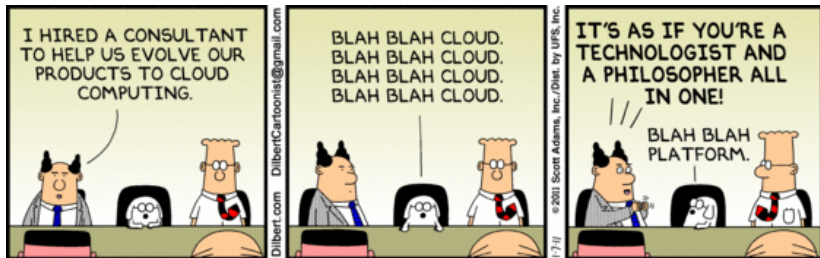
Se puede resolver consenso bizantino para n procesos y k fallas si y sólo si $n > 3 \cdot k$ y la *conectividad* es mayor que $2 \cdot k$.

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

(25) Clusters

- En sentido científico: un conjunto de computadoras conectadas por una red de alta velocidad, con un scheduler de trabajos común.
- En el resto: un conjunto de computadoras que trabajan cooperativamente desde alguna perspectiva. A veces para proveer servicios relacionados, a veces para proveer el mismo de manera redundante.
- Grids: conjunto de clusters, cada uno bajo dominio administrativo distinto.
- Clouds: clusters donde uno puede alquilar una capacidad fija o bajo demanda.

(26) Cloud Computing



(27) Scheduling en sistemas distribuidos

- Dos niveles
 - Local: dar el procesador a un proceso listo
 - Global: asignar un proceso a un procesador (*mapping*)
- 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*)
- Compartir vs *balancear*
 - Balancear: repartir equitativamente
 - Evaluar costo-beneficio

(28) Scheduling en sistemas distribuidos (cont.)

- Migración
 - Iniciada por el procesador sobrecargado
(*sender initiated*)
 - Iniciada por el procesador libre
(*receiver initiated / work stealing*)
- Política de scheduling
 - Transferencia: **cuándo** hay que migrar un proceso.
 - Selección: **qué** proceso hay que migrar.
 - Ubicación: **a dónde** hay que enviar el proceso.
 - Información: **cómo** se difunde el estado.

(29) Bibliografía extra

- “Communicating Sequential Processes”, C. A. R. Hoare, Prentice Hall, 1985. <http://www.usingcsp.com/>
- “Parallel Program Design - A Foundation”, K. Chandy y J. Misra, Addison-Wesley, 1988.
- L. Lamport. Time, clocks, and the ordering of events in a distributed system. CACM 21:7 1978.
<http://goo.gl/ENh2f7>
- L. Lamport, R.Shostak, M.Pease. The Bizantine Generals problem. ACM TOPLAS 4:3, 1982. <http://goo.gl/DY0Qis>

(30) Bibliografía extra

- Nicholas Carriero, David Gelernter: Linda in Context. CACM, 32(4), 1989. <http://goo.gl/gfgbsQ>
- Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. ACM Trans. Comput. Syst. 2, 1 (February 1984), 39-59. <http://goo.gl/3eIskN>
- A. L. Ananda, E. K. Koh. A survey of asynchronous RPC. <http://goo.gl/t96vFg>
- Andrew S. Tanenbaum. RPC. <http://goo.gl/9N3zKz>
- T.L. Casavant, J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. IEEE TSE 14(2):141-154, 1988.
- M. Singhal and N. G. Shivaratri. Advanced Concepts in Operating Systems. McGraw Hill, 1994.