Problemas de sincronización de procesos distribuidos

Rodolfo Baader[1]

[1]Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Sistemas Operativos, segundo cuatrimestre de 2024

(2) Algunos algoritmos interesantes

- Esta parte de la teórica es sobre algoritmos relacionados con los sistemas distribuidos que no se pueden ignorar.
- Si bien los vamos a presentar en una forma simple, constituyen la base desde la que luego se pueden ampliar para escenarios más complicados.
- Recordemos que en clases pasadas vimos algoritmos basados en *proxies*.
- El enfoque de hoy, en general, es sobre los que son completamente distribuidos.

(3) Modelo de fallas

- Cuando se trabaja con algoritmos distribuidos es importante determinar el modelo de fallas.
- Algunas alternativas (que se pueden combinar):
  - Nadie falla (ie, los resultados son correctos si no hay fallas).
  - Los procesos caen pero no levantan.
  - Los procesos caen y pueden levantar.
  - Los procesos caen y pueden levantar pero sólo en determinados momentos.
  - La red se particiona (ojo con los algoritmos de acuerdo).
  - Los procesos pueden comportarse de manera impredecible (*fallas bizantinas*).
- Cada una de ellas induce algoritmos distintos.
- En esta materia nos vamos a concentrar en versiones sin fallas (casi exclusivamente).

(4) Métricas de complejidad

- En este tipo de algoritmos, una métrica que suele tener mucho sentido es la cantidad de mensajes que se envían a través de la red.
- Aunque no siempre: redes dedicadas de altísima velocidad o algoritmos que tienen otros cuellos de botella.
- Otra métrica: qué tipos de fallas soportan.
- Otra forma de evaluarlos: cuánta información necesitan.
  - El tamaño de la red.
  - La cantidad de procesos.
  - Cómo ubicar a cada uno de ellos.

(5) Problemas

- Vamos a abordar problemas que pertenecen a estas tres grandes familias:
  - Orden de ocurrencia de los eventos.
  - Exclusión mutua.
  - Consenso.

(6) Exclusión mutua distribuida

- La forma más sencilla se llama *token passing*. 
- La idea es armar un anillo lógico entre los procesos y poner a circular un token.
- Cuando quiero entrar a la sección crítica espero a que me llegue el token.
- Notar: si no hay fallas, no hay inanición.
- Desventaja: hay mensajes circulando aún cuando no son necesarios.
- Implementaciones:
  - Fiber Distributed Data Interface (FDDI).
  - Time-Division Multiple-Access (TDMA).
  - Timed-Triggered Architecture (TTA).

(7) Exclusión mutua distribuida (cont.)

- Otro enfoque.
- Cuando quiero entrar a la sección crítica envío a todos (incluyéndome) `solicitud(`P_i`, ts)`, siendo *ts* el timestamp.
- Cada proceso puede responder inmediatamente o encolar la respuesta.
- Puedo entrar cuando recibí todas las respuestas.
- Si entro, al salir, respondo a todos los pedidos demorados.
- Respondo inmediatamente si:
  - No me interesa entrar en la sección crítica.
  - Quiero entrar, aún no lo hice y el *ts* del pedido que recibo es menor que el mío, porque el otro tiene prioridad.
- Este algoritmo exige que todos conozcan la existencia de todos.
- No circulan mensajes si no se quiere entrar a la sección crítica.

(8) Exclusión mutua distribuida (cont.)

- Presupuestos:
  - No se pierden mensajes.
  - Ningún proceso falla.

(9) Locks distribuidos

- Ya hablamos de la versión del coordinador de locks centralizado.
- Analicemos una alternativa completamente 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 $n/2 + 1$ sitios.
- Cada sitio responde si puede o no dárnoslo.
- Cada copia del objeto tiene un número de versión. Si lo escribimos, tomamos el más alto y lo incrementamos en uno.
- Ojo: puede producir deadlock, y 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.

(10) Locks distribuidos (cont.)

- ¿Podría leer una copia desactualizada?
- Para que eso suceda debería tener $k \geq n/2 + 1$ locks cuya máxima marca sea t y existir otra copia j, cuya marca $t_j > t$.
- Eso significa que el último que escribió, el que escribió la versión t_j, lo hizo en menos de $n/2 + 1$ copias...
- ...porque si lo hubiese hecho en más copias al menos una de las mías tendría marca t_j.
- Pero eso no puede ser, porque cada proceso escribe en al menos $n/2 + 1$ copias.

(11) Elección de líder

- Una serie de procesos debe elegir a uno como *líder* para algún tipo de tarea.
- En una red sin fallas, es sencillo.
- Le Lann, Chang y Roberts (N. Lynch, cap. 3 y cap. 15.1)
- Mantengo un status, que dice que no soy el líder.
- Organizo los procesos en anillo, hago circular mi ID.
- Cuando recibo un mensaje comparo el ID que circula con el mío. Hago circular el mayor.
- Cuando el mensaje dio toda una vuelta sabemos quién es el líder.
- Ponemos a girar un mensaje de notificación para que todos lo sepan.
- Complicaciones posibles:
  - Varias elecciones simultáneas.
  - Procesos que suben y bajan del anillo.

(12) Elección de líder (cont.)

- Tiempo:
  - Sin fase de *stop* $\mathcal{O}(n)$.
  - Con fase de *stop* $\mathcal{O}(2 \cdot n)$.
- Comunicación:
  - $\mathcal{O}(n^2)$
  - Cota inferior $\Omega(n \log n)$. Algoritmo de Hirschberg y Sinclair.

(13) Instantánea global consistente

- Supongamos que tengo un estado $E = \Sigma E_i$ siendo E_i la parte del estado que le corresponde a P_i.
- Además, lo único que modifica el estado son los mensajes que los procesos se mandan entre sí, y no eventos externos.
- Quiero obtener una instantánea (*snapshot*) consistente de E.
- Esto es, en un momento dado, a partir de que hago el pedido, cuánto valían los E_i y qué mensajes había circulando en la red.

(14) Instantánea global consistente

- 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 otros procesos.
- En ese momento, P_i empieza a registrar todos los mensajes que recibe de cada vecino P_j hasta que recibe `marca` de todos ellos.
- En ese momento queda conformada la secuencia $\text{Recibidos}_{i,j}$ de todos los mensajes que recibió P_i de P_j antes de que éste tomara la instantánea.
- El estado global es que cada proceso está en el estado C_i y los mensajes que están en Recibidos están circulando por la red.

(15) Instantánea global consistente

- Se puede usar para:
  - Detección de propiedades estables (una vez que son verdaderas, lo siguen siendo).
  - Detección de terminación.
  - Debugging distribuido.
  - Detección de deadlocks.

(16) 2PC

- Este algoritmo se llama *Two Phase Commit*, o algo así como "acuerdo en dos etapas".
- La idea es realizar una transacción de manera atómica. Todos debemos estar de acuerdo en que se hizo o no se hizo.
- El espíritu es que en una primera fase le preguntamos a todos si están de acuerdo en que se haga la transacción.
- Si recibimos un *no*, abortamos.
- Si recibimos un *sí*, vamos anotando los que dijeron que sí.
- Si pasado un tiempo máximo no recibimos todos los *sí*, también abortamos.
- Si recibimos todos los *sí*, ahí avisamos a todos que quedó confirmada (esta sería la segunda fase).
- No protege contra todas las fallas, pero sí contra muchas.

(17) 2PC (cont.)

- Descripción (problema del **COMMIT**):

Valores	$V = \{0\ (abort), 1\ (commit)\}$
Acuerdo	$\nexists i \neq j.\ decide(i) \neq decide(j)$
Validez	1. $\exists i.\ init(i) = 0 \implies \nexists i.\ decide(i) = 1$ 2. $\forall i.\ init(i) = 1 \wedge$ 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.

(18) 2PC (cont.)

- 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).

(19) Consenso: otros tipos de acuerdo y aplicaciones

- Acuerdos
  - k-agreement (o k-set agreement):
    $$decide(i) \in W, \text{ tal que } |W| = k$$
  - Aproximado:
    $$\forall i \neq j.\ |decide(i) - decide(j)| \leq \epsilon$$
  - Probabilístico:
    $$Pr[\exists i \neq j.\ decide(i) \neq decide(j)] < \epsilon$$

- Aplicaciones:
  - Sincronización de relojes (NTP, RFC 5905 y anteriores).
  - Tolerancia a fallas en sistemas críticos.

(20) Más de sistemas distribuidos

- 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.

(21) Repaso

- Vimos
  - Hablamos de modelos de fallas y métricas de complejidad.
  - Exclusión mutua y locks distribuidos.
  - Elección de líder.
  - Instantánea global consistente.
  - 2PC.

(22) Bibliografía

- Nancy Lynch, Distributed Algorithms, Morgan Kaufmann, 1996. ISBN 1-55860-348-4.
- Hermann Kopetz, Günther Bauer: The time-triggered architecture. Proceedings of the IEEE 91(1): 112-126 (2003). `http://goo.gl/RPqfas`
- R. Jain. FDDI Handbook. Addison Wesley, 1994. `http://goo.gl/YZ2Hyl`