.UBAfiuba
FACULTAD DE INGENIERÍA

Técnicas de Programación Concurrente I

Concurrencia Distribuida - Parte III

Ing. Pablo A. Deymonnaz
pdeymon@fi.uba.ar

Facultad de Ingeniería
Universidad de Buenos Aires

.UBA fiuba FACULTAD DE INGENIERÍA

Índice

Transacciones: Modelo

- El sistema está conformado por un conjunto de procesos independientes; cada uno puede fallar aleatoriamente
- Los errores en la comunicación son manejados transparentemente por la capa de comunicación
- Storage estable:
  - Se implementa con discos
  - La probabilidad de perder los datos es extremadamente pequeña

Transacciones: Primitivas

- BEGIN TRANSACTION: marca el inicio de la transacción
- END TRANSACTION: finalizar la transacción y tratar de hacer commit
- ABORT TRANSACTION: finalizar forzadamente la transacción y restaurar los valores anteriores
- READ: leer datos de un archivo u otro objeto
- WRITE: escribir datos a un archivo u otro objeto

Transacciones: Propiedades

- Atómicas: la transacción no puede ser dividida
- Consistentes: la transacción cumple con todos los invariantes del sistema
- Aisladas o serializadas: las transacciones concurrentes no interfieren con ellas mismas
- Durables: una vez que se commitean los cambios, son permanentes

En inglés, **ACID**

Excepción: transacciones anidadas no son durables

Transacciones: Implementación (I)

Private Workspace

- Al iniciar una transacción, el proceso recibe una copia de todos los archivos a los cuales tiene acceso
- Hasta que hace commit, el proceso trabaja con la copia
- Al hacer commit, se persisten los cambios
- Desventaja: extremadamente costoso salvo por optimizaciones

Transacciones: Implementación (II)

Writeahead Log

- Los archivos se modifican in place, pero se mantiene una lista de los cambios aplicados (primero se escribe la lista y luego se modifica el archivo)
- Al commitear la transacción, se escribe un registro commit en el log
- Si la transacción se aborta, se lee el log de atrás hacia adelante para deshacer los cambios (*rollback*)

Transacciones: Implementación (III)

Commit en dos fases

- El coordinador es aquel proceso que ejecuta la transacción
- Fase 1
  1. El coordinador escribe *prepare* en su log y envía el mensaje *prepare* al resto de los procesos
  2. Los procesos que reciben el mensaje, escriben *ready* en el log y envían *ready* al coordinador
- Fase 2
  1. El coordinador hace los cambios y envía el mensaje *commit* al resto de los procesos
  2. Los procesos que reciben el mensaje, escriben *commit* en el log y envían *finished* al coordinador

Transacciones: Control de Concur. (I)

Lockeo: two-phase locking

- Fase de expansión: se toman todos los locks a usar
- Fase de contracción: se liberan todos los locks (no se pueden tomar nuevos locks)
- Garantiza propiedad serializable para las transacciones
- Pueden ocurrir deadlocks
- Strict two-phase locking: la contracción ocurre después del commit

Transacciones: Control de Concur. (II)

Concurrencia Optimista

- El proceso modifica los archivos sin ningún control, esperando que no haya conflictos
- Al commitear, se verifica si el resto de las transacciones modificó los mismos archivos. Si es así, se aborta la transacción

Ventajas: Libre de deadlocks y favorece el paralelismo.
Desventajas: Rehacer todo puede ser costoso en condiciones de alta carga.

Transacciones: Control de Concur. (III)

Timestamps

- Existen timestamps únicos globales para garantizar orden (ver algoritmo de relojes de Lamport)
- Cada archivo tiene dos timestamps: lectura y escritura y qué transacción hizo la última operación en cada caso
- Cada transacción al iniciarse recibe un timestamp
- Se compara el timestamp de la transacción con los timestamps del archivo:
  - Si es mayor, la transacción está en orden y se procede con la operación
  - Si es menor, la transacción se aborta
- Al *commitear* se actualizan los timestamps del archivo

Índice

Deadlocks: Detección (I)

Algoritmo centralizado

- El proceso coordinador mantiene el grafo de uso de recursos
- Los procesos envían mensajes al coordinador cuando obtienen / liberan un recurso y el coordinador actualiza el grafo
- Problema: los mensajes pueden llegar llegar desordenados y generar falsos deadlocks
- Posible solución: utilizar timestamps globales para ordenar los mensajes (algoritmo de Lamport)

Deadlocks: Detección (II)

Algoritmo distribuido

- Cuando un proceso debe esperar por un recurso, envía un probe message al proceso que tiene el recurso. El mensaje contiene: id del proceso que se bloquea, id del proceso que envía el mensaje y id del proceso destinatario
- Al recibir el mensaje, el proceso actualiza el id del proceso que envía y el id del destinatario y lo envía a los procesos que tienen el recurso que necesita
- Si el mensaje llega al proceso original, tenemos un ciclo en el grafo

Deadlocks: Prevención (I)

Algoritmo wait-die

- Se asigna un timestamp único y global a cada transacción al iniciar (algoritmo de Lamport)
- Cuando un proceso está por bloquearse en un recurso (que tiene otro proceso), se comparan los timestamps
  - Si el timestamp es menor (proceso más viejo), espera
  - Si no, el proceso aborta la transacción

Deadlocks: Prevención (II)

Algoritmo wound-wait

- Se asigna un timestamp único y global a cada transacción al iniciar (algoritmo de Lamport)
- Cuando un proceso está por bloquearse en un recurso (que tiene otro proceso), se comparan los timestamps
  - Si el timestamp es menor (proceso más viejo), se aborta la transacción del proceso que tiene el recurso, para que el más viejo pueda tomarlo
  - Si no, el proceso espera

Bibliografía

- **Distributed Operating Systems**, Andrew S. Tanenbaum, capítulo 3
- **Computer Networks**, Andrew S. Tanenbaum y David J. Wetherall, quinta edición