
IIC2523

Sistemas Distribuidos

— Hernán F. Valdivieso López —
(2025 - 2 / Clase 15)

Control de Concurrency

¿Cómo manejar accesos simultáneos? ¿uso *locks*?

Temas de la clase

1. Conflictos por concurrencia
2. Estrategias de Control de concurrencia: optimista y pesimista
3. Resolución de conflictos

Conflictos por conurrencia

¿En qué consisten?

Problemas comunes

Transacción Distribuida

La clase pasada vimos que

La presencia de múltiples usuarios en un sistema distribuido genera solicitudes concurrentes a sus recursos. Cada recurso debe diseñarse para ser seguro en un entorno concurrente.

Ahora vamos a estudiar los problemas y soluciones que existen producto de la concurrencia

Conflictos por Concurrency

- ◆ Un par de operaciones entra en conflicto si su efecto combinado depende del orden en que se ejecutan.
- ◆ Los casos a analizar son:
 - ◆ Lectura-Escritura (*Read-Write*): El efecto de una operación de lectura dependerá del momento en el que queda la operación de escritura.
 - ◆ Escritura-Escritura (*Write-Write*): El resultado final de escritura dependerá de cuál de las 2 operaciones se ejecuta al final.
 - ◆ Lectura-Lectura (*Read-Read*): No hay conflicto, ya que la lectura será igual sin importar el orden.

Conflictos por Concurrency

Problemas comunes: *Lost Update Problem*.

- ◆ Conflicto tipo *Write-Write*.
- ◆ Dos o más transacciones concurrentes trabajan sobre una misma variable. Puede ser que sobrescriben o bien, una de las transacciones no vea la actualización de la otra antes de realizar su propia escritura.

Conflictos por Concurrencia

Problemas comunes: *Lost Update Problem*.

- ◆ Conflicto tipo *Write-Write*.
- ◆ Dos o más transacciones concurrentes trabajan sobre una misma variable. Puede ser que sobrescriben o bien, una de las transacciones no vea la actualización de la otra antes de realizar su propia escritura.
- ◆ Ejemplo: 2 transacciones van a modificar el precio de un producto que estaba a 5000. Una (T1) para a reducir en 1000 pesos su precio y otra (T2) le aplicará un descuento del 50%. El precio final puede ser:
 - ◆ Opción 1: 4000 (T1 sobrescribe a T2)
 - ◆ Opción 2: 2500 (T2 sobrescribe a T1)
 - ◆ Opción 3: 2000 (T2 se ejecuta después de T1)
 - ◆ Opción 4: 1500 (T1 se ejecuta después de T2)

Conflictos por Concurrency

Problemas comunes: *Inconsistent Retrievals Problem*

- ◆ Conflicto tipo *Read-Write*.
- ◆ Una transacción (T1) está modificando una variable, todavía no *commitea* y otra transacción (T2) lee esa misma variable. Si la transacción T1 consolida la información, la transacción T2 está en un punto obsoleto.

Conflictos por Concurrency

Problemas comunes: *Inconsistent Retrievals Problem*

- ◆ Conflicto tipo *Read-Write*.
- ◆ Una transacción (T1) está modificando una variable, todavía no *commitea* y otra transacción (T2) lee esa misma variable. Si la transacción T1 consolida la información, la transacción T2 está en un punto obsoleto.
- ◆ Ejemplo: Cargamos un producto para ver su precio (T1), pero mientras estaba cargando la página, le aplican un 50% de descuento (T2). El precio final puede ser:
 - ◆ El original (T1 sin ver lo que hizo T2).
 - ◆ El con 50% (T1 viendo lo que hizo T2).

Estrategias de Control de conurrencia

¿Cómo controlamos la
conurrencia para evitar el
conflicto?

Control pesimista

Control optimista

Estrategias de Control de Concurrency

- ◆ Los protocolos o estrategias de control de concurrency son mecanismos para gestionar el acceso simultáneo de múltiples usuarios o procesos a los mismos datos, evitando conflictos y manteniendo la integridad de la información.
- ◆ Estudiaremos 2 estrategias:
 - ◆ **Control Pesimista:** Evita los conflictos a medida que ocurren las operaciones.
 - ◆ **Control Optimista:** Validar conflicto al momento de querer hacer *commit* y abortar/rechazar en caso de detectar conflicto.

Estrategias de Control de Concurrencia - Pesimista



Estrategias de Control de Concurrency - Pesimista

- ◆ Uso de Exclusión Mutua para ordenar las transacciones que acceden al mismo recurso y así evitar conflictos.
- ◆ Generalmente, la Exclusión Mutua es del tipo ***Many Readers o Single Writer***.
 - ◆ Múltiples transacciones de lectura concurrentes o bien una sola transacción de escritura.
 - ◆ Si llega una solicitud de lectura, solo permiten otras de lectura, pero nunca de escritura hasta que finalicen todas las de lectura.
 - ◆ Si llega una operación de escritura, no permite que ocurra ninguna otra operación sobre el mismo dato hasta que finalice la escritura.

Estrategias de Control de Concurrency - Pesimista

- ◆ Uso de Exclusión Mutua para ordenar las transacciones que acceden al mismo recurso y así evitar conflictos.
- ◆ Generalmente, la Exclusión Mutua es del tipo Many Readers o Single Writer.
 - ◆ Múltiples transacciones de lectura concurrentes o bien una sola transacción de escritura.
 - ◆ Si llega una solicitud de lectura, solo permiten otras de lectura, pero nunca de escritura hasta que finalicen todas las de lectura.
 - ◆ Si llega una operación de escritura, no permite que ocurra ninguna otra operación sobre el mismo dato hasta que finalice la escritura.
- ◆ También existen las exclusiones Jerárquicas, que permite diferentes granularidades de bloqueo.
 - ◆ Por ejemplo, bloquear a nivel de base de datos completo, solo una tabla o una fila de una tabla en particular.

Estrategias de Control de Concurrency - Pesimista

Desventaja

- ◆ El uso de Exclusión Mutua puede conllevar eventualmente algún *deadlock* distribuido.
- ◆ Una solución es recurrir al *timeout* para que una transacción bloqueada aborta posterior a un tiempo de espera. Así se rompe la cadena de espera.
- ◆ Otra solución, y la más ocupada, es la detección de interbloqueos buscando ciclos en el grafo de espera (*wait-for graph*) y abortar alguna de las transacciones de dicho ciclo bajo alguna heurística.

Estrategias de Control de Concurrency - Pesimista

Ejemplo

T1: *Begin* *read X* *Commit*

T2: *Begin* *write Z* *Commit*

T3: *Begin* *write X* *write Z* *Commit*

1. Cuando T3 hace *write X*, aplica un "Lock" a X.
2. Cuando T1 hace *read X*, debe esperar porque está con "Lock".
3. Cuando T2 hace *write Z*, aplica un "Lock" a Z.
4. Cuando T3 hace *write Z*, debe esperar porque está con "Lock".
5. T2 hace *commit* y "Lock" Z.
6. T3 puede hacer *write Z*, le aplica *lock*. Luego hace *commit* libera Z y X.
7. T1 puede hacer *read X*, le aplica *lock*. Luego hace *commit* y libera X.

Estrategias de Control de Concurrencia - Optimista



Estrategias de Control de Concurrency - Optimista

- ◆ Permite que las transacciones procedan de forma aislada hasta que estén listas para confirmar (*commit*).
- ◆ Antes de finalizar el *commit* se incluye una fase de validación para verificar conflictos con otras transacciones. Si la validación detecta conflicto, la transacción no se le permite hacer *commit*.
 - ◆ Se puede dejar en *stand-by* o bien ser abortada. Dependerá del sistema.
- ◆ La validación puede ser mediante:
 - ◆ **Timestamps:** contrastando los tiempos de último modificación consolidada versus la hora que empezó la transacción.
 - ◆ **Versiones:** cada recurso tiene un contador con su versión. La transacción sabe la versión de cada recurso al momento de empezar y espera que esa versión se mantenga al momento de finalizar.

Estrategias de Control de Concurrency - Optimista

- ◆ Existen 2 tipos de validación para detectar el conflicto *Read-Write*.
- ◆ *Forward Validation* y *Backward Validation*

Estrategias de Control de Concurrency - Optimista

- ◆ Existen 2 tipos de validación para detectar el conflicto *Read-Write*.
- ◆ ***Forward Validation*** y *Backward Validation*
- ◆ La transacción TX que quiere hacer *commit* toma las datos que hizo *WRITE* y verifica que ninguna transacción abierta (todavía no hace *commit*) haga *READ* de alguna de los datos modificados.
- ◆ En resumen, se busca no hacer *commit* si eso repercute en que otra transacción vaya a tener que abortar.
- ◆ La transacción "mira hacia el adelante", no puede hacer *commit* si eso va a generar un conflicto con otra transacción que tiene potencial de hacer *commit*.



Estrategias de Control de Concurrency - Optimista

- ◆ Existen 2 tipos de validación para detectar el conflicto *Read-Write*.
- ◆ *Forward Validation* y ***Backward Validation***
- ◆ La transacción TX que quiere hacer *commit* toma los datos que hizo READ y revisa que ninguna transacción que logró hacer *commit* durante el tiempo de vida de la TX (desde BEGIN hasta COMMIT) haga WRITE de los datos que TX leyó.
- ◆ En resumen, se busca no hacer *commit* si se lee un valor que ya no es el que corresponde actualmente.
- ◆ La transacción "mira hacia atrás", no puede hacer *commit* si eso va a generar un conflicto con una transacción que ya hizo *commit*.

Estrategias de Control de Concurrency - Optimista

- ◆ Existen 2 tipos de validación para detectar el conflicto *Read-Write*.
- ◆ Sea T_v la transacción a validar

Aspecto	<i>Forward Validation</i>
T_v se valida contra	Transacciones activas
Operación a validar	Escrituras de T_v contra otras lecturas
Intención	No invalidar lecturas futuras
Riesgo de abortos	Depende del solapamiento con transacciones activas
Complejidad	Requiere seguimiento de transacciones activas




Estrategias de Control de Concurrency - Optimista

- Existen 2 tipos de validación para detectar el conflicto *Read-Write*.
- Sea T_v la transacción a validar

Aspecto	<i>Forward Validation</i>	<i>Backward Validation</i>
T_v se valida contra	Transacciones activas	Transacciones ya consolidadas
Operación a validar	Escrituras de T_v contra otras lecturas	Lecturas de T_v contra otras escrituras
Intención	No invalidar lecturas futuras	Asegurar que las lecturas fueron válidas
Riesgo de abortos	Depende del solapamiento con transacciones activas	Depende del solapamiento con <i>commits</i> previos
Complejidad	Requiere seguimiento de transacciones activas	Mirar historial de <i>commits</i> consolidadas desde que empezó T_v




Estrategias de Control de Concurrency - Optimista

Considerando las siguientes transacciones:

T1		<i>Begin</i>	<i>read X</i>	<i>Commit</i>
T2		<i>Begin</i>	<i>write Z</i>	<i>Commit</i>
T3		<i>Begin</i>	<i>write X</i> <i>write Z</i>	<i>Commit</i>

Estrategias de Control de Concurrency - Optimista

Considerando las siguientes transacciones:




T1		<i>Begin</i>	<i>read X</i>	<i>Commit</i>
T2		<i>Begin</i>	<i>write Z</i>	<i>Commit</i>
T3		<i>Begin</i>	<i>write X</i> <i>write Z</i>	<i>Commit</i>

Validación *Forward*

1. Cuando T3 quiera hacer *commit*, se detecta que T1 leyó X. → No puede.
2. Cuando T1 quiera hacer *commit*, no escribió en ningún recurso. → Se acepta.
3. Cuando T2 quiera hacer *commit*, ya no hay transacciones activas. → Se acepta.

Estrategias de Control de Concurrency - Optimista

Considerando las siguientes transacciones:

T1		<i>Begin</i>	<i>read X</i>	<i>Commit</i>
T2		<i>Begin</i>	<i>write Z</i>	<i>Commit</i>
T3		<i>Begin</i>	<i>write X write Z</i>	<i>Commit</i>

Validación *Backward*

1. Cuando **T3** quiera hacer *commit*, no hay transacciones *commiteadas*. → Se acepta.
2. Cuando **T1** quiera hacer *commit*, esta leyó X y T3 escribió en ella. → Se aborta.
3. Cuando **T2** quiera hacer *commit*, esta no leyó nada. → Se acepta.

Estrategias de Control de Concurrency - Optimista

- ◆ Existen 2 tipos de validación para detectar el conflicto *Read-Write*.
- ◆ Sea T_v la transacción a validar

Aspecto	<i>Forward Validation</i>	<i>Backward Validation</i>
T_v se valida contra	Transacciones activas	Transacciones ya consolidadas
Operación a validar	Escrituras de T_v	Lecturas de T_v

! Importante ! ! Importante ! ! Importante ! ! Importante !

- ◆ Por defecto se validan conflictos *Read-Write*, pero se puede implementar para también validar conflictos *Write-Write*.
- ◆ Solo se incluye, en la lista de operaciones a validar, el otro tipo de operación según la validación a realizar.

Resolución de Conflictos Concurrentes

Aceptemos que ocurren
conflictos e vamos a resolverlos
de alguna forma

Resolución de Conflictos Concurrentes

- ◆ En vez de utilizar una estrategia para evitar el conflicto. Ahora se genera y se ofrece alguna política para resolverlo.
- ◆ Rompe *one-copy serializability* y **puede intentar repararse según la política.**
- ◆ La forma en que resolvemos estos conflictos es crítica y puede tener un impacto significativo en la experiencia del usuario y en la complejidad del sistema.

Resolución de Conflictos Concurrentes

- ◆ Esta capacidad de resolución no es excluyente a las estrategias de control. Un sistema distribuido puede utilizar ambas en distintas capas.
 - ◆ Por ejemplo, cada archivo puede ser accedido por 1 nodo a la vez (control pesimista), pero varias personas o programas que usan dicho nodo pueden modificar el archivo y eventualmente existirán conflictos que se deberán solucionar.
- ◆ Las políticas que vamos a ver no son excluyentes, pueden aplicarse más de una a la vez.

Resolución de Conflictos Concurrentes

Última escritura gana (*Last Write Wins*)

- ◆ Una política de resolución simple donde la versión con la marca de tiempo más reciente se considera la correcta.
- ◆ DynamoDB de Amazon.com utiliza este tipo de resolución.



Resolución de Conflictos Concurrentes

Versionamiento

- ◆ Genera más de una versiones de datos. Puede ser en el mismo recurso donde deja una versión como "actual" y otra en el historial. O bien generar múltiples recursos.
- ◆ Dropbox utiliza este tipo de resolución.



Resolución de Conflictos Concurrentes

Merge manual

- ◆ El sistema obliga al usuario a intervenir para resolver el conflicto antes de continuar funcionando correctamente.
- ◆ Git utiliza este tipo de resolución.



Resolución de Conflictos Concurrentes

Merge manual

- ◆ El sistema obliga al usuario a intervenir para resolver el conflicto antes de continuar funcionando correctamente.
- ◆ Git utiliza este tipo de resolución.

```
If you have questions, please  
<<<<<< HEAD  
open an issue  
=====  
ask your question in IRC.  
>>>>>> branch-a
```

Resolución de Conflictos Concurrentes

Merge semántico

- ◆ El sistema utiliza conocimiento específico de la aplicación para resolver automáticamente los conflictos. Ya sea que el usuario le indica una estrategia o el sistema tiene una implementada por defecto.
- ◆ Git puede hacer eso con el comando `git merge theirs` o `git merge ours`



Resolución de Conflictos Concurrentes

Operational Transformation (OT)

- ◆ Un tipo de *merge semántico* donde no se comparan estados, sino que se analizan y modifican los estados para ser operaciones que se pueden en algún ordenes
- ◆ Google Docs hace este protocolo para la edición en tiempo real.



Resolución de Conflictos Concurrentes

Conflict-Free Replicated Data Types (CRDTs)

- ◆ El sistema utiliza una estructuras de datos diseñadas para tener conflictos, pero garantizando convergencia automática y eventual mediante **reglas algebraicas**.
- ◆ Un juego (*League of Legend*) utiliza Riak CRDTs (base NoSQL) para solucionar conflictos en ciertas funcionalidades.



Resolución de Conflictos Concurrentes

Protocolo	Cómo resuelve	Pros	Contras
<i>Last Write Wins</i>	Se queda con el valor con <i>timestamp</i> mayor	Simple	Pierde datos concurrentes
Versiones	Genera más de un recurso	Simple	Costoso en almacenamiento
<i>Merge manual</i>	Un usuario o proceso decide qué versión conservar	Preciso	Costoso, lento, requiere intervención
<i>Merge semántico</i>	Usa lógica del dominio para combinar versiones	Automatizado y preserva más información	Requiere lógica específica por tipo de dato, complejo de implementar.
Operational Transformation	Reescribe operaciones para que puedan aplicarse en cualquier orden		
<i>Conflict-free Replicated Data Types</i>	Diseñados para que todas las operaciones concurrentes se pueden combinar sin conflicto.		Limitado a tipos de datos específicos.

Poniendo a prueba lo que hemos aprendido 🙄

Estás diseñando un sistema de monitoreo médico en tiempo real para pacientes hospitalizados. Varios sensores distribuidos con múltiples tipos de datos escriben sobre el paciente analizado en la misma base de datos (su ritmo cardíaco, la presión, oxigenación, etc.).

El sistema debe estar disponible en todo momento, incluso cuando nadie esté revisando el sistema y cualquier pérdida de datos por conflictos puede poner en riesgo vidas humanas. ¿Cuál es la **estrategia más apropiada** para enfrentar esta situación?

- a. Aplicar control pesimista.
- b. Aplicar control optimista.
- c. Utilizar *Last Write Wins*.
- d. Utilizar Versionamiento.
- e. Utilizar *Merge* manual.

Poniendo a prueba lo que hemos aprendido 🙄

Estás diseñando un sistema de monitoreo médico en tiempo real para pacientes hospitalizados. Varios sensores distribuidos con múltiples tipos de datos escriben sobre el paciente analizado en la misma base de datos (su ritmo cardíaco, la presión, oxigenación, etc.).

El sistema debe estar disponible en todo momento, incluso cuando nadie esté revisando el sistema y cualquier pérdida de datos por conflictos puede poner en riesgo vidas humanas. ¿Cuál es la **estrategia más apropiada** para enfrentar esta situación?

- a. Aplicar control pesimista.
- b. Aplicar control optimista.
- c. Utilizar *Last Write Wins*.
- d. Utilizar Versionamiento.**
- e. Utilizar *Merge* manual.

Próximos eventos

Próxima clase

- ◆ Teoremas CAP y PACELC
- ◆ ¿Cuales son las límites reales de Fiabilidad en los Sistemas Distribuidos? ¿puedo garantizar todos los beneficios?

Evaluación

- ◆ Control 4 ya disponible, se entrega el lunes.
- ◆ Último día de la tarea 2.
- ◆ Última tarea se publica la otra semana, abordaremos mecanismo de control de concurrencia con 2PC para simular un entorno de transacciones distribuidas.

IIC2523

Sistemas Distribuidos

— Hernán F. Valdivieso López —
(2025 - 2 / Clase 15)

Créditos (animes utilizados)

Medalist



Chainsaw man

