Exercises
Distributed Systems: Part 2
Summer Term 2015

6. Distributed Concurrency Control and Replication

Albert-Ludwigs-Universität Freiburg

Anas Alzoghbi

Department of Computer Science

Databases and Information Systems

UNI FREIBURG

Exercise 1

1- Verify whether or not the schedules are serializable

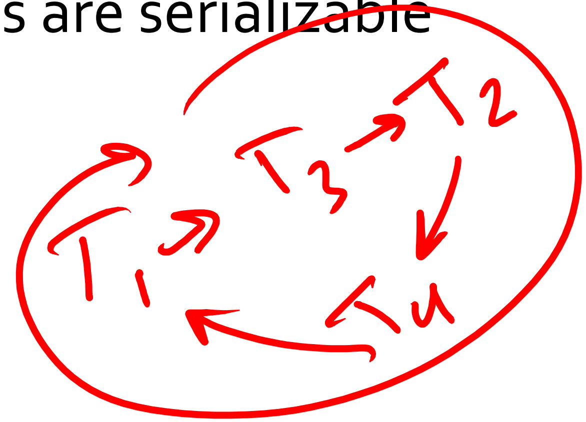
- $S_1 : R_1A \ W_1A \ R_2A \ W_2A$
  - $T_1 \rightarrow T_2$
- $S_2 : R_2B \ W_2B \ R_1B \ W_1B$
  - $T_2 \rightarrow T_1$

- $S_1 : R_1A \ W_2A$
  - $T_1 \rightarrow T_2$
- $S_2 : R_3B \ W_1B \ R_2C \ W_3C$
  - $T_2 \rightarrow T_3 \rightarrow T_1$

Exercise 1

1- Verify whether or not the schedules are serializable

- S_1 : R_1A R_3A R_3B W_3A W_3B R_2B
  - $T_1 \rightarrow T_3 \rightarrow T_2$
- S_2 : R_4D W_4D R_1D R_2C R_4C W_4C
  - $T_2 \rightarrow T_4 \rightarrow T_1$



- S_1 : W1A c1 R3A R3B c3 W2B c2
  - $T_1 \rightarrow T_3 \rightarrow T_2$
- S_2 : W2C c2 R4C R4D c4 W1D c1
  - $T_2 \rightarrow T_4 \rightarrow T_1$

Exercise 1

2- Demonstrate that applying Distributed 2PL prevents non serializable schedules.

- $S_1 : R_1A\ \ W_1A\ \ R_2A\ \ W_2A$
  $S_2 : R_2B\ \ W_2B\ \ R_1B\ \ W_1B$
  - S_1 : T_1 waits for the last operation in S_2
  - S_2 : T_2 waits for the last operation in S_1

- $S_1 : R_1A\ \ W_2A$
  $S_2 : R_3B\ \ W_1B\ \ R_2C\ \ W_3C$
  - S_1: T_1 waits for W_1B from S_2
  - S_2: T3 cannot unlock until the end

➔ Even locally on S_2 not applicable!

Exercise 1

2- Demonstrate that applying Distributed 2PL prevents non serializable schedules.

- S_1 : R_1A R_3A R_3B W_3A W_3B R_2B
- S_2 : R_4D W_4D R_1D R_2C R_4C W_4C
  - S_1: T_1 waits for R_1D from S_2
  - S_2: T4 cannot unlock until the end
  - ➔ Even locally on S_2 not applicable!

- S_1 : W1A c1 R3A R3B c3 W2B c2
- S_2 : W2C c2 R4C R4D c4 W1D c1
  - Local commit violate global 2PL protocols if they went through. At c_1, the lock on A can't be released since T_1 On S_2 has not yet claimed all of its locks

Exercise 1

2- Demonstrate that applying Distributed Timestamp Protocol prevents non serializable schedules.

- S_1 : R_1A W_1A R_2A W_2A
  S_2 : R_2B W_2B R_1B W_1B
  - $Z(T_1) < Z(T_2)$: So R_1B performs a read on B which has been written to by a "later" transaction before (W_2B)

- S_1 : R_1A W_2A
  S_2 : R_3B W_1B R_2C W_3C
  - $Z(T_1) < Z(T_3)$, so W_1B performs a write on B which has been read to by a "later" transaction before (R_3B)

Exercise 1

2- Demonstrate that applying Distributed Timestamp protocol prevents non serializable schedules.

- S_1 : R_1A R_3A R_3B W_3A W_3B R_2B

  S_2 : R_4D W_4D R_1D R_2C R_4C W_4C
    - $Z(T_1) < Z(T_4) < Z(T_3)$, so R_1D performs a read on D which has been written to by a "later" transaction before (W_4D)

- S_1 : W_1A c1 R_3A R_3B c3 W_2B c2

  S_2 : W_2C c2 R_4C R_4D c4 W_1D c1
    - $Z(T_1)$)< $Z(T_2) < Z(T_3) < Z(T_4)$ so W_2B performs a write on B which has been read by a "later" transaction before (R_3B).
    - The same problem occurs for W_1D and R_4D.

Exercise 1

3- Check whether or not the schedules are rigorous

(Commits should occur before any conflicting operation!)

(i) Commits at the global end of a transaction

Then all schedules are not rigorous, since conflict pairs exist before abort or commit

- $S_1: R_1A \ W_1A \ R_2A \ W_2A$

  $S_2: R_2B \ W_2B \ R_1B \ W_1B$
- $S_1: R_1A \ W_2A$

  $S_2: R_3B \ W_1B \ R_2C \ W_3C$
- $S_1: R_1A \ R_3A \ R_3B \ W_3A \ W_3B \ R_2B$

  $S_2: R_4D \ W_4D \ R_1D \ R_2C \ R_4C \ W_4C$

Exercise 1

3- Check whether or not the schedules are rigorous

(Commits should occur before any conflicting operation!)

(ii) Commit as soon as possible after the local end

- S_1 : R_1A W_1A c_1 R_2A W_2A c_2 (rigorous)

  S_2 : R_2B W_2B c_2 R_1B W_1B c_1 (rigorous)

- S_1 : R_1A c_1 W_2A c_2 (rigorous)

  S_2 : R_3B W_1B c_1 R_2C c_2 W_3C c_3 (not rigorous)

- S1 : R_1A c_1 R_3A R_3B W_3A W_3B c_3 R_2B c_2 (rigorous)

  S2 : R_4D W_4D R_1D c_1 R_2C c_2 R_4C W_4C c_4 (not rigorous)

Exercise 1

3- Check whether or not the schedules are rigorous

- S_1 : W_1A c_1 R_3A R_3B c_3 W_2B c_2

  S_2 : W_2C c_2 R_4C R_4D c_4 W_1D c_1

All commits happen before any conflicting operation

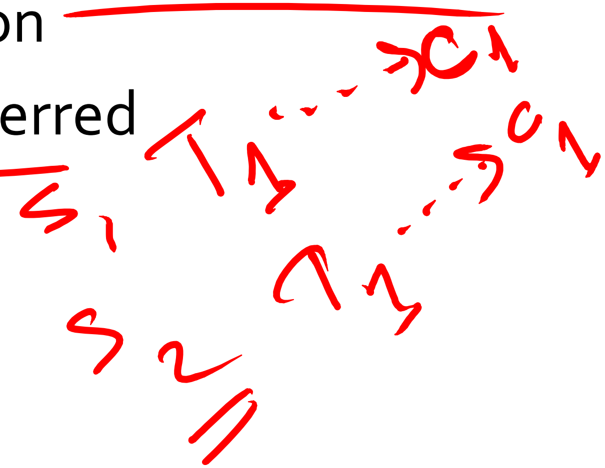
➔ rigorous

Exercise 1

3- Check whether or not the schedules are commit-deferred.

(i) Commits at the global end of a transaction

➔ By definition all schedules are commit deferred



Exercise 1

3- Check whether or not the schedules are commit-deferred.

(ii) Commit as soon as possible after the local end

- $S_1 : R_1A\ \ W_1A\ c_1\ R_2A\ \ W_2A\ c_1$

  $S_2 : R_2B\ \ W_2B\ c_2\ R_1B\ \ W_1B\ c_1$

T_1 at S_1 commits before T_1 at S_2 ➔ Not commit deferred

- $S_1 : R_1A\ c_1\ W_2A\ c_2$

  $S_2 : R_3B\ \ W_1B\ c_1\ R_2C\ c_2\ W_3C\ c_3$

➔ Commit deferred

- $S1 : R_1A\ c_1\ R_3A\ \ R_3B\ \ W_3A\ \ W_3B\ c_3\ R_2B\ c_2$

  $S2 : R_4D\ \ W_4D\ \ R_1D\ c_1\ R_2C\ c_2\ R_4C\ \ W_4C\ c_4$

T_1 at S_1 commits before T_1 at S_2 ➔ Not commit deferred

Exercise 1

3- Check whether or not the schedules commit-deferred.

- S_1 : W_1A c1 R_3A R_3B c3 W_2B c2

  S_2 : W_2C c2 R_4C R_4D c4 W_1D c1

T_1 at S_1 commits before T_1 at S_2 ➔ Not commit deferred

Exercise 1

4- Demonstrate that applying Ticket-based concurrency control prevents non-serializable schedules

- S_1 : $\mathbf{R_1 I_1 W_1 I_1}$ R_1A W_1A $\mathbf{R_2 I_1 W_2 I_1}$ R_2A W_2A
- S_2 : $\mathbf{R_2 I_2 W_2 I_2}$ R_2B W_2B $\mathbf{R_1 I_2 W_1 I_2}$ R_1B W_1B

Local detection is not possible, but the access to I_1 and I_2 happens in different order. Using dependency graph on the tickets we can detect the cycle: $T_1 \rightarrow T_2$, $T_2 \rightarrow T_1$

Exercise 1

4- Demonstrate that applying Ticket-based concurrency control prevents non-serializable schedules

- S_1 : $\mathbf{R_1 I_1}$ $\mathbf{W_1 I_1}$ $R_1 A$ $\mathbf{R_2 I_1}$ $\mathbf{W_2 I_1}$ $W_2 A$

  S_2 : $R_3 B$ $\mathbf{R_1 I_2}$ $\mathbf{W_1 I_2}$ $W_1 B$ $\mathbf{R_2 I_2}$ $\mathbf{W_2 I_2}$ $R_2 C$ $W_3 C$

Tickets introduce $T_1 T_2$ order on S_2 which makes the conflict explicit and locally detectable at S_2 → conflict is locally detectable

Exercise 1

4- Demonstrate that applying Ticket-based concurrency control prevents non-serializable schedules

- $S_1 : R_1A\ R_3A\ R_3B\ W_3A\ W_3B\ R_2B$

  $S_2 : R_4D\ W_4D\ R_1D\ R_2C\ R_4C\ W_4C$

Like in the previous case, ticket introduce T1T2 order on S2, making the conflict locally detectable.

- $S_1 : W_1A\ c1\ R_3A\ R_3B\ c3\ W_2B\ c2$

  $S_2 : W_2C\ c2\ R_4C\ R_4D\ c4\ W_1D\ c1$

Like in the first case, no local detection is possible, but a dependency graph on the tickets detects the conflict.

Exercise 2

Keeping consistency in replicated data is a key issue, for which several approaches exist

a) Compare the combinations of update primary copy/update anywhere and eager/lazy propagation in terms of availability, consistency and cost for read/write operations

Exercise 2

Keeping consistency in replicated data is a key issue, for which several approaches exist

a) Compare the combinations of update primary copy/update anywhere and eager/lazy propagation in terms of availability, consistency and cost for read/write operations
   - All eager methods suffer from write availability and performance problems. Consistency is strong. Lazy has opposite behavior
   - Primary copy might lead to bottleneck, Write anywhere don't. but can lead to deadlock or can provide very weak guarantees.

Exercise 2

Keeping consistency in replicated data is a key issue, for which several approaches exist

b) What kind of consistency problems could occur with a read quorum 2/3N+1 and a write quorum of N/3+1?

Exercise 2

Keeping consistency in replicated data is a key issue, for which several approaches exist

b) What kind of consistency problems could occur with a read quorum 2/3N+1 and a write quorum of N/3+1?

- Since the write quorum is below N/2+1, two write operations cannot be performed concurrently without excluding each other (no majority of participants needed). As a results, conflicting write operations are possible.
- On the other hand, the read and write quora do form a majority, so the are no read/write consistency issues.

Exercise 3

Eventual consistency provides high availability and scalability, but limits consistency

a) Provide examples of consistency problems/anomalies that could occur!

Exercise 3

Eventual consistency provides high availability and scalability, but limits consistency

a) Provide examples of consistency problems/anomalies that could occur!

- Each replica may perform update on its data elements independently and will later propagate the outcome to other replicas. This way, the updates have no ordering guarantee. Without any additional measures, write may get lost, dirty writes may occur, ...

Exercise 3

Eventual consistency provides high availability and scalability, but limits consistency

b) In current cloud storage systems, Latest write wins is a popular approach to resolve concurrent updates. Explain the problems that may occur when using physical/wall-clock timestamps!

Exercise 3

Eventual consistency provides high availability and scalability, but limits consistency

b) In current cloud storage systems, Latest write wins is a popular approach to resolve concurrent updates. Explain the problems that may occur when using physical/wall-clock timestamps!

- Wall/physical clocks cannot be kept fully in global sync, and they might not even be monotonic (meaning that they might jump backwards). As a result, older results make overtake newer results, essentially invalidating the Last Write Wins guarantee.

Exercise 3

Eventual consistency provides high availability and scalability, but limits consistency

c) Describe an approach that uses logical clocks to handle such concurrent updates

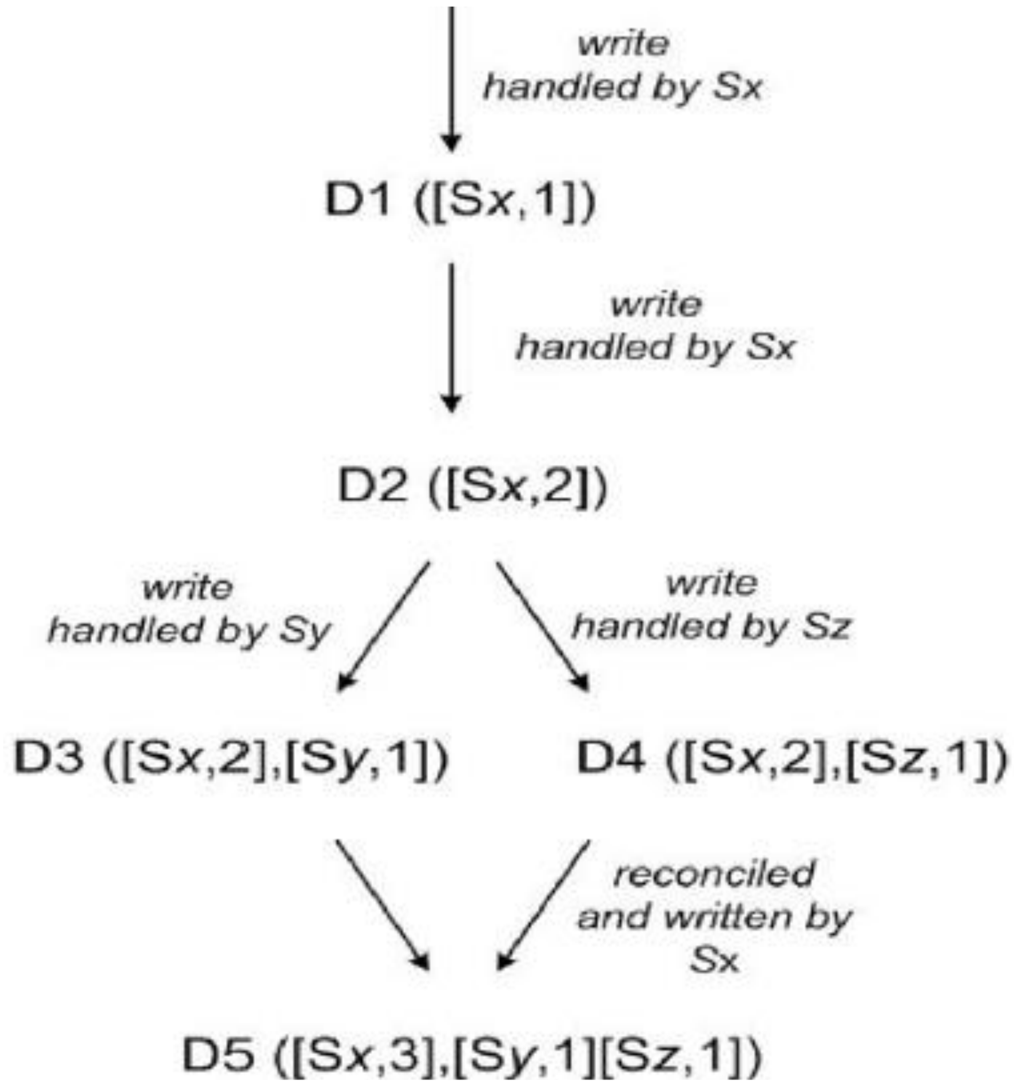
Exercise 3

Eventual consistency provides high availability and scalability, but limits consistency

c) Describe an approach that uses logical clocks to handle such concurrent updates

- Vector clocks are being used to denote timestamps/versions. Each update is performed specifying the base version and leads to an increase in the vector clock. A partially ordered/graph/branching history is built when performing concurrent updates. Reconciliation can be performed at the application level, similar to merging in a version control system.

Exercise 3


write
handled by Sx
D1 ([Sx,1])
write
handled by Sx
D2 ([Sx,2])
write
handled by Sy
write
handled by Sz
D3 ([Sx,2],[Sy,1])
D4 ([Sx,2],[Sz,1])
reconciled
and written by
Sx
D5 ([Sx,3],[Sy,1][Sz,1])

Exercise 4

- Different consistency models provide different tradeffos between availability and consistency

a) Explain why preventing lost updates can lead to unavailability

Exercise 4

- Different consistency models provide different tradeffos between availability and consistency

a) Explain why preventing lost updates can lead to unavailability

T_1 : R(X; 100)W(X; 100 + 20 = 120)

T_2 : R(X; 100)W(X; 100 + 30 = 130)

- Regardless of whether x = 120 or x = 130 is chosen by a replica, the database state could not have arisen from any serial execution of T1 and T2. To prevent this, either T1 or T2 should not have committed. Each client's respective server might try to detect that another write occurred, but this requires knowing the version of the latest write to x. This is only possible by communicating

Exercise 4

- Different consistency models provide different tradeffos between availability and consistency

b) How can you guarantee Read Committed, but stay available? Describe an approach that uses logical clocks to handle such concurrent updates

Exercise 4

- Different consistency models provide different tradeffos between availability and consistency

b) How can you guarantee Read Committed, but stay available? Describe an approach that uses logical clocks to handle such concurrent updates

- If each client never writes uncommitted data to shared copies of data, then transactions will never read each others' dirty data. As a simple solution, clients can buffer their writes until they commit, or, alternatively, can send them to servers, who will not deliver their value to other readers until notified that the writes have been committed