Universität Freiburg
Institut für Informatik
Prof. Dr. P. Fischer/Prof. Dr. F. Kuhn

Georges-Köhler Allee, Geb. 51
D-79110 Freiburg
peter.fischer@informatik.uni-freiburg.de

Exercises
Distributed Systemes: Part 2
Summer Term 2015
21.7.2015
Solution Proposal

6. Exercise sheet: Distributed Concurrency Control and Replication

Exercise 1
Consider the following local schedules:

- $S_1: \; R_1A \quad W_1A \quad R_2A \quad W_2A$
  $S_2: \; R_2B \quad W_2B \quad R_1B \quad W_1B$
- $S_1: \; R_1A \quad W_2A$
  $S_2: \; R_3B \quad W_1B \quad R_2C \quad W_3C$
- $S_1: \; R_1A \quad R_3A \quad R_3B \quad W_3A \quad W_3B \quad R_2B$
  $S_2: \; R_4D \quad W_4D \quad R_1D \quad R_2C \quad R_4C \quad W_4C$
- $S_1: \; W_1A \quad c_1 \quad R_3A \quad R_3B \quad c_3 \quad W_2B \quad c_2$
  $S_2: \; W_2C \quad c_2 \quad R_4C \quad R_4D \quad c_4 \quad W_1D \quad c_1$

(1) Verify whether or not the schedules are serializable.
(2) Demonstrate that applying Distributed 2PL/Timestamp Protocol prevents non-serializable schedules.
(3) Check whether or not the schedules are rigourous and commit-deferred.
(4) Demonstrate that applying Ticket-based concurrency control prevents non-serializable schedules.

Solution:

(1)
- locally yes, globally no, S_1: $T_1 \to T_2$, S_2: $T_2 \to T_1$
- locally yes, globally no, S_1: $T_1 \to T_2$, S_2: $T_2 \to T_3 \to T_1$
- locally yes, globally no, S_1: $T_1 \to T_3 \to T_2$, S_2: $T_2 \to T_4 \to T_1$
- locally yes, globally no, S_1: $T_1 \to T_3 \to T_2$, S_2: $T_2 \to T_4 \to T_1$

(2) Distributed 2PL: If a local transaction reaches the point when it would start unlocking, it would ask the other sites running the same transaction if they also reached this point. If not, it would have to wait
- not possible, since either transaction cannot make progress after the first two steps
- not possible, since $R_1 1$ and $R_3 2$ cannot make progress after the first step
- not possible, since $R_1 1$ and $R_4 2$ cannot make progress after the first and second step, respectively
- local commit violate global 2 PL protocols if they went through. At c_1, the lock on A cannot be released since $T_1 2$ has not yet claimed all its locks

Timestamp Protocol: Abort transactions, if a conflicting access is performed with a later timestamp. Without restricting generality, we always assume that S_1 start is transactions earlier than S_2
- $TS_1 < TS_2$, so R_1B performs a read on B which has been written to by a "later" transaction before (W_2B)
- $TS_1 < TS_3$, so W_1B performs a write on B which has been read to by a "later" transaction before (R_3B)
- $TS_1 < TS_4 < TS_3$, so R_1D performs a read on D which has been written to by a "later" transaction before (W_4D)
- $TS_1 < TS_2 < TS_3 < TS_4$, 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.

(3) The last case is easiest: The schedules are rigorous since all a commit happens before any conflicting operation. It is not commit-deferred since e.g. $T_1 1$ commits before $T_1 2$.

In first three cases, no commit is specified. We therefore have the options to either perform the commit at (i) the global end of a transaction or (ii) as soon as possible after the local end. (i) would make the schedules commit-deferred (by definition), but not rigorous, since conflict pairs exist before abort or commit. (ii) would make some of the schedules rigorous, but not all of them.

(4) Tickets are expressed by adding a Ticket access into the local schedules of global transactions. When "locking" the ticket (which we denote as I_j for server j) we add an explicit Read/Write Operation.

- $\begin{array}{lllllllll} S_1: & R_1I_1 & W_1I_1 & R_1A & W_1A & R_2I_1 & W_2I_1 & R_2A & W_2A \\ S_2: & R_2I_2 & W_2I_2 & R_2B & W_2B & R_1I_2 & W_1I_2 & R_1B & W_1B \end{array}$ In this case, no local detection is 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.
- $\begin{array}{lllllllll} S_1: & R_1I_1 & W_1I_1 & R_1A & R_2I_1 & W_2I_1 & W_2A & & \\ S_2: & R_3B & R_1I_2 & W_1I_2 & W_1B & R_2I_2 & W_2I_2 & R_2C & W_3C \end{array}$ Tickets introduce T_1T_2 order on S_2 which makes the conflict explicit and locally detectable at S_2, since the execution without ticket yields the order $T_2 \rightarrow T_3 \rightarrow T_1$
- Like in the previous case, we ticket introduce T_1T_2 order on S_2, making the conflict locally detectable.
- Like in the first case, no local detection is possible, but a depedency graph on the 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

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

Solution:

a)
- All eager methods will suffer from write availability and write performancy problems, since all replicas need to be contacted for a commit. Consistency, on the other hand, is strong. Lazy methods show the opposite behavior.
- Primary copy approaches allow simple updates (only local locking, fast bulk propagation), but may become the bottleneck from due to contention for writing and replicating (mostly in eager, though)
- Write anywhere method are flexible, do not provide a single bottleneck, but may run into deadlock due to distributed locking (eager) or can only provide very weak guarantees.
- ...

b) 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!

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!

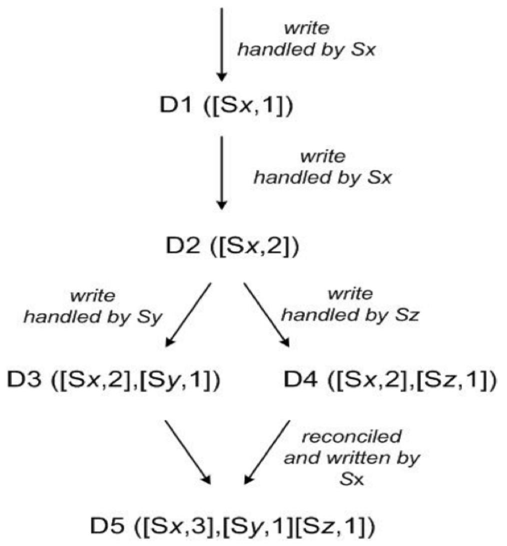
c) Describe an approach that uses logical clocks to handle such concurrent updates

Solution:

a) 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, ...

b) 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.

c) 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 4

Different consistency models provide different tradeoffs between availability and consistency

a) Explain why preventing lost updates can lead to unavailability

b) How can you guarantee Read Committed, but stay available?

Solution:

a) Consider two clients who submit the following T1 and T2 on opposite sides of a network partition which both access a data item X:
$T1 : R(X, 100)W(X, 100 + 20 = 120)$
$T2 : R(X, 100)W(X, 100 + 30 = 130)$
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, since each side may make progress on its own. Formally speaking, we would now require Linearizability, which is the consistency level specified for the CAP theorem.

b) 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. Unlike a lock-based implementation, this implementation does not provide recency or monotonicity guarantees but it satisfies the implementation-agnostic definition.