School of Computer Science, McGill University

COMP-512 Distributed Systems

Assignment 2; Due on 28-October, 9am – no extensions

Please be aware that students have to work individually on this assignment. After discussion with the group partner submit then one single solution.

Exercise 1: Total Order Multicast (30 Points)

In class, we discussed several total order multicast primitives. Now assume a group whose members are scattered in several clusters. Each cluster consists of a set of members in a local area network, while the different clustered are connected via wide-area links. A total order algorithm could use a mix of a sequencer and a token-based approach. Within each cluster there is a sequencer that receives the messages from the other nodes in its cluster for forwarding, while a token circulates among the sequencers of the different clusters to control system wide message dissemination.

Write the message in pseudo-algorithmic form as practiced in class.

Exercise 2: Serializability and Schedules (30 Points)

1. $S_1 = r_1(a), r_2(b), r_2(a), w_2(a), c_2, w_3(c), r_1(c), c_1, r_3(b), c_3$
2. $S_2 = w_1(a), w_2(b), r_3(c), r_3(a), c_3, r_1(b), c_1, w_2(c), c_2$

For each of the schedules answer the following questions:

a.) For both S1 and S2 build the dependency graph and determine whether the schedule is serializable or not. If it is serializable, give all equivalent serial schedules. If it is not serializable, indicate the conflicts in the schedule that induce the cycle.

b.) Let's have a closer look at S_1 in a homogenous distributed system. Object a resides on node $N1$, b on node $N2$, and c on node $N3$. Assume the order of operations indicated in S_1 is the order in which transactions submit their operations to the system. Assume T_1 submits to $N1$, T_2 submits to $N2$, and T_3 submits to $N3$.
Assume all DBS use strict 2PL with a local lock manager. *Show how the execution progresses (similar to the examples in class) indicating when messages are exchanged, where operations take place, and when/where locks are requested and released and who commits/aborts. If an operation is blocked because of a lock conflict all following operations of the same transaction are also blocked. Give a conflict equivalent serial schedule (omitting any aborted transactions).*

c.) Let's have a closer look at S_2 in a middleware based system. Object a resides on node $N1$, b on node $N2$, and c on node $N3$. Assume the order of operations indicated in S_2 is the order in which transactions submit their operations to the system. Assume a strict 2PL scheduler is implemented at the middleware.
Show how the execution progresses (similar to the examples in class) indicating when messages are exchanged, where operations take place, and when/where locks are requested and released and who commits/aborts. Assume that each transaction requests commit immediately after its last operation. Give a conflict equivalent serial schedule (omitting any aborted transactions).

Exercise 3: Serializability (25 Points)

The algorithm below describes a concurrency control protocol (commonly found in current application servers) that allows some non-serializable executions. During the execution of a transaction it keeps local copies of all data items accessed by the transaction. Changes are done on the local copies. Only at the end of transaction, are the real database items changed. The algorithm is a special form of an optimistic concurrency control mechanism.

- Upon *beginTransaction* request for transaction T_i
  1. $WriteSet(T_i) := ReadSet(T_i) := \emptyset$
- Upon $read(x)$ request of T_i
  1. If $x \in WriteSet(T_i)$
     a.) return x from $WriteSet(T_i)$
  2. else if $x \in ReadSet(T_i)$
     a.) return x from $ReadSet(T_i)$
  3. else
     a.) request shared lock for x
     b.) make copy of x (last committed value of x) and add to $ReadSet(T_i)$.
     c.) release shared lock for x
- Upon $write(x)$ request of T_i
  1. If $x \in WriteSet(T_i)$
     a.) update value of copy of x in $WriteSet(T_i)$
  2. else if $x \in ReadSet(T_i)$
     a.) copy x from $ReadSet(T_i)$ into $WriteSet(T_i)$
     b.) update value of copy of x in $WriteSet(T_i)$.
  3. else
     a.) request shared lock for x
     b.) make copy of x (last committed value of x) and add to $ReadSet(T_i)$ and $WrtieSet(T_i)$.
     c.) release shared lock for x
     d.) update value of copy of x in $WriteSet(T_i)$
     e.) return ok
- Upon *abortTransaction* request of T_i
  1. abort T_i and discard $ReadSet(T_i)/Writeset(T_i)$
  2. return abort
- Upon *closeTransaction* of T_i
  1. For each $x \in Writeset(T_i)$
     a.) request exclusive lock for x
     b.) if current value of x not the same as copy of $x \in ReadSet(T_i)$
        i. abort T_i and discard $Writeset(T_i)/ReadSet(T_i)$
        ii. release all locks
        iii. return abort
  2. For each $x \in Writeset(T_i)$
     a.) set x to value of copy of $x \in WriteSet(T_i)$.
  3. commit transaction
  4. release all locks
  5. return commit

1. *Are the following anomalies possible: (i) lost update, (ii) unrepeatable read, (iii) write skew, (iv) dirty read, (v) dirty write? For each of them: if not possible give an explanation of what in the algorithm avoids it. If possible indicate what part of the algorithm makes this possible.*

2. *Provide an example execution/schedule of a set of transactions that is allowed under this concurrency control mechanism but that is not serializable and is neither a lost update, an unrepeatable read or a write skew. Indicate what is the reason why the schedule is not serializable (e.g., show the pairs of conflicting operations that lead to a cycle in the serialization graph) and what part of the algorithm leads to such behavior.*

Exercise 4: Location of Lock Manager (15 Points)

Given the possible architectures for data distribution and the location of the lock manager: (i) a middleware and the middleware has a lock manager, (ii) a homogenous system and clients can connect to any of the nodes; each node has its own lock manager, (iii) a homogenous system and clients can connect to any of the nodes; the nodes request locks from a central lock manager whenever needed.

For each of the configurations give one advantage is has over the others (in terms of message overhead, complexity, deadlocks, etc.