School of Computer Science, McGill University

COMP-512 Distributed Systems, Fall 2015

Written Assignment 3

Due date: Tuesday, 8-Dec; Extension without penalty 14-Dec 10am; no submission accepted after

Exercise 1: Replication based on TO-multicast (30 Points)

In this exercise you have to develop a eager, update anywhere ROWA replication algorithm that uses a total-order multicast to help with serializing transactions.

The idea is to execute a transaction T completely locally using strict 2PL at the local database replica R to which T is submitted. At commit time (before commit is returned to the user), a write set (containing all write operations and maybe some information about read operations) is constructed (obviously, during execution the transaction has to keep track of which read and write operations it has executed). The replicas build a group and we use the total order multicast primitive provided by group communication systems to send write sets. As a result, when replica R_1 executes transaction T_1 locally, and R_2 executes T_2 locally and both send the write sets WS_1 and WS_2 concurrently, the group communication system will order them. That is, either all database replicas receive WS_1 before WS_2 or vice versa. if WS_1 is received before WS_2, then T_1 has to be serialized before T_2 and if this is not possible, then T_2 (the second to be delivered) has to abort. That is, T_2 may only commit if for any pair of conflicting operations $((r_1(a), w_2(a)), (w_1(a), r_2(a)), (w_1(a), w_2(a)))$, T_1's operation is executed before T_2's operation on all replicas where both operations are executed (reads are only executed on the local replica!).

The tricky thing is that read operations are local and only the local replica can see conflicts with local and remote writes (when remote write sets arrive). Thus, the local replica has to make the final decision about T_i's outcome (because it knows the read operations) and multicasts this commit/abort decision (this will happen sometime after the multicast of the WS_i).

1. *Write the algorithm that handles replica and concurrency control. Consider all events that can occur at a replica R (submission of read, write and commit operations of local transactions, receiving of writesets, etc).*
2. *Argue why your algorithm is correct (only allows serializable executions).*

The algorithm itself is not long but thinking about the correct behavior at any given site is not trivial. I suggest you to go through example executions and consider what is the right behavior. If you have come up with a sequence of actions and writesets, go through a few examples to ensure that the algorithm does what you want it to do....

It can look something like

- Upon start request for local transaction T_i, do something
- Upon $r_i(a)$ request of local transaction T_i, do something
- Upon $w_i(a)$ request of local transaction T_i, do something
- Upon c_i request of local transaction T_i, do something
- // Upon receiving the writeset WS_i of transaction T_i, do something
- //Maybe more events to consider....

Exercise 2: Replication Execution (20 Points)

Assume a database system with three objects a, b, c and two database replicas N_1 and N_2. We assume full replication, i.e., both N_1 and N_2 have copies of all three objects. There are four transactions $T_1 - T_4$. T_2 and T_4 are submitted to N_1; T_1 and T_3 are submitted to N_2.

Assuming that N_1 and N_2 have neither concurrency control nor replica control, the operations are submitted (and executed at the respective replica) in the following order.

$r_1(a), r_2(a), r_3(b), r_4(b), w_2(a), w_3(b), c_3, w_2(c), c_2, w_4(b), c_4, r_1(a), c_1$

Now assume the database system uses strict 2PL and a ROWA approach.

Show for the following 2 replica control protocols the execution as time proceeds. In case of blocking of an operation, assume that all following operations of the same transaction are also blocked. Indicate the messages that are exchanged, where operations take place, when/where locks are requested/granted and released, and who commits/aborts, etc.

1. The system uses eager primary copy replication in the following way. All read operations (executed to either primary or secondary) acquire a local shared lock, execute the read operation locally and return the result to the user. Write operations lead to an abort of the transaction if submitted to a secondary. When the primary receives a write operation, it sends the write request to the secondaries, then acquires an exclusive lock locally and executes the operation locally and then returns to the clients. Thus, while a write operation is forwarded to the secondaries when it is submitted, the primary does not wait for the secondaries to confirm that they have executed it before returning the ok to the client. It only has to be executed locally. Only at commit time, when the 2PC protocol is run, it is checked whether the secondaries were actually able to handle the requests. That is, when the primary sends the vote request for a transaction T and a secondary is still waiting for an exclusive lock for T to be granted, the secondary aborts T locally and votes no.
2. The system uses lazy update everywhere replication. Assume that the write sets of all update transaction are propagated only after all transactions have executed locally (i.e., after the final c_1 at N_2). To resolve conflicts, assume the system uses site priorities whereby N_1 has the higher priority.

Exercise 3: Paxos (20 Points)

Assume a kind-of Chubby replicated database based on a Paxos maintained replicated log. There are three database servers S1, S2 and S3. Each accepts requests to the database. When a database server receives a request it runs a Paxos instance to decide on the position of the request in a replicated log. After a successful execution of the Paxos instance, the request will be included at each replica of the log at exactly the same position. The database then executes all requests in the order they were successfully added to the log.

Now assume S1 receives the request *update*$(x, 5)$ requesting to set the value of object x to 5, and S2 receives the request *delete*(x) requesting to delete object x.

Both start Paxos instances at the same time, first attempting to become leader and then propose their request to be put into the replicated log at position 25. S1 chooses as BallotID 6, and S2 chooses s BallotID 7. Each sequence below shows the start of a possible sequence of message exchanges for the two Paxos instances. The sequence shows the order in which messages are received. One can assume that when a message is received, the actions following the receipt are successfully executed.

1. *For each of the sequences indicate whether it is possible that the servers agree to update the object (the request of S1) or agree to delete the object (request of S2) for the position 25 of the replicated log.*

2. *Provide reasoning for decision, i.e., explain why this is possible or why Paxos prevents it.*

Note that *propose* refers to the message sent by a server to ask whether it can be leader, *promise* is the message returned by an acceptor that it accepts the node as a leader with the requested BallotID, *accept* is the message in which the server sends the value to the acceptors, and *accept-ack* is the message in which an acceptor confirms to the leader that it has accepted the value.

1. Sequence 1

```
S1 to S1: propose(6)
S1 to S1: promise(6,NIL)
S1 to S2: propose(6)
S2 to S1: promise(6,NIL)
S1 to S3: propose(6)
S3 to S1: promise(6,NIL)
S2 to S1: propose(7)
S2 to S2: propose(7)
S2 to S3: propose(7)
...the rest of the messages in some order
```

2. Sequence 2

```
S1 to S1: propose(6)
S1 to S1: promise(6,NIL)
S1 to S2: propose(6)
S2 to S1: promise(6,NIL)
S1 to S3: propose(6)
S3 to S1: promise(6,NIL)
S1 to S1: accept(6, ‘‘update(x,5) at position 25")
S1 to S1: accept-ack(6,ok)
S2 to S1: propose(7)
S2 to S2: propose(7)
S2 to S3: propose(7)
...the rest of the messages in some order
```

3. Sequence 3

```
S1 to S1: propose(6)
S1 to S1: promise(6,NIL)
S1 to S2: propose(6)
S2 to S1: promise(6,NIL)
S1 to S3: propose(6)
S3 to S1: promise(6,NIL)
S1 to S1: accept(6, ‘‘update(x,5) at position 25")
S1 to S1: accept-ack(6,ok)
S1 to S3: accept(6, ‘‘update(x,5) at position 25")
S3 to S1: accept-ack(6,ok)
S2 to S1: propose(7)
S2 to S2: propose(7)
S2 to S3: propose(7)
...the rest of the messages in some order
```

NOTE: there is a slight error on slide 11, where the actions on receiving a ballot proposal are described. It should be:

- If BallotID higher than any you have seen so far, *promise*
    - Log ballotID in persistent memory
    - return to would-be leader:
        - highest ballotID to which promised so far
        - if already accepted value, then return value, too
- else (already sent promise to higher ballotID)
    - do nothing or send a refuse

Exercise 4: Recovery in fault-tolerant systems (30 Points)

In class, we discussed how a fault-tolerance system can be built using either active or passive replication. If nodes fails, eventually new replicas have to be brought into the system. When a new node joins the system it has to get the current state of the system. Assume that all data is maintained in main-memory and the new server starts with an empty data storage. In a simple offline-recovery system, request executed would be halted, i.e, all currently executing requests would be finished while new incoming requests would be put into a queue. Only when there is no request executing, an existing node would transfer copies of all objects to the new node. Once the transfer is completed, request execution can resume. If there are many objects to transfer request execution could be halted for a long time.

Now assume the replicas use a group communication system that offers primitives to multicast messages (FIFO- and total-order), and to join (and leave a group). The group communication system synchronizes the change in group configuration and message delivery with what is named *view synchrony*. For the purpose of this exercise, we do not consider node failures here but only the action of adding a new node (replica) to the group. Assume all replicas are in a group named "rep-group". Let view V be the set of current members of the "rep-group" at a certain time (e.g, R_A and R_B). Then a new replica R_C wants to be added; it makes a join call to the GCS layer to join group "rep-group". After the GCS has successfully added the new replica it will deliver to each replica a *view change notification* VCN. This notification indicates the new set $V' = \{R_A, R_B, R_C\}$ of group members including the identifiers of all old replicas and the new one. Furthermore, the GCS provides the following property: If there is a message m that was multicast by one of the old members of old view V, then m will either be delivered to all members of the old view V before the view change notification V', or it will be delivered to all members of the new view V' after the view change notification. That is, assume, for instance, a total order multicast and the GCS delivers the messages $m_1, m_2, m_3, ...m_n$ in that order. Assume replicas R_A and R_B are members of view V, and R_A, R_B and R_C are members of the new view V'. Then, if the delivery sequence at R_A is $m_1, m_2, VCN(R_A, R_B, R_C), m_3, ...m_n$, then R_B receives exactly the same sequence while R_C receives $VCN(R_A, R_B, R_C), m_3, ...m_n$.

Assume a system that maintains approx. 10,000 different objects of reasonable size. For both active and passive replication outline how you would do recovery without stopping request execution during recovery. Feel free to describe your solution both through informal descriptions and (half-) formal algorithms. Provide argumentation why request execution can continue without loosing data consistency. Discuss the efficiency of your algorithm and the load put on the different nodes during the recovery process and how this might impact client response times. Use one to two pages.