School of Computer Science, McGill University

COMP-512 Distributed Systems, Fall 2015

Project Part II: Transactions and Concurrency Control

Due: 11 November, 2015; 10:00 pm

NOTE: Like the previous deliverable, this deliverable has again a maximum of 100 points, and if you do a solid implementation where everything is working and you give a good demonstration, then you will receive a maximum of 95 points. Of course, if something is missing or does not work correctly, there will be point reductions. 5 points are reserved for extras that are not specifically asked for in the description below. For example, an outstanding performance evaluation, or similar.

In this part you will implement transactions and distributed concurrency control. So far, we assume there are no crashes in the system. We describe the work to be done in two steps. First, add transactions to a single RM (assuming there is only a single RM in the system and clients connect to this single RM). Then, expand the concepts to the distributed system. This division is only a suggestion. At the end, only the distributed system is tested. But it might be easier to start with a centralized system and then distribute in order to understand the concepts and their interrelationships better.

1 Tasks

1. *Modify the lock manager so that it can convert locks.*
   We have provided a lock manager package/class. The operations that are supported include:

   - `lock(Xid, thingBeingLocked, read|write) throws DeadlockException`
   - `unlockAll(Xid)`

   Xid is an abbreviation for transaction identifier. The lock manager handles deadlocks by a timeout mechanism. A failed (deadlocked) lock operation throws an exception.
   You need to implement lock conversion: a transaction has a shared lock on object X and needs an exclusive lock on X. For example:

   ```
   lock(xid, foo, read);
   /* read foo */
   lock(xid, foo, write);
   /* write foo */
   ```

   Keep in mind that other transactions may have read locks on `foo`, so deadlock is possible. You would need to include error checking and exception handling. Enhancing the lock manager is independent of the other tasks.

2. *Add transactions to a centralized system* (one RM with many clients).
   a.) *The RM must support the methods* `start`, `commit`, *and* `abort`. All data access (read/write) operations implemented in the last part of the project are associated with a transaction.
   b.) *The RM has to lock data appropriately for each transaction, and unlock when the transaction commits or aborts.* Different locking granularities are possible.
   c.) *Add atomicity to the single RM.* Your system has to handle the abort of a transaction during normal processing. The necessary undo information can be maintained in main memory.

3. *Implement distributed transactions.*
   Since we do not consider failures and since with strict 2PL nothing can go wrong anymore once the commit is submitted, you only need to implement a one-phase commit. In general, we assume that the client decides on the transaction boundaries. That is, the client will call `start` and `commit/abort` methods that decide on which operations are going to belong to one transaction. The `id` parameter that is part of each method can be used to transmit the transaction identifier of the transaction to which the operation belongs.

   a.) *Locking*: You can have a centralized lock manager at the middleware server or lock managers at each site. Decide depending on what is more appropriate in your scenario.

   b.) *Implement a transaction manager (TM).* It must (at least) support the following operations: `start`, `commit`, and `abort`. Whenever the middleware server receives a start/commit/abort call by the client it forwards it directly to the TM. The TM coordinates the distributed transactions taking place across the several RMs involved. It must keep track of which RMs are involved in which transactions (i.e., whenever a request of transaction T is made to an RM, the TM must be informed that this RM is involved in transaction T: method `enlist` of the TM). The TM will maintain a list for each active transaction of which RMs are involved, and implement one-phase commit (simply tell the RMs that they should commit/abort the transaction).
   You have to decide on what kind of component the TM is. For instance, it can be an independent server on its own, but it could also be part (object) of the middleware server. In any case, the client never interacts directly with the TM, hence, no client interface will be provided for the TM.

4. *Handle client crashes by implementing a time-to-live (TTL) mechanism for transactions.*
   Every time an operation involving this transaction is carried out, the TTL is reset. If the TTL expires, the transaction should be aborted. This ensures that if a client abandons a transaction, the transaction is aborted after some time.

5. *Performance Analysis.*
   Make a performance analysis of the system during normal processing (because this is the most common case). Build a client program that can submit requests in a loop. The idea is as follows. The client contains several transaction types (e.g., an itinerary or a set of methods booking individual flights, hotels, etc.). Each transaction type could take input parameters to variate the requests. Then, the client contains a main loop that is executed many times (the number of transactions submitted during the test run). In each loop, a transaction type with input parameter is chosen (depends on what you want to analyze) and the transaction executed. The client program measures the response time of the transaction. In order to control the load submitted to the system, in transactions per second (tps), the client sleeps after each transaction for a certain time. For instance, if the load is 2 tps, the client should submit a transaction every 500 ms. Thus, if a transaction takes 200 ms to execute, then the client should sleep 300 ms before submitting the next transaction. Of course, if a transaction takes longer than 500 ms to execute then the client cannot create anymore a load of 2 tps. Also, to put some variation, the client typically does not submit exactly after 500 ms but chooses a time equally distributed within an interval $[500 - x, 500 + x]$. For all experiments, make sure that you run the test long enough to have stable results.

   a.) *Determine response times when there is a single client in the system.* For that, there is no need to sleep between transaction submissions as there is no concurrency in the system. Throughput does not play a role. Choose one transaction type that involves only one resource manager, and one transaction type that accesses all three resource managers. Both transaction types should have the same number of operations to make the two transaction types comparable in overhead. Analyze where the time is spent (e.g., middleware, RM, or maybe communication?)

b.) *Determine response times when there are many clients in the system.* Determine a suitable number of clients. Then determine the response time with increasing load. For instance, assume you have 10 clients and you want to start with a load of 1 tps, then every client should submit a transaction every 10 seconds. Then, for a load of 5 tps, every client submits a transaction every 2 seconds, and for a load of 10 tps, each client has to submit a transaction every second. Depending on the performance of your system, you have to figure out the number of clients and the throughput area that you want to test so that you show the system before and after saturation. Analyze your performance figures. What is the bottleneck in the system? Is it CPU, network, the middleware, or maybe the concurrency control (too many blocked transactions). If the latter is the case, you might want to experiment with a larger data set so that you don't have that many conflicts.

You have to provide similar milestone deliveries as for the first part:

- A 2-page documentation of your system architecture, special features of your implementation, problems encountered that you would like to mention, and how you tested the system.
- Furthermore, you have to present the performance evaluation: description of the testbed (client, measurement techniques, transaction types, etc.), performance figures, textual description of the figures and analysis of behavior.
- You have to provide a live demonstration to the TA.

~~2~~ Interface Additions: minimum requirement

The following methods must be provided at the appropriate servers (maybe at all servers, maybe only at specific servers).

1. `public int start();`
   Start a new transaction and return its id.
2. `public boolean commit(int transactionId);`
   Attempt to commit the given transaction; return true upon success.
3. `public boolean abort(int transactionId);`
   Abort the given transaction.
4. `public boolean shutdown();`
   Shutdown gracefully:
   - at RM: it is assumed that shutdown is only called when there is no transaction active at the RM. When a shutdown RM restarts, it does not need to perform any recovery;
   - at middleware layer: call shutdown of all RMs;
   - at TM: if relevant.