School of Computer Science, McGill University

COMP-512 Distributed Systems, Fall 2018

Project Milestone 2: Transactions and Concurrency Control

Report Date: November 7 @ 6 PM, Demo date: November 8/9

For this milestone of the project, you will implement distributed transactions and concurrency control, and evaluate the performance of your system. This milestone is still under the assumption that there are no crashes.

Question 1: *Implementation* (75 Points)

The implementation is divided into 3 steps (although you may skip #2 if you wish): (1) extend the lock manager; (2) implement centralized transactions (a single `ResourceManager`); and (3) implement distributed transactions. Only the distributed system will be evaluated as part of your submission, but a centralized system may be easier to learn the concepts.

Lock Manager

On `myCourses` you will find a `LockManager.tar.gz` tarball which includes a basic lock manager for your system. It supports:

- `lock(xid, identifier, read|write) throws DeadlockException`
- `unlockAll(xid)`

`xid` is an abbreviation for transaction identifier, and the identifier is the name of the lock to hold. The lock manager handles deadlocks by a timeout mechanism, which throws an exception on failure.

Your task: Implement lock conversion at the 2 places indicated by `TODO` notes. If a transaction `xid=1` has a shared lock on object `X` and requests an exclusive lock, it should convert *if no other transaction has a shared lock on `X`*. Note that deadlock is possible.

```
lock(xid, X, read);
/* read X */
[...]
lock(xid, X, write); // lock conversion, potential failure
/* write foo */
```

Centralized Transactions

Begin by adding transactions to a centralized system – this means there is one `ResourceManager` with many clients. All data operations (read/write) implemented in the last part of the project are associated with a transaction. To do so:

- **Locking**: The `ResourceManager` must lock data appropriately for each transaction, and unlock when the transaction commits or aborts. Different locking granularities are possible.
- **Transactions**: Add atomicity to the `ResourceManager`. Your system must handle the abort of a transaction during normal processing. You may store necessary undo information in main memory, or maintain updated information for each transaction in a local copy (one for each transaction).

We assume the client decides on the transaction boundaries using commands: `start`, `commit,<xid>` and `abort,<xid>` that you must add to the client. The client thus decides which operations are part of the transaction by passing the `xid` returned by `start` as its first argument. Full details on the new interface are found at the end of the document.

Distributed Transactions

Implement distributed transactions, assuming failures do not happen – this means that since we are using strict 2PL (and nothing can go wrong after commit), you only need to implement a 1-phase commit. To do so:

- **Locking**: Extend the lock manager to your distributed system. You may either have a centralized lock manager at the `Middleware` or lock managers at each site depending on your implementation.
- **Transactions**: Implement a transaction manager (`TM`) that coordinates transactions across all necessary `ResourceManagers`. The client thus sends commands: `start`, `commit` and `abort` to the `Middleware` which forwards it directly onto the `TM`. The `TM` should:
  - Maintain a list of active transactions
  - Keep track of which `ResourceManagers` are involved in a transaction (i.e. for each operation of transaction `T`, the `TM` must be informed of all necessary `RMs`)
  - Implement 1-phase commit: tell the appropriate `ResourceManagers` that they should commit/abort the transaction
  - Handle client disconnects by implementing a time-to-live mechanism. Every time an operation involving a transaction is performed, the time is reset. If the time-to-live expires then the transaction should be aborted

  You should decide how to implement the `TM`. It can be an independent server, or part of the `Middleware`. In either case it must never interact directly with the client.

As in the first milestone, your implementation is evaluated through a live demonstration!

Custom Functionality

The final part of the milestone again requires some type of additional functionality depending on your interests. You may also email the TA with any other ideas you may have for approval – use your creativity!

- Subtransactions for the itinerary command – your textbook has some relevant sections!
- An extra dimension on the performance analysis described below (easiest)

Question 2: *Performance Analysis* (15 Points)

Run a performance analysis of your system during normal processing. To do so we will have to:

1. Build a client that submits requests in a loop
2. Create a parametrized transaction type (e.g. an itinerary, or set of individual bookings)
3. For each iteration of the loop, execute a new transaction with different input parameters
4. At the client, measure the response time of the transaction

To control the load submitted to the system (e.g. 2 transactions per second), the client should sleep after each transaction. For example, if the desired load is 2 transactions per second, then the client should submit a transaction every 500 ms. If the request 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 a rate of 2 transactions per second is no longer possible. Additionally, to have some variation, the client does not submit after exactly 500 ms but rather a time equally distributed within the interval $[500 - x; 500 + x]$. For all experiments make sure to run the test long enough to have stable results.

Evaluation

1. Determine response times when there is a single client in the system. Note that this for this case, it is not necessary to sleep since there are no concurrent transactions – throughput does not play a role. Choose one transaction type that involves only one `ResourceManager`, and one transaction type that accesses all three. 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 (`Middleware`, `RM`, DB, or communication)

2. Determine response times when there are many clients in the system. To do so, first select a number of clients (your choice) and measure response times as the load increases. For example, assume 10 clients and a starting load of 1 transaction per second. Each client should then submit 1 transaction every 10 seconds. For an increased load of 5 transactions per second, each client submits 1 transaction every 2 seconds. Lastly, for a load of 10 transactions per second, each client has to submit 1 transaction every second. You should select the number of clients and transaction rates to show your system before and after saturation.

3. 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 many conflicts.

For this section, prepare a short analysis report (around 2 pages) containing: a description of the test bed (client, measurement techniques, transaction types, etc.), performance figures (tables and graphs), a textual description of the figures and an analysis of behaviour. Your analysis should be similar to the type discussed in class.

Question 3: Report (10 Points)

Write a short (2 page **maximum**) report detailing your architecture and design for distributed transactions. This should include a high level description of you strategy for locking, lock conversion, and transaction management (start/operations/commit/abort).

Interface Additions

The following methods must be provided at the appropriate servers. It is possible that there are additional exceptions in your implementation, and you should add `TransactionAbortedException` and `InvalidTransactionException` throughout the remote methods from the first milestone.

```
public int start() throws RemoteException;
public boolean commit(int transactionId) throws RemoteException,
        TransactionAbortedException, InvalidTransactionException;
public void abort(int transactionId) throws RemoteException,
        InvalidTransactionException;
public boolean shutdown() throws RemoteException;
```

The `shutdown` method should gracefully exit all servers. You can assume no active transactions, and no recovery is necessary.