### Distributed Transaction Application in C#

Project partners: Andrew Xu, andrewxu@cs  
Weili Liu, weilil@cs

**Features Implemented**

We have implemented all features outlined in the project description document. Following are a few highlights:

* *Lock conversion – to implement the lock conversion, we added a new intention lock - Update lock mode.* The Update lock tells the LM that this transaction will convert a read lock it holds to a write lock
  + Allow to wait for other transactions to release their read locks on the resource before the write lock can be granted.
  + Avoid other transactions to put write lock on after the transaction releases the read lock and before the write lock is in.
* *Two Phase Commit –* Use .Net thread pool to improve the currency. TM uses thread pool threads to send out the *Prepare or Commit* commands to all involved RMs asynchronously.
* *Scavenger Thread* *–* In TM, we also implemented a scavenger thread, which monitors all on going transactions. If WC starts a transaction, but forgets to commit in a timely manner, this thread will automatically notify all RMs to abort the transaction.
* *Data isolation* – According to our design, a transaction can read its own uncommitted data and only committed data from other transactions. A transaction cannot read its own deleted, but uncommitted data, while other transactions are still able to read it before the Commit.
* *Page files* **–** we implemented page files for the database storage. The page size is 4K. Each instance of the database has a manifest file which contains all tables it owns. Each table has a page table file which maintains the index for all records. The index module currently uses a hash function to distribute the records. In theory, it can be replaced with an industrial strengthen hash or B-tree index. Records can be located and updated individually in parallel transactions and some special logics are implemented to achieve the goal. For example, we use a single index file instead of multiple shadow files to maximize the parallelism.
* *Unit tests* –We added extensive unit test cases to ensure a good level of correctness and functionalities.

**Data Structures**

**TM structure**:

Inside the TM, we defined following dictionary to track all on going transactions.

*Dictionary*<Transaction, ResourceManagerList> \_resourceManagersEnlistedInTransactions

On Commit/Abort:

MyTM reads all RMs associated with a transaction from above dictionary, and then the two phase commit manager ***asynchronously*** notify the RMs.

The two phase commit manager internally maintains

*Dictionary*<Transaction, CommitedTransaction> committedTransactions = new *Dictionary*<Transaction, CommitedTransaction>();

It handles transaction logging and recovery base on the logged transaction information.

Log structure:

**TwoPhaseCommit.cs:**

This class is able to load the committed transaction log from disk and restore the states, then redo or undo the changes. Once TM asks this class to perform two phase commit, it will use .Net thread pool threads (ThreadPool.QueueUserWorkItem) to async call CommitedTransaction.StartCommit

TwoPhaseCommit class will use the CommitedTransaction class to perform the actual 2PC.

**CommitedTransaction.cs**

This class does the actual 2PC. It will use .Net thread pool threads (ThreadPool.QueueUserWorkItem) to start different threads to call individual RM. Once TwoPhaseCommit asks this class to perform the commit, it will start commit in Prepare and Commit phases.

These are the commit states in 2PC.

public enum CommitState

{

None, // the default state

Committed, // when the 2PC starts to commmit

Prepared, // All RMs are prepared

Done, // All RMs are commited

Rollbacked // Transaction rollbacked

}

**RM changes:**

In support 2PC, we augmented the RM’s interface as below:

public interface RM

{

void Enlist(Transaction context);

XaResponse Commit(Transaction context);

XaResponse Abort(Transaction context);

XaResponse Prepare(Transaction context);

…Omitted

}

public enum XaResponse

{

XA\_OK,

XAER\_RMERR, // something wrong

}  
Please note that although we have defined the above *Enum*, but we didn’t use all its members.

**Persistence structure**: we designed a special shadowed page files for atomic writes, please refer to the diagram below.

Each instance of the database has a manifest file that contains all tables it manages. Each table has a single page table file that maintains the index for all records. The index module computes a hash mode with the total pages to distribute new records. If a page is full, it will try the next page until it finds one slot or throws a database full exception.

Records can be located and updated individually in parallel transactions. Pages reside in the either data files of two. If Page N in file 0 is the active one, the Page N in the file 1 is the shadow one. They alternate between each other like a seesaw. There is no additional garbage file cleanup needed.

For simplicity, currently we limit the table page file to a single page. Each entry in this table contains:   
1.) A unique key for the record, like the primary key in a real database 2.) Page Index 3.) Row Index 4.) Active file Id 5.) Shadow file Id 6.) Dirty flag 7.) Transaction Id for prepared transactions. The data structure can more compact to save more space.

When an RM receives a Read command, it read the page table first to get page index, row index and active file id based on the key, then opens the corresponding file and reads the page and fetches the row.

When an RM updates a record, it will use the same approach to read the record, save the transaction id and the record in memory in a dictionary. Once it receives a Prepare command, it reads the pages, update them and save pages into the shadow file, meanwhile it also saves shadow file id and transaction id to the page table.

When an RM receives a Commit command, it set the shadow id as the active file id and removes the transaction ids. Similarly, when it receives Abort, it cleans up the shadow ids and transaction ids.

On recovery, the database will scan the page table to figure whether there is any prepared transactions. If it finds any, it will initialize its state and the transaction list in its memory, and waits for Commit or Abort command.

public interface ISimpleDatabase

{

void UpsertRecord(Transaction tid, string tableName, string key, Row record);

void DeleteRecord(Transaction tid, string tableName, string key);

Row ReadRecord(Transaction tid, string tableName, string key);

Dictionary<string, Row> ReadAllRecords(Transaction tid, string tableName);

void Commit(Transaction tid);

void Abort(Transaction tid);

void Prepare(Transaction tid);  
 …Omitted

}



**Committing a transaction**



**Recovery**

Recovery process has two start points. When TM loads, it will first check the log file to see if there is any transactions:

* If the state of the transaction is in None, Done or Rollback, we will not do any recovery.
* If the state of the transaction is Committed, 2PC will send Abort to all RMs and does not wait for response
* If the state of the transaction is Prepared, 2PC will keep trying to send Commit to all RMs and wait for the OK response.

**Extra Credits**

*Page files* **–** as already mentioned above, we have implemented a page and shadowing based database storage. A page is an atomic read and write unit. However, records can be located and updated individually in parallel transactions; thus, some special logics are designed to achieve the goal, which has been described in detail earlier in this report. The downside of this approach is that the implementation is complicated and proper locking is critical.

**Code Modules**

Code modules are self-evident from the initial template. We mainly augmented the MyTM , MyLM and MyRM.

* MyTM & TwoPhaseCommit : two phase commit transaction support and recovery logics, including eager logging transaction
* MyLM: implemented the locks and lock conversion
* MyRM: implemented the durable storage(tables, page, records, etc.) using a special version of shadowing

**Additional Information**

**Scenarios for demo** at least include following:

* Submit an itinerary with car, flight and hotel. Show the values are updated. Demo Read, write and commit.
* Add an itinerary, kill RM before commit. Restart the RMs and you can see nothing changed (still c1 state). Demo shadow copy works after failure
* Have T1 and T2 read R1, both shall get the result immediately.Have T1, T2 both write R2 concurrently, The result shall be correct. Demo locks and two transaction runs concurrently.
* T1, T2 write R1, T1 commit, T2 abort, see only T1's change in. Demo abort.
* All RMs returns prepared, except one fails to prepare, the transaction should abort
* All RM prepared, one RM dies before receiving Commit, on recovery, the RM should recover the transaction, WC shouldn’t notice this
* TM dies before receive prepare from all RM, on recovery, the transaction abort
* TM dies before receiving Done from all RMs, on recovery, TM should recommit.

**Update Lock example**:

The following are examples of how a sequence of operations converts in the implementation (ul1 means lock update lock on transaction 1, uul2 means unlock update lock on transaction 2, operations in () are consider one atomic operation)

**rl1[x], wl1[x] 🡪(the actual implementation) ul1[x], ,wl1[x]**

**rl1[x], rl2[x], wl1[x], url2[x]🡪 ul1[x], rl2[x], *(transaction 1 blocked here)*, url2[x], wl1[x]**

**rl1[x], rl2[x], wl3[x], wl1[x], url2[x], uwl1[x]🡪 ul1[x], rl2[x], *(transaction 3 blocked here)*. *(transaction 1 blocked here)*, url2[x], wl1[x], uwl1[x], wl3[x]**