**Design Document for Final Project**

Individual work: Maeda Hanafi

Dec 8, 2015

Code: <https://bitbucket.org/maedah/repcrec/src>

**File structure**

Test cases can be found in /TestCases. The main class is Test.java. The major components are:

1. Parser.java
   * Parses the input text. (See parse())
   * Controls the timestamp i.e. increments it at every newline. (See parse())
   * Iterates over all commands and sends each command i.e. split by semicolon or newline to the Transaction Manager. (See parse())
   * It parses the string into an array i.e. beginRO(T3) 🡪 [“beginRO”, “3”]. Such arrays are interpretable by the transaction manager into execution. (See parseNextInstruction())
2. TransactionManager.java
   * Receives the command details from the parser. (See processOperations())
   * It knows all the transactions, active and queued. It will not execute an operation for a transaction if it is not active i.e. queued. An incoming operation from a queued transaction will be added to the end of the queued operations of that queued transaction. (See processOperations())
   * Transactions are created by the manager and are immediately set to active. (See processOperations())
   * An operation is not sent to a transaction unless the transaction manager has acquired all locks and ensured that there are sites to read from.
3. Transaction.java: an interface defining the basic framework of a transaction. Implementations are RWTransaction.java (read-write) and ROTransaction.java (read-only).
   * Given that read-only transactions will use multiversion protocol, it will not need any locks and it will never be aborted.
   * Read-write transactions will use 2PL to access and modify the data. It may be queued or aborted based on wait-die protocol.
4. DataManager.java, Site.java, Variable.java, and Version.java are all related to storing the data.
   * The data manager controls access to the sites and knows which the variables are in which sites. All transactions have access to the data manager.
   * Site controls access to its variables. Any events i.e. commit, fail, recovery, are recorded in the site manager. It gives locks away to the transaction manager, which will give it to the requesting transaction.
   * Each variable has a list of versions containing information on the value, time, and originating transaction.
5. Log.java is the log located at each site recording each event.

**How to read the output**

When running the tool, the output has the following format:

<Class message is from>: <message>

Example:

Parser:begin(T1)

Interpretation: The parser class printed out the following message, where message is begin(T1) i.e. it read the command “begin(T1)”

Example:

TM: New RWTransaction: T1

Transaction is created with transaction number 1.

**Command details**

**Read operation:** A read operation for a read-write transaction will cause the TransactionManager to request read locks as needed from the sites that are going to be read from, TransactionManager.processRead(). If the transaction is read-only, then the transaction will use multiversion protocol and read the committed value before the transaction began, Transaction.processCommand().

If the read-write transaction gets a read or exclusive lock, it reads the latest value written to it i.e. it is not necessarily committed. Otherwise, the transaction will be queued or abort based on wait-die. And any other operations incoming for this transaction will be queued as well. It will have a chance at execution when an abort or commit is called for other transactions.

**Write operation**: A write operation can only be invoked by RWTransaction. (See TransactionManager.processWrite()). It figures out which sites to write to and requests locks as needed from those sites.

Locks can be upgraded or acquired, which is decided by 2PL, see Site.requestLock(), Site.upgradeRequest(). If locks were not acquired, wait-die protocol is used to decide whether to abort or wait, TransactionManager.processWrite(). Based on the timestamps at which the conflicting transaction began at, the transaction either waits or aborts.

Waiting means setting the transaction from active to queue, TransactionManager.queueTransaction(), and any operations incoming for this transaction will be added to a queue of operations, Transaction.addQueuedOperation(). The next time the transaction would have an opportunity to go back to the active state is when an abort or commit happens.

If the locks were acquired successfully, then the write operations would run. According to the all-available-copies algorithm, the write will affect all copies unless a site is down. A write will create a new version of the variable, (Variable.write()), and that version is set to uncommitted and toggled at commit time.

**Commit operation**: If the transaction was read-write, then it would have used 2PL, TransactionManager.commit(), and the validation would run, TransactionManager.validate, and locks would be released on a successful validation i.e. Site.releaseLocks(). If the validation went wrong, then it would be aborted. Otherwise, the transaction will be committed.

**Abort operation**: Abort initiates from transaction manager. Aborts never happen to read only transaction that use the multiversion read consistency protocol. The transactions are erased and the locks for those are released by the transaction. It notifies the lock managers of each site it has accessed to erase its locks. See TransactionManager.abort().

**Recovery**: the variables that aren’t replicated are set to allow reads and the variable that are replicated reset the pointers pointing to latest committed version and the latest version (that is possibly not committed). See TransactionManager.

**Fail**: On failure the site set all its variables to not allow reads. Its site is set to down. All locks are erased. See TransactionManager.