Replicated Concurrency Control and Recovery

Authors: Tushar Anchan and Ashutosh Mahajan

Introduction

The project provides a simulation of a Distributed Database, complete with multi-version concurrency control, deadlock detection, replication, and failure recovery.

Modules

Transaction Manager
- Multiversion Read Concurrency(For Read-Only transactions)
- Available Copies Algorithm(For distributed Read-Write Transactions)
- Failure and Recovery
- Commit and Abort

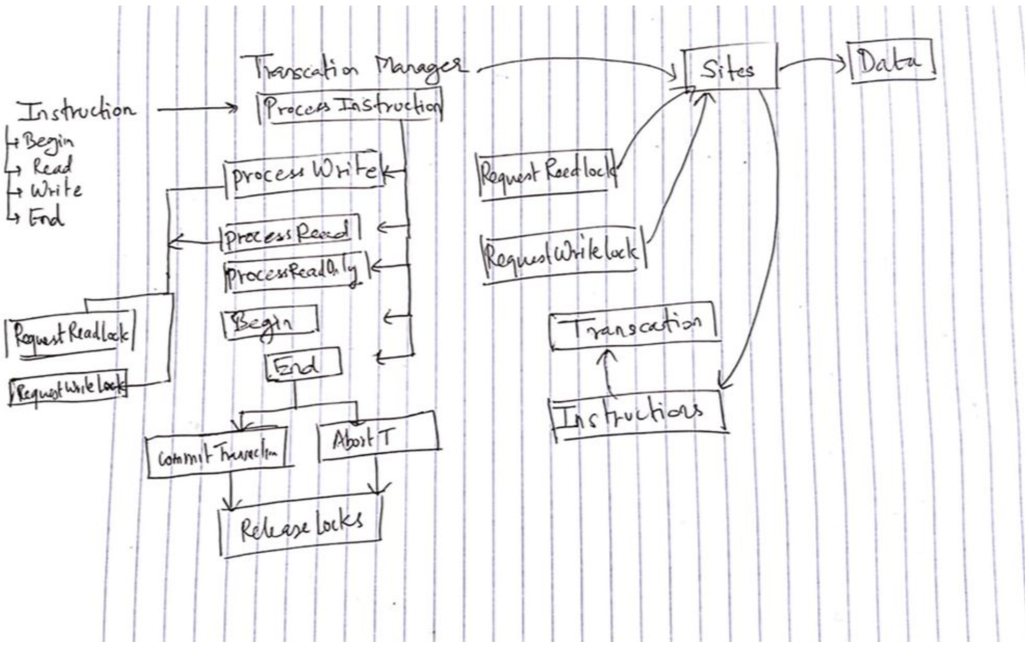
Deadlock Manager
- Cycle Detection
- Selection of Youngest Transaction to Abort

Input-Output and Execution of the Program

The program takes as input a file in the form of a test script which contains a sequence of operations. The file can be fed directly via the program or through the command line.
If the supplied test cases are used we only need to enter the file name.
The output is dumped onto the console after the program is executed.
The dump() functions are provided, which can be added to the end of the test script to get the values of the data items at individual sites. It can be used in 3 formats

dump() → Gives values of all data items at all sites
dump(i) → Gives values of all data items at site i
dump(xi) → Gives values of data item xi at all sites

Design Diagram:



Pseudocode:

```
ProcessInstruction(Instruction, Transaction id){
if(readonly) processReadOnly(Instruction, id)
If(write) processWrite(Instruction id)
if(read) processRead(Instruction id)
}

ProcessWrite() {
if (write-lock can be granted) : Set write-lock on appropriate sites
Else : add the Instruction to the waiting queue
}
ProcessRead() {
if (readlock can be granted) :
        Set readlock on the first available site
        Read the value to console
Else : add the Instruction to the waiting queue
}

ProcessReadOnly() {
        Read from the first available site from the cloned snapshot of sites created when the transaction begins
}

EndTransaction() {
        Perform deadlock detection
        (abort youngest transaction if true)
        Validate transaction by checking all instructions had appropriate locks on appropriate sites.
        Commit the transaction if validation was true or else abort it
        Release all locks from all sites this transaction holds.
}
```

Example Scripts

```
begin(T1)
begin(T2)
begin(T3)
W(T3,x2,22) //T3 gets writelocks on all sites at x2
W(T2,x4,44) //T2 gets writelocks on all sites at x4
R(T3,x4) //T3 is blocked as T2 holds locks at x4 on all sites
end(T2) //T2 commits and writes 44 at all sites at x4 then T3 is given the lock at the first available site at x4 and it reads the value 44.
end(T3) //T3 commits and writes 22 at x2 at all sites.
R(T1,x2) //T1 gets the lock at the first available site and reads the value 22.
end(T1) //T1 commits
```

begin(T1)
begin(T2)
R(T1,x3) // T3 gets the lock at site 4 and reads the value 30.
W(T2,x8,88) // T2 gets the lock at all sites at x8
fail(2) // site 2 is down and its lock tables are erased.
R(T2,x3) // T2 gets the lock at site 4 and reads the value 30.
W(T1, x4,91) // T1 gets the lock at all sites except 2.
recover(2) // site 2 is up and running
end(T2) // T2 is aborted because site 2 failed after it accessed and the lock info is lost.
end(T1) // T1 commits and writes value 91 at all sites except site 2.

Classes and Major Functions

These are a brief summary of the major functions in the classes. For a more detail description refer to the source code.

RepCRec(main class):
public static void main(String[] args)
This is the main method of the application. It takes in the filename of the test case to be run as an argument and parses the instructions in it. These instructions are sent to the transaction manager to process. Note that the test cases should be present in the directory test_scripts.

Data:
This class represents the variables that are stored in a site. It holds the index that refers to the variable(x1, x2 or so on) along with the value that is to be stored. The data is initialized to 10*(index) initially.

Site:
This class has an independent read lock table and a write lock table that holds transaction objects. The read lock table is a nested arraylist as read locks are to be shared. In addition, the site also has methods to check read or write locks on Data members, failing and recovering a site, and resetting the lock tables.

Transaction:
public void commitInstructions(ArrayList<Site> originalSites)
This method processes all write Instructions of the transaction by writing the value at sites accordingly. It also handles the case when a site that just recovered is being written to is to be allowed reads in the future.

Instruction:
This class holds the information of the Instruction to be processed: the data member to work on, the operation to perform(read or write) and the sites the transaction had access to when it was called(all available sites that were up).

Transaction Manager:
public void processInstruction(Instruction I, int transaction_id, boolean flag)
This method reads the type of the Instruction to process and calls the appropriate function to process it.

public void processWrite(Instruction I, int transaction_id, boolean flag, boolean addFlagToCheckRead)
The is the main method to process a Write Instruction. It first calls the request Lock function and grants the lock accordingly or blocks the Instruction in the waiting queue. Note that the write locks are granted to all available sites as per the algorithm.

public void processRead(Instruction I, int transaction_id, boolean flag, boolean addFlagToCheckRead)
Similar to processWrite. The difference here is that after the locks are granted, it also reads the value from the site and outputs to console. Note that the read lock here is granted to the first available site as per the algorithm requirements. If no site is available, the Instruction is blocked.

public void processReadOnly(Instruction I, int transaction_id)
This is read method to process ReadOnly transactions. It accesses the snapshot of the sites when the transaction was created and reads the values from it. This does not request any locks.

boolean requestWriteLock(int data_item, Instruction I, boolean flag, ArrayList<Site> sites)
This method checks if the Instruction can be given a write lock. It checks all sites of a variable for read locks and write locks and returns a boolean accordingly. This method also handles the case where a read lock is to be promoted into a write lock.

boolean requestReadLock(int data_item, Instruction I, boolean flag, ArrayList<Site> sites)
Similar to the requestWriteLock method, this method checks for any write lock conflicts at all sites of a variable. This method also handles the case where the lock is to be promoted into a write lock.

void endTransaction(int transaction_id)
This the method to process the “end” instruction for a transaction. It first calls a method to perform deadlock detection and then validates all the instructions for the transaction by checking if it has locks on all appropriate sites or if any site failed after an access. In addition, it calls methods to commit or abort the transaction and further processes the next instruction from the waiting queue.

private void commitTransaction(Transaction T)
This method commits the transaction by calling the commitInstructions method in the Transaction class and further calls a method to release all locks this Transaction holds.

private void abortTransaction(Transaction T)
This method aborts the transaction and calls a method to release locks the transaction holds.

public void releaseLocks(ArrayList<Site> originalSites, Transaction T)
This method releases all locks at all sites that the Transaction holds.

Graph:
void addEdge(String start_vertex, String end_vertex)
This method add an edge to a graph and vertices to the list of vertices. An edge is added whenever a transaction requests a data item.

void reverseEdge(String start_vertex, String end_vertex)
This method reverses the direction of an edge in the graph. An edge is reversed when a lock is given to a transaction on the data item it requested.

boolean detectDeadlock()
This method is used to detect any deadlock by checking whether a cycle is present in the resource-transaction graph

boolean dfs(Vertex current)
This method traverses the graph in a depth first search fashion recursively and segregates the vertices into 3 sets, white, black and gray. A vertex found more than once in the gray set indicates the graph contains a cycle and thus returns true, i.e. a deadlock is detected.

void removeEdge(String vertex)
This method removes an edge from the graph containing the specifies vertex. An edge is removed whenever a transaction aborts or commits, showing that it no longer exist in the execution environment.

Auxiliary Methods:
void clearAdjacentVertices()
This method clears the adjacency list of all vertices for a new iteration of deadlock detection

void updateAdjacentVertices()
This method updates the adjacency list of all vertices. An adjacency list changes when an edge or a vertex is removed from the graph and/or when an edge is reversed.

Edge:
Edge(Vertex start_vertex, Vertex end_vertex)
This constructor creates an instance of an edge object in the graph.

Vertex:
Vertex(String vertex_id)
This constructor creates an instance of a vertex

void addAdjacentVertex(Vertex v)
This method adds a vertex in the adjacency list of a particular vertex