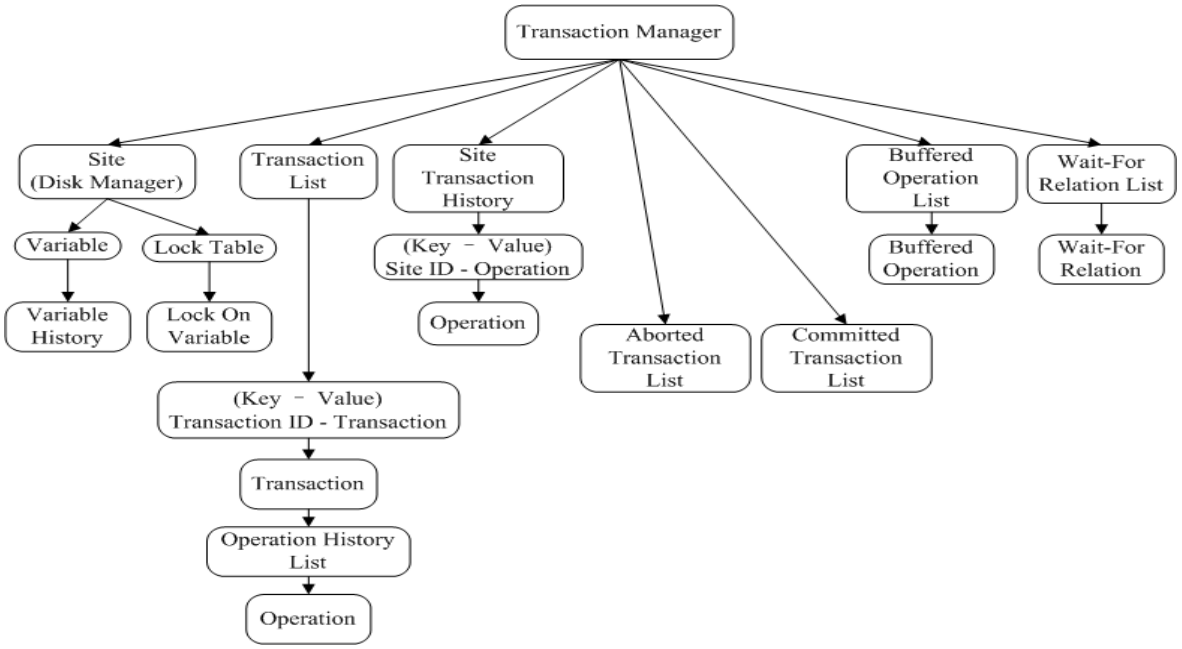
Project Report

Fang, Minda
Kim, Tae Young

Project Main Structure:

1. TransactionManager
   --- Core of this project
2. Site (Disk Manager)
   --- Used for constructing a site
3. Transaction
   --- Used for constructing a transaction
4. LockTable
   --- Used for constructing a lock table consisting of locks on variable
5. LockOnVariable
   --- Used for constructing a Read/Write lock on variable
6. Operation
   --- Used for constructing an operation recording detailed operation information
7. BufferedOperation
   --- Used for constructing an buffered operation
8. Variable
   --- Used for constructing a variable stored in one site
9. VariableHistory
   --- Used for constructing a variable value history stored inside a variable
10. WaitFor(Wait-For Relation)
    --- Used for constructing a wait-for relation between two transactions
11. Vertex
    --- Used for constructing a vertex in directed wait-for graph when loop detecting
12. Enum class used in this project



TransactionManager

It has:

1) 10 sites;
2) A map containing key(transactionID) – value(transaction) pair.
3) A internal clock which goes by one after executing one line of command;
4) A list for recording the committed transaction ID;
5) A list for recording the aborted transaction ID;
6) A list for containing buffered operations
7) A list for containing wait-for relations;
8) A list for containing to be aborted transaction ID;
9) A map containing key(siteID) – value(this site's transaction history) pair;

So this transaction manager mainly does four main jobs: **processing buffered operations, processing newcome operations, deciding whether some transactions can end, and checking deadlocks.**

Before every new line is executed, this transaction manager will check the buffered operation list and re-execute these buffered operations before executing new ones(this is reasonable since these buffered operations has time priority higher than these new ones. And for those buffered operations, if they can be executed successfully, then it will be okay. If not, they will be inserted into buffered operation list again.

After buffered operations are processed, then the transaction manager will execute new concurrent operations. The way we process the command end(Tx) is that we put this Tx into the list for recording the aborted transaction ID. After all new operations are executed, we will start to decide whether these transactions can commit, should wait or should abort.

After that, the transaction manager will check the wait-for relation list size and if the size is not zero(which means that some transaction is waiting for some transaction), it will check if there exists any deadlock. Detailed introduction will be in the next part of this report. If there exists, then abort the youngest transaction to until no deadlock is left. After no deadlock exists, this transaction manager will go to next round of **processing buffered operations, processing newcome operations, deciding whether some transactions can end, and checking deadlocks.**

- begin(Tx) & beginRO(Tx)
  These two commands will begin new transactions with the ordered transaction ID. In our project, we have a class named ***Transaction***, so every newly started transaction is an object. New created transactions will be put into a hashmap with a key(their transaction ID), so it will be very easy to check the detailed information about one transaction. And for the Transaction object, it has fields recording transaction ID, transaction type, transaction start-time(useful for RO transaction and deciding which one to abort when deadlock occurs), a list recording all operations history(which will be used when commit this transaction and report all its operations outcomes).

```
public class Transaction {
    private final int transactionID;
    private final TypeOfTransaction type;
    private final int startTime;
    private List<Operation> operationHistory;
    ...
    public void addToOperationHistory(Operation op) { this.operationHistory.add(op);}
}
```

Transaction object has method to add successfully executed operations into its operation history list.

- read

  For transaction read operation, we will show the pseudocode of this to elaborate it.

```
if this transaction is a read-only transaction
    then
        if the variable it wants to read is available on any one up site
            then
                it reads the last committed value before this transaction begins
                insert read-value, site number, read-time into OP history of both this site and this transaction
            else
                it should be put into buffered operation list
else // this transaction is a read-write transaction
    if there exist some buffered operations on the same variable
        then this transaction should be buffered and put a wait-for relation(wait for whom) in the wait-for list
        return
    if we cannot find any available copy of this variable on any sites
        then we should put this operation into the buffered operation list
        return
    if this transaction has already got a write lock on all available ones of this variable
        then
            read latest written value
            insert read-value, site number, read-time into OP history of both this site and this transaction
        else
            if exists any conflicting write locks on that variable
                then this operation will be buffered and put a wait-for relation(wait for whom) in the wait-for list
                else
                    get a read lock on the variable it reads from(if needed), read its current committed value
                    insert read-value, site number, read-time into OP history of both this site and this transaction
```

- write

  For transaction write operation, we will also show the pseudocode of this to elaborate it.

```
if there exist some buffered operations on the same variable
    then this transaction should be buffered and put a wait-for relation(wait for whom) in the wait-for list
    return
if we cannot find any available copy of this variable on any sites
    then we should put this operation into the buffered operation list
    return
if this transaction has already got a write lock on all available ones of this variable
    then
        write to all available variables
        insert write-value, site number, write-time into OP history of both this site and this transaction
    else
        if exists any conflicting write locks on that variable
            then
                get write locks on variables that are not locked by other transactions
                this operation will be buffered and put a wait-for relation(wait for whom) in the wait-for list
            else
                write to all variables in up sites, add a write lock on variables or upgrade read lock to a write one
                insert write-value, site number, write-time into OP history of both this site and this transaction
```

- end(Tx)

  When deciding if a transaction should commit, wait or abort, we have different judgements on read-only and read-write transactions.

  - For a read-only transaction, it can only commit if it has no buffered operations left and if all the sites if reads from is up at the time it commit. Otherwise, this read-only transaction should wait.

  - For a read-write transaction, if it has buffered operations, it should wait. If it has no buffered operations, it could commit. In this project, we will abort the transaction right away if it leads to a deadlock and is the youngest transaction in that cycle or due to failure of sites making it lose some locks. So we will not wait until end(Tx) to abort the transaction that should be aborted.

- fail(x)

  When a site fails, it will erase all its locks in its locktable. Meanwhile it will check all the locks in its locktable list, and then output affected transactions list(read-only transactions will not be affected) to the TM. Then TM will abort those affected transactions.

- recover(x)

  When a site recovers, it will make non-replicated data avaiable for read and write, and set replicated data only available for write. If any new writes comes, then replicated data will be available again.

- abort(transactionID)

  This is a method called when TM wants to abort some transactions to break a deadlock or due to transaction lock losing due to site failure.

When aborting some transactions, TM will do the following list of tasks:

1. Clear away all related buffered operations in the buffered operation list;
2. Clear away all related wait-for relations in the wait-for relation list;
3. Cancel off all the effect brought by this transaction (getting rid of all the locks brought by this transaction);
4. Add this transaction to already aborted transaction list;
5. Remove this transaction from current running transaction list;
6. Remove all related transaction history records in all the sites;

- deadLockRemoval(List<WaitFor> waitForList)

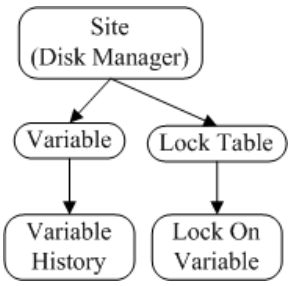
  We have our deadlock detection method which takes the input of the wait-for relation list stored in this TM.

```
/**Parts with the deadlock checking
        * 1. given wait-for lists
        * 2. generate adjacency matrix for directed wait-for graph
        * 3. while there exists cycle in that directed graph {
        * 4.     use DFS traversal to find one cycle
        * 5.     decide which one on the cycle to abort
        * 6.     export that one to abortList
        * 7.     revise the matrix
        * 8. }
 */
```

When deadlock checking, the transaction manager will look into its wait-for relation list, and generate an adjacency matrix for a directed wait-for graph derived by these wait-for relation list. Then depth-first-search will be adapted to find a back-edge in this directed graph. We know that a back-edge in a directed graph will indicates that this back-edge is a part of a loop. So after we get a loop which contains all vertices on that loop, we find out which transactions are represented by these vertices, find out the young transaction and record this transaction into the to be aborted transaction list. Then we will update the adjacency matrix and the have an updated directed wait-for graph and keep on loop detecting until no loop is found. Finally, TM will add all transactions that needs to be aborted to the to be aborted transaction list and then end transaction one by one in this list.

Site (Disk Manager)

Inside each site, there is a variable list consisting all the variables, and a lock table consisting all the locks that are on the variables in this site.

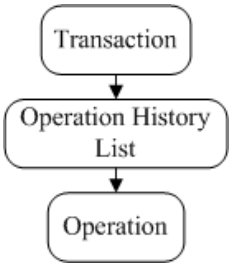


This disk manager has following methods:

- **BuildSite()** // build this site, create replicated or non-replicated copy of a variable
- **dumpOutput()** // output variable value in this site
- **ifContainsVariable(int variableID)** // return true if this site contains requested variable
- **ifThisVariableIsAvailable(int variableID)**
  // return true if requested variable is available for reading
- **writeToVariableCurrValueInThisSite(int variableID, int value)**
  // write value to variable's uncommitted value field
- **returnThisVariableCurrentValue(int variableID)**
  // return this variable's current uncommitted value
- **returnThisVariableValue(int variableID)** // return this variable's latest committed value
- **releaseThatLock(LockOnVariable lock)** // release the specific lock
- **CommitTheWrite(LockOnVariable lock, int commitTime)**
  // have uncommitted value committed
- **failThisSite()** // fail this site
- **recoverThisSite()** // recover this site

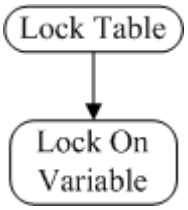
Transaction

Inside each transaction, there is a field for transaction ID, a field for transaction Type, a field for transaction start-time, and a field for a list of this transaction’s operation history.



LockTable

Inside each locktable, there is a list for locks on variables



LockOnVariable

Inside each lock, there is a field for transaction ID, a field for variable ID, a field for lock type.

Operation

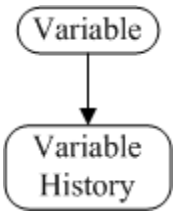
Inside each operation, there is a field for transaction ID, a field for type of operation, a field for site ID which records in which site this operation happens, a field for variable ID, a field for read/write value, a field for operation successfully executed time.

BufferedOperation

Inside each operation, there is a field for type of this buffered operation(blocked by transaction or variable unavailable), a field for transaction ID, a field for the waiting for transaction's ID, a field for variable ID, a field for the type of this transaction, a field for the type of this buffered operation, a field for read/write value, a field for this operation buffered time.

Variable

Inside each variable, there is a field for its ID, a field for its last committed value, a field for the available for reading state, a list for the committed value history, a field for its latest written value, a field for indicating if it is replicated or non-replicated.



VariableHistory

Inside each variable history, there is a field for the value, and time field for recording the committed time of this value.

WaitFor(Wait-For Relation)

For a wait-for relation, it tells which transaction is waiting for which transaction and what time is this first transaction start time.

Thus, it has three fields int from, int to, int time

Vertex

A vertex is used for constructing a vertex in directed wait-for graph when loop detecting. So it has field for its discover time in DFS traversal, its finish time in DFS traversal, its color indicating if it is already visited in one DFS traversal.

Enum class used in this project

```
public enum TypeOfTransaction{
    Read_Write,
    Read_Only
}
```

```
public enum TypeOfOperation {
    OP_READ,
    OP_WRITE
}
```

```
public enum TypeOfLock{
    Read,
    Write
}
```

```
public enum TypeOfBufferedOperation {
    VariableUnavailable,
    TransactionBlocked
}
```

* Used for indicating the type of this buffered operation, which tells why this operation is buffered

- VariableUnavailable - buffered due to unable to find any available variable in any up sites
- TransactionBlocked - buffered due to unable to have a lock due to transaction blocking

```
public enum Color {
    white,
    black
}
```

* Used for indicating if a vertex is visited in a DFS traversal

- Black – visited already
- White - not visited yet