PROJECT DOCUMENTATION

Ritika Motwani - rm6383

Project Introduction

Implement a distributed database, complete with serializable snapshot isolation, replication, and failure recovery.

Data

The data consists of 20 distinct variables x1, ..., x20 (the numbers between 1 and 20 will be referred to as indexes below). There are 10 sites numbered 1 to 10. A copy is indicated by a dot. Thus, x6.2 is the copy of variable x6 at site 2. The odd indexed variables are at one site each (i.e. 1 + (index number mod 10)). For example, x3 and x13 are both at site 4.
Even indexed variables are at all sites. Each variable xi is initialized to the value 10i (10 times i).
Each site has an independent Serializable Snapshot Isolation information. If that site fails, the information is erased.

Design

Snapshot Isolation working

1. Reads from transaction Ti: read committed data as of the time Ti began.
2. Writes follow the first committer wins rule: Ti will successfully commit only if no other concurrent transaction Tk has already committed writes to data items where Ti has written versions that it intends to commit.

Snapshot isolation serializable is to prevent such cycles from arising by judiciously aborting certain transactions. The only conflict edges between transactions that could involve concurrent transactions are rw edges.
Implementation:
At each variable x, if transaction T wants to read x from site s, record the start time for T and perform the read of the value produced by the last transaction T' that committed x on s before the start time of T and s was up the whole time before the start of T. Note however that if T itself wrote x, then the read should return the value that T wrote, because a transaction can always see its own writes.
If T writes at site s, then record the value written but do not allow anyone to see it until (and if) T commits.

When an end(T) occurs, for each access of T, determine whether T should abort:
- for available copies reasons (i.e. T wrote x on a site that later failed)

- for Snapshot Isolation reasons (i.e. some other transaction T' modified x after T began, T wrote x before or after T' committed and T' committed before the end(T) occurred) - first committer wins
- because committing T would create a cycle in the serialization graph including two rw edges in a row.

Language: Python

How to run

At the root of the project run:

python3 main.py <input file.txt>

I have created individual files for all 25 test cases and they pass for me locally.

In the reprozip after unpacking you can do:
python3 main.py input/input1.txt (or other files). The input directory has all the input test cases.

Classes and methods

1. VirtualClock
   All test-cases are sequential in nature, and there is no concurrent operation. Therefore an application level clock can be maintained, which simulates time. The methods and parameters are as follows:
   a. time = an integer parameter starting with 0.
   b. get_time(self) = method to return the current time.

2. Site
   a. __init__(self, idx, status)
      | Site constructor
      | :param idx: site number as index.
      | :param status: whether site status is up or down.
      | :param vars: all the variables on that site. For every variable we have this associated to it {"val": var.idx * 10, "committed_at": virtual_clock.get_time(), "uncommitted_at": None, "transaction_snapshots": {}}
      | :param recovery_history: when was the site last recovered. Initially all sites are recovered at the start time.
      | :param failure_history: add the time to this list when the site failed.
   b. __repr__(self)
      | Debugging logs
   c. fail(self)
      | Site class fail method. When the site fails we add that time to the failure history list of that particular site.

| The status of the site is changed to down.

d. recover(self)

| Site class recover method. When the site recovers we add that time to the recovery history list of that particular site.

| The status of the site is changed to up.

3. Var

a. __init__(self, idx, sites=[])

| Var constructor

| :param idx: variable number as index.

| :param name: x + str(idx)

| :param uncommitted_vals: Value of the variable within a transaction before the transaction ends.

| :param committed_version: version. initial or the transaction name.

| :param sites: List of sites that have this variable.

| :param last_write_success: Was the last write for the variable successful.

| :param read_blocked: Is read blocked for the variable.

b. read_var(self, transaction) -> Union[int, NoneType]

| Reads the var from the site(s)

| :return: value of the var

| For odd variables:

| Upon recovery of a site s, all non-replicated variables are available for

| reads and writes.

| For even variables:

| Regarding replicated variables, the site makes them available for writing,

| but not reading for transactions that begin after the recovery until a commit

| has happened. In fact, a read from a transaction that begins after the recovery

| of site s for a replicated variable x will not be allowed at s until a committed

| write to x takes place on s

c. write_var(self, transaction, val)

| Writes the var

| The function iterates over all sites. If the site status is up, it finds the variable that has to be updated,

| we update the transaction snapshot for it. The flow is site -> vars -> var name -> transaction snapshot -> (value,

| whether the site was updated since T began, update time, update attempt time, read blocked for that transaction).

| We cannot commit the write before the end, so we just update the transaction snapshot for that variable. In the transaction snapshot,

| the new values is added, site updated gets True, add the time for when the write happens, add the time when the attempt is done, and if the read is blocked for that var.
| If the site is down, in the transaction snapshot everything else remains same as previous, we only change the timer of whether attempt to update was done.
| The 'whether update is done' is to check if we try to write on a failed site and later check whether to abort the transaction for that.
| If at least one successful write was made we make the write success as True else print an error.

4. TransactionLogEntry
    a. __init__(self, transaction_identifier, op, variable=None, value=None)
        | TransactionLogEntry constructor
        | :param op: The operation (read, write, begin) in the transaction log.
        | :param variable: The variable that the transaction log is working with.
        | :param value: The value with the transaction log's operation
        | :param transaction_identifier: transaction name
        | :param timestamp: timestamp associated with the log.

5. Transaction
    a. __init__(self, name, last_seen_commits)
        | Transaction constructor
        | :param name: transaction name
        | :param last_seen_commits: latest write commit by every variable
        | :param state: whether active or committed
        | :param start_time: start time of a transaction
        | :param committed_at: when the transaction was committed
        | :param log: logs (of type TransactionLogEntry) list of a transaction
    b. log_begin(self)
        | Append a log entry to the log list parameter of Transaction. The log entry constructor requires transaction name,
        | the operation which begins in this case.
    c. log_read(self, variable)
        | Append a log entry to the log list parameter of Transaction. The log entry constructor requires transaction name,
        | the operation which is read in this case, and the variable that has to be read.
    d. log_write(self, variable, value)
        | Append a log entry to the log list parameter of Transaction. The log entry constructor requires transaction name,
        | the operation which is 'write' in this case, variable to write to with the value that has to be written.

6. DataManager

a. __init__(self)
   | DataManager constructor
   | :param x...: All variables are initialized in the main datamanager class with the type as class Var
   | :param s..: All sites are initialized here as class Site type. The initial state is UP for all sites.
   | :param sites: List of all initialized sites.
   | :param variables: List of all initialized variables.
   | :param variables_map: dictionary of variable name with initialized correct variable.
   | :param sites_map: dictionary of site index with initialized sites.
   | :param transactions_map: In the datamanager class have a dictionary of all transaction names with the transaction.

b. attempt_transaction_commit(self, transaction: transaction_manager.TransactionManager.Transaction, transaction_logs)
   | When end Transaction happens this function is called.
   | We find if because of any conflict the transaction should be aborted.
   | Iterate over all variables. For every variable iterate over the sites it is on.
   | Cases:
   | Case 1. For the conflict about site failing after a transaction attempts to write to it and ends after the failure.
   | We check for every failure of the site and
   | if it happened after the transaction attempted to write to it, we abort the transaction.
   | Case 2. We first check if the current transaction snapshot for that variable has an associated write.
   | That means that the current transaction has attempted to write on that variable in its course.
   | we check if for the variable the last seen commit is the same as the variable's committed version of the transaction.
   | If it is not, that means another transaction committed to it before it could. So by the logic of first committer wins
   | the transaction is aborted.
   | Case 3. We check if in the transaction snapshot for a variable on a site the read got blocked because
   | of failed sites, and even if it recovered no one wrote to it, we aborted the transaction.
   | Then we exit the loop. For the last case we have to check if the serialization graph has a cycle.
   | Case 4. We call the dependency graph class function to check the cycle and pass the transaction name, logs and map.
   | If the create cycle function returns True we abort the transaction with the cycle reason.

| If none of the cases are True, we update the value of the variable, the committed at becomes the new time
| and the committed version of the variable has the current transaction name.

c. dump(self)
| The dump function iterates over every site and
| prints the variables and the variable values associated with that site.

d. get_last_commits(self)
| Get the latest commit variable commit version.

e. get_logs_by_var(self, transaction_logs)
| Get logs at variable level for every transaction. The logs are sorted by timestamp.
| They will be needed to detect a cycle in the graph of transactions.

f. get_sites(self, idx)
| If the index is even then the get site has to return all sites.
| If the index is odd then only one site has that variable, so return the correct site.

g. handle_fail_site(self, site)
| When a site fails the site class' fail method is called.

h. handle_recover_site(self, site)
| This function first calls the recover method associated with the site class.
| Then it iterates over all sites to find what transactions have been read blocked because of
| no site being up (even variables) and unblocks them to avoid aborting the transaction if they have not ended.

i. initialize(self)
| Iterate over every variable in the parameter of the datamanager class.
| For every variable send the variable index to the getsite function
| and find the site where the variable is at.
| For the sites returned (all in case of even, 1 in case of odd) for every site.vars[var.name]
| initialize with the value (index * 10), committed_at: current time, uncommitted_at: None
| and an empty dictionary for transaction snapshots as there are no snapshots currently, for any variable.
| This is just an initialisation function to start before reading the transactions.

j. register_transaction_begin(self, transaction)

| It is called when the transaction begins.
| In the transaction map we add the new transaction name and the class object.
| Iterate over all the sites and if the site status is UP, we iterate over all variables at that site.
| To every variable we initialize the transaction snapshot object. We add the tuple:
| (value, False(not writing), time, time, False(reading not blocked)). If the site is down then
| the tuple becomes (None(no val as site is down), False(not writing), None, None, False(reading not blocked))

k. register_transaction_read(self, transaction, varName)
| It prints the value read by the transaction when executing.

l. register_transaction_write(self, transaction, varName, value)
| This function first calls the write variable function associated with the variable class.

7. TransactionManager
   a. __init__(self, data_manager)
   | TransactionManager constructor
   | :param data_manager: The data_manager class object associated here with this class object.
   | :param active_transactions: when the transaction begins add all transactions here in the dict.
   | :param states: unused debugging var

   b. get_transaction_states(self)
   | For every active transaction it adds the logs to the states.
   | It is required to call the transaction commit function.

   c. handle_begin_transaction(self, transaction)
   | Begin transaction function. It adds the new transaction to active_transactions.
   | And this in turn calls the data manager with the register_transaction_begin function.

   d. handle_end_transaction(self, transaction)
   | End transaction function. It calls the attempt transaction function which checks if the transaction
   | should be committed or aborted. If the outcome to commit is True, the transaction state is made COMMITTED.
   | Else it prints the abort transaction part.

e. handle_read(self, transaction, variable)
| 1. get active transactions.
| 2. Check if some active uncommitted transaction has written this variable prior to it, if yes, it's a rw dependency from
| that transaction to this one.
| :return: None

f. handle_write(self, transaction, var, val)
| 1. get active transactions.
| 2. Add logs which will later help to check ww edges.
| 3. register the write with database manager class that helps to add transaction snapshots, helping check write first logic.

8. Node
   a. __init__(self, transaction)
   | Node constructor
   | :param transaction: transaction as the node
   | :param ww_edges: ww edges associated
   | :param wr_edges: wr edges associated
   | :param rw_edges: rw edges associated
   | :param depends_on: node depends on what node, to get dependencies
   | :param uncommitted_rw_dependencies: debugging param
   | :param committed: debugging param

9. DependencyGraph
   a. __init__(self)
   | DependencyGraph constructor
   | :param nodes: nodes for a graph
   | :param edges: a set of graph edges for rw, ww, wr

   b. will_create_cycle(self, transaction, logs_by_var, transactions_map)
   | The inputs comprise of the current transaction that
   | has called the will_create_cycle function to check if it
   | can abort without forming a cycle. The logs by variable and the transaction map (name: Transaction).
   | It first creates a node for the current transaction, then we have a loop for adding edges.
   | It iterates over the logs by variables. log of logs of variables. If the log is associated with the current transaction then
   | has_current_transaction_began is set to True and then we check the operation of the current transaction.
   | Else if the other transaction has a read operation start appending to the read_write dependency array.

| After the inner loop is done, we check if not has_current_transaction_began or (not current_transaction_write)

| to see the rw dependency added is really a dependency or not If not, we clear the list.

| In a similar way we check for the ww dependencies.

| In the set of edges we add all the rw and ww dependencies with a label rw and ww. Then we create a graph.

| We use an adjacency list for it. Then to check the cycle we have these two steps:

| Step 1. Check if the graph has two consecutive rw dependencies, then a deadlock cycle should be removed.

| Step 2. Check if the new transaction causes a cycle in the graph and whether it should be aborted.

Other than these classes we have a main file. It is the starting file of the program that initiates all classes and takes a filename as an input, parses the file based on the functions provided in the project and helps generate results from those inputs. The file is *main.py*

Based on the line parsed it calls the related function from data manager or transaction manager files.

The **summary** of working of the code is as follows:

1. The main file reads the input file. The input sent is **1 test case at a time**.
2. It **parses the file line by line** and calls the necessary functions.
3. For begin it calls the **handle_begin_transaction** function in the transaction manager class.
4. For reading it calls the **handle_read** function. Every site-> variable has a **transaction snapshot associated with it**. It only changes that snapshot till the end transaction is called and the transaction is committed. So the read, reads the value of the variable from that snapshot and not the committed values. If the same transaction writes to the variable then only the snapshot associated updates.
5. For write, same as the case for read we **make changes to the transaction snapshot associated with the variable**, we update the local value, when the attempt to update was made, whether updated booleans. This happens only if the site is up, if it is down other updates take place.
6. For **fail and recover**, the site status is changed and for recovery the read that was blocked due to a site failure and no writes is changed incase the site had a write before the transaction and was working well before the transaction.
7. The **dump function** prints all variables associated with every site.
8. The **end transaction** calls the **handle_end_transaction** function. It has the following responsibilities:
    a. Check the following **conflicts**:

        i. Not the first write committer.
        ii. Cycle due to two rw dependencies happening consecutively.
        iii. Site failed after the transaction was written on that site.

   b. If any of the **conflicts** happen the transaction is **aborted.**
   c. If there is **no conflict** then the transaction is **committed** with the current virtual clock timestamp and the values it has updated with the writes in that transaction that are stored in the local transaction snapshot are merged with the real values of variables on every site.

Class Diagram

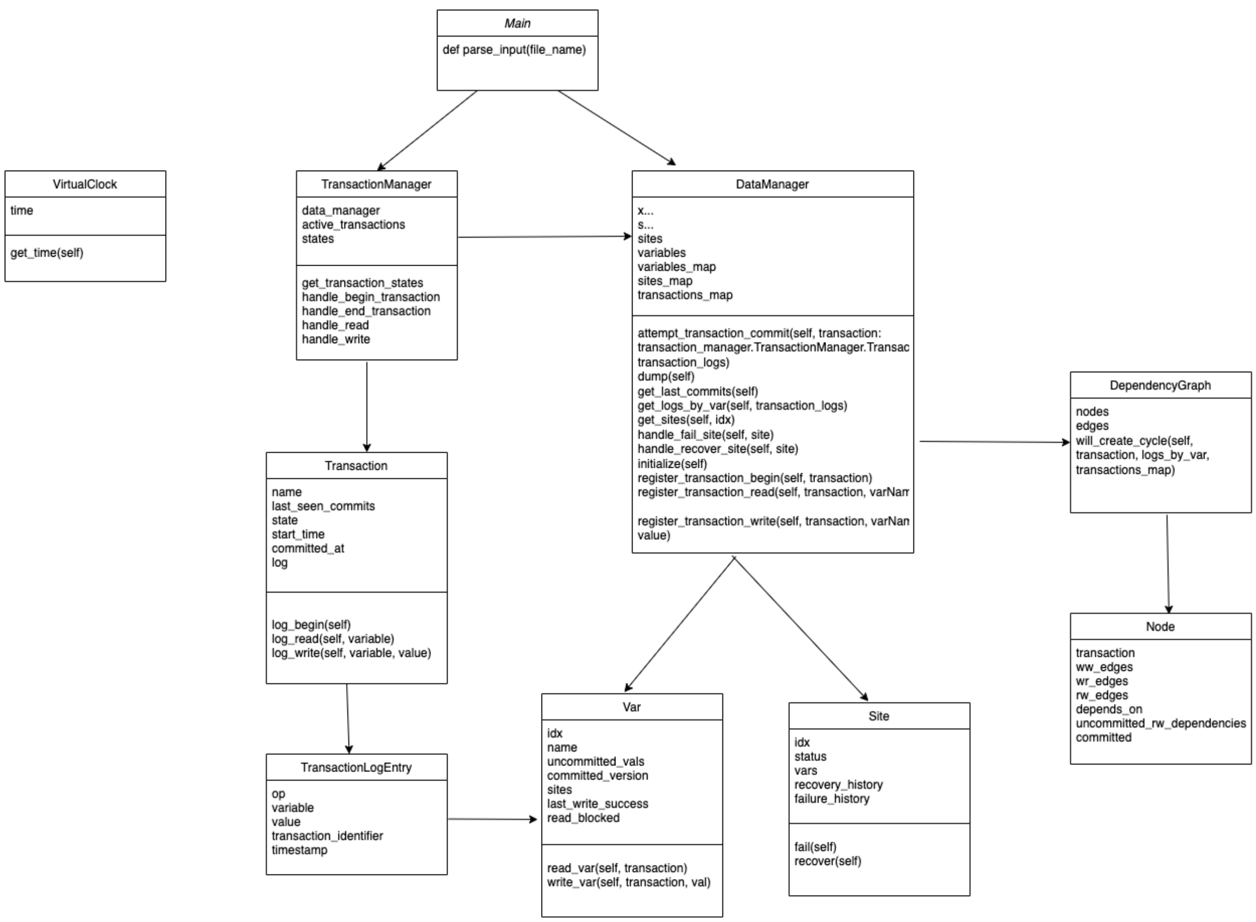

Figure 1: Class Diagram of the application

Conclusion

The code shows how multiple transactions are handled with serialization snapshot isolation and the cases where we have to abort a certain transaction because of conflicts.