Advanced Database Systems
Project Design Document
Nisha Visweswaran (nv740) and Christian Fernandez (cf86)

Overview:

The Replicated Concurrency Control and Recovery Database Project is developed using Python 2.7.

This project also implements multi-version concurrency control (keeping track of when the value of a variable was updated), deadlock detection (using graph algorithms and the youngest transaction is aborted), replication (odd variables are not replicated, but even variables are replicated), failure recovery (when sites fail, the lock table is cleared and when site is recovered, the replicated variables are marked not readable until a write commit happens), available copies algorithm (only sites which are up are read from[if variables are readable] or written to) and strict 2 phase locking (locks are obtained on all sites for both read/write and locks are cleared when transaction is aborted or committed).

For the Distributed Database, there is a single TransactionManager. There are multiple sites and each site has its own DataManager which takes care of its own list of Variables. From the input file (test case), each line is parsed into an Instruction object. From the instructions, a Transaction Object is created (when a transaction begins) and is managed by the TransactionManager. When a lock is obtained, a Lock Object is created using the Transaction that obtained the lock and the Variable Object on which the lock is obtained and maintained by the DataManager of the site. There is a Clock object which is used for each tick and to obtain the time for the execution of any instruction.

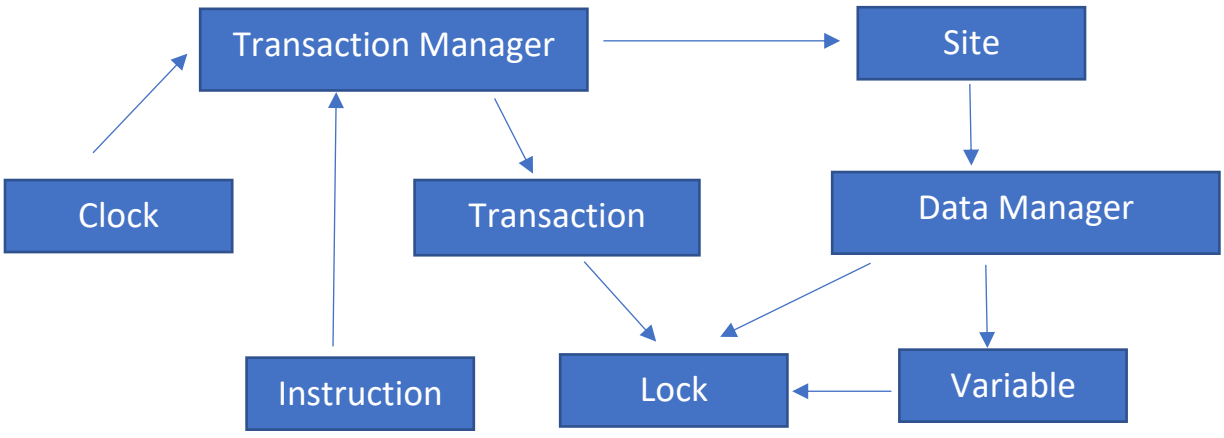
Components

The major components of this project are –

1. TransactionManager
2. Site
3. DataManager
4. Variable
5. Instruction
6. Transaction
7. Lock
8. Clock

There are some utilities classes and objects as well, for parsing and logging.

Interconnection Diagram



Main Classes

1. **TransactionManager**:
   - **a.** The TM keeps track of the operations and the status of the transactions. The TM also keeps track of the sites that are part of the distributed database and calls the corresponding site's DataManager to perform the read/write operations, and to keep track of locks, variables. It will relay the site related instructions to that site. The TM also performs deadlock detection and kills the youngest transaction.
   - **b.** **Interacts with**: Site (and the site's DataManager), Transaction
   - **c.** **Variables**:
     i. transactions: dictionary (key: transaction identifier, value: transaction object)
     ii. sites: dictionary (key: site identifier, value: site object)
     iii. site_to_variable_map: dictionary to map each site to the variables it contains.
     iv. variables_to_site_map: dictionary to map each variable to each site that contains it.
     v. waiting_transactions_instructions_map: dictionary of waiting transactions
     vi. blocked_transactions_instructions_map: dictionary of blocked transactions (key: blocking transaction id, value: list of instructions that are blocked)
     vii. sites_transactions_accessed_log: dictionary of transaction identifier to the set of sites it accessed (obtained locks on)
     viii. sites_transactions_read_write_log: dictionary of transaction identifier to the set of sites the transaction read from or wrote to – used in aborting transaction during fail
   - **d.** **Functions**:
     i. execute(instruction): This is called by the Main class. This ticks the clock by 1, and reruns any waiting instruction and then executes the current instruction parsed from the file.
     ii. execute_Instruction(instruction): This method calls the corresponding action in the instruction.
     iii. begin_transaction(instruction): This begins the transaction, i.e. inserts the transaction object in the transactions dict. For read only transactions, it also creates the snapshots of the variables.
     iv. end_transaction(instruction): This ends the transaction. If the transaction is not aborted, then it commits the transaction.
     v. commit (instruction): This updates the status of the transaction to committed. For each site that the transaction accessed and is UP, it calls the site's data manager's commit method and prints that the transaction is committed. It also does some cleanup and reruns the instructions (and the transaction) that were blocked by the transaction that is being committed.
     vi. abort(transaction_ident, site_index): This updates the status of the transaction to aborted. For each site that the transaction accessed, it calls the site's data manager to clear the locks and entries variables. It also does some cleanup and reruns the instructions (and the transaction) that were blocked by the transaction that is being aborted. If site index is present, the transaction aborted due to site failure, else due to deadlock, and prints that the transaction is aborted and also the reason why it was aborted.
     vii. rerun(instructions): This reruns instructions in the waiting_transactions_instructions_map, if any. If instructions list is passed, then it reruns those instructions.

viii. read(instruction): This executes the read operation. For read only transactions, it reads the variable value from the snapshot, else it tries to obtain read locks (using that site's data manager) on all the sites the variable is present, (provided the site is up and the variable is readable). If all the locks are obtained, the transaction reads from the first available site (value is printed), else is blocked. If there is no available site, the transaction goes into waiting state.

ix. write(instruction): This executes the write operation. TM tries to obtain write locks on all available sites (through the site's data manager). If any write lock is not obtained, the transaction is blocked and an entry is inserted/updated in the blocked_transactions_instructions_map with the transaction id that has the lock, and the current instruction which is blocked. If write locks are obtained on all the sites, successful write happens (provided transaction commits). If there is no available site, then the transaction is in waiting.

x. dump(instruction): Prints the values of the variables (and calls the site's data manager's dump method)

xi. fail(instruction): Calls the site's fail method to fail the site and aborts the transaction (which is not committed) that performed a read/write operation on the failed site.

xii. recover(instruction): Calls the site's recover method to recover the site. In case the variable is not present in the readonly transaction, we update the snapshot (this case is possible in case of non-replicated variables and the site was down when the snapshot was taken)

xiii. check_for_deadlock(), create_graph(active_transactions_list), dfs_check_cycle(vertex, graph, visited_set, done_set), get_transaction_id_to_kill(visited_set): These methods are used in deadlock detection. A graph dictionary is created using the active_transactions_list and the blocked_transactions_instructions_map (when the transaction in blocked). The graph has a transaction identifier as the key, and the list of transactions that is blocking that transaction. With this graph, a dfs is performed to detect a cycle. If a cycle is detected there is a deadlock and we get the youngest transaction from the visited set and abort that transaction.

2. **Site**:
   a. Site object to manage the sites
   b. **Interacts with**: DataManager of the site
   c. **Variables**:
      i. identifer: site id
      ii. status: site status (UP/DOWN)
      iii. create_time: time when the site was created/recovered
      iv. data_manager: Site's data manager
   d. **Functions**:
      i. dump(): This method returns the variables in the data manager
      ii. recover(time): This updates the status of the site to UP, and the variables' readable status is updated through the data manager if the variable is replicated.
      iii. fail(): This updates the status of the site to DOWN and clears the locks and entries variables of the data manager.

3. **DataManager**:
    a. This manages the variables, locks, and maintains the history/log of the variable updates.
    b. **Interacts with**: Variable, Lock
    c. **Variables**:
        i. entries: dictionary (key: transaction_ident, value: dictionary (key: variable identifier, value: variable object)) – This keeps track of the variable updates by each transaction.
        ii. variables: dictionary (key: variable identifier, value: value object)
        iii. locks: dictionary (key: variable identifier, value: list of lock objects on that variable)
    d. **Functions**:
        i. obtain_write_lock(instruction, transaction): This method is used to check if a write lock can be obtained. This checks the locks dictionary and if there are no locks on that variable, write lock can be obtained – create a new lock and update the locks dictionary. If there are any locks on that variable, and the transaction holding the lock is not the same transaction requesting the lock, then write lock cannot be obtained, if it is the same transaction, then write lock can be obtained and the lock type is updated to write lock. If write lock is not obtained, then the transaction identifier already holding the lock is returned, else none is returned.
        ii. obtain_read_lock(transaction, instruction): This method is used to check if a read lock can be obtained. This checks the locks dictionary; if the variable is not readable, or if there is a write lock on that variable (not held by the same transaction), then read lock is obtained, and the locks dictionary is updated. This method returns a Boolean value.
        iii. read(transaction, instruction): This method reads the value or the last updated value(if the same transaction holds a write lock is issuing a read) of the variable.
        iv. get_write_lock_value(transaction, instruction): This method returns the latest value written by the transaction if it holds a write lock.
        v. get_write_lock_owner(instruction): This method returns the transaction identifier that holds the write lock (if there are any write locks on that variable)
        vi. write_new_data(time, variable_ident, new_value, trans_identifier): This method updates the new value of the variable in the entries dictionary.
        vii. commit(time, transaction): This updates the value of the variable if there are any write locks on that variable, and then clears the locks held by that transaction.
        viii. clear_locks(transaction_ident): This method removes the any locks that the transaction holds.
        ix. clear_entries(transaction_ident): This method removes any entry in the entries dictionary with key as that transaction ident, when the transaction is aborted.

Other Classes

1. **Variable**:
    a. Variables:
        i. index: var_id
        ii. identifier: “x”var_id (Example: x1)
        iii. replicated: True if the variable is even, False if it is odd
        iv. readable: True (by default, updated to False when the site recovers and variable is replicated)
        v. value: 10*var_id (by default, and updated when commit happens)
        vi. written_values: dictionary (key: time, value: value) – log of all the new written values
2. **Lock**
    a. Variables:
        i. lock_type: type of lock (READ/WRITE)
        ii. transaction: transaction object holding the lock
        iii. variable: variable object on which the lock is held
3. **Instruction**
    a. Variables:
        i. instruction_type: type of instruction (BEGIN/BEGIN_RO/DUMP_ALL/DUMP_SITE/ DUMP_VAR/END/FAIL/ READ/RECOVER/WRITE)
        ii. variable_identifier: variable identifier in the instruction (for read/write/dump variable)
        iii. site_identifier: site identifier in the instruction (for fail/recover/dump site)
        iv. transaction_identfier: transaction identifier in the instruction (for begin/ beginRO/read/write/end)
        v. value: value to be updated for a write instruction
4. **Transaction**
    a. Variables:
        i. index: transaction index (Example: 1 if T1)
        ii. identifier: transaction identifier (Example: T1)
        iii. transaction_type: type of transaction (READ_ONLY or READ_WRITE)
        iv. start_time: time when the transaction began
        v. end_time: time when the transaction ended (committed) or aborted
        vi. state: status of the transaction (ABORTED/BLOCKED (if locks cannot be obtained)/ COMMITTED/RUNNING (default)/WAITING(if no available site)
5. **Clock**
    a. Variables: time – initialized to 0
    b. Functions: tick() – increment time by 1