Project Report

Sai Preetham
sb9509

Nikhil Kommineni
nk3853

1 Design

1.1 ValueType

The ValueType class is designed to store the value of a specific variable at a given timestamp. It also records the ID of the transaction (txnId) responsible for writing that value.

```
class ValueType {
public:
    int value;
    int timestamp;
    int txnId;
};
```

1.2 Operation

The OperationType enum is used to categorize the type of operation based on the initial part of the input string. The Operation class represents an operation and stores its details depending on the OperationType, it stores the transaction ID if the operation is not FAIL, RECOVER, or DUMP, the variable it acts upon for READ or WRITE operations, the value involved in a WRITE operation, the timestamp when the operation was received.

```
enum OperationType {
    BEGIN, END, READ, WRITE, FAIL, RECOVER, DUMP, UNKNOWN
};

class Operation {
public:
    OperationType op_type; // Type of operation
    int transactionId; // ID of the transaction
    string variable; // Target variable (for READ/WRITE operations)
    int value; // Value to be written (for WRITE operations)
    int timestamp; // Timestamp of the operation
};
```

1.3 Transaction

The Transaction class is designed track the lifecycle of a transaction. The transactionID uniquely identifies the transaction, while the startTime and commitTime record when the transaction begins and when it commits if successful. The current status of the transaction is stored in the currentStatus, which can indicate whether the transaction is active, committed, or aborted. If the transaction is aborted, the reason for the abort is stored in the abortReason. The class maintains a record of all previously executed operations in pastOperations and tracks any operation currently in a waiting state through waitingOperation. Additionally, keeps tracks of variable access for pre-commit checks. For write operations, the writeSites attribute lists all the sites available for writing for that operation.

```
enum class TxnStatus {
    ACTIVE,
    COMMITTED,
    ABORTED
};

class Transaction {
public:
    int txnId; // Transaction ID
    int start_ts; // Start time of transaction
    int commit_ts; // Commit time of transaction
    TxnStatus status; // Status of transaction
    string reason_4_abort; // Reason for aborting transaction
    // Maintains the first access details of each variable
    map<string, pair<int, ValueType> > var_access_map;
    // Tracks the variables written by the txn, useful at commit time
    map<string, bool> is_written;
    // Tracks the variables read by the txn, useful for SSI pre-commit checks
    map<string, bool> is_read;
    // Tracks the active sites to write for an operation
    map<Operation, vector<int> > active_sites_for_write_op;
    vector<Operation> past_operations; // Past operations of this transaction
    Operation* waiting_operation; // Waiting operation
    // Tracks current state of the transaction
    map<string, int> current_state;

    Transaction(int txnId, int startTime);
    // Core transaction operations
    void add_operation(Operation op);
    void set_waiting_operation(Operation* op);
    void commit(int timestamp);
    void abort(string reason);

};
```

1.4 Data Manager

The Data Manager manages site-specific data, monitoring the site's availability and the accessibility of variables. It maintains the current values of all variables and tracks their historical snapshots, providing a record of how each variable has evolved over time.

```
class DataManager {
public:
    int id; // Site ID
    bool isUp; // Tracks the site status
    // Tracks if each variable is accessible
    map<string, bool> accessible;
    // Current values for variables in this site
    map<string, ValueType> values;
    // Historical snapshots of variables
    map<string, vector<ValueType> > snapshots;

    DataManager(int site_id); // Constructor
    bool is_site_up();
    ValueType read(string variable, int at_ts);
    void commit(string variable, ValueType value, int commit_ts);
    void take_snapshot(string variable);
};
```

1.5 Transaction Manager

This central class acts as the core of the system, maintaining the integrity and flow of operations. It performs pre-commit checks for all transactions to ensure consistency. The class handles read, write, begin, end, recover, fail, and dump operations for sites. It ensures that each transaction is properly tracked and managed throughout its lifecycle.

```
class TransactionManager {
public:
    map<int, DataManager*> sites;
    map<int, Transaction*> transactions;
    map<int, vector<Operation> > site_history;
    vector<Transaction*> committed_txns;
    vector<Operation*> waiting_operations;

    TransactionManager();
    static TransactionManager& get_instance();
    Transaction* get_transaction(int transactionId);
    void begin_transaction(int transactionId, int timestamp);
    int read_operation(int transactionId, string variable, int timestamp);
    void write_operation(int transactionId, string variable, int value, int timestamp);
    bool end_transaction(int transactionId, int timestamp);
    void fail_site(int siteId, int timestamp);
    void recover_site(int siteId, int timestamp);
    bool check_for_cycle(vector<Transaction*> committedTransactions, Transaction* txn);
    void dump_system_state();
};
```

2 Flow

After parsing each line according to the type of operation, the following steps are carried out.

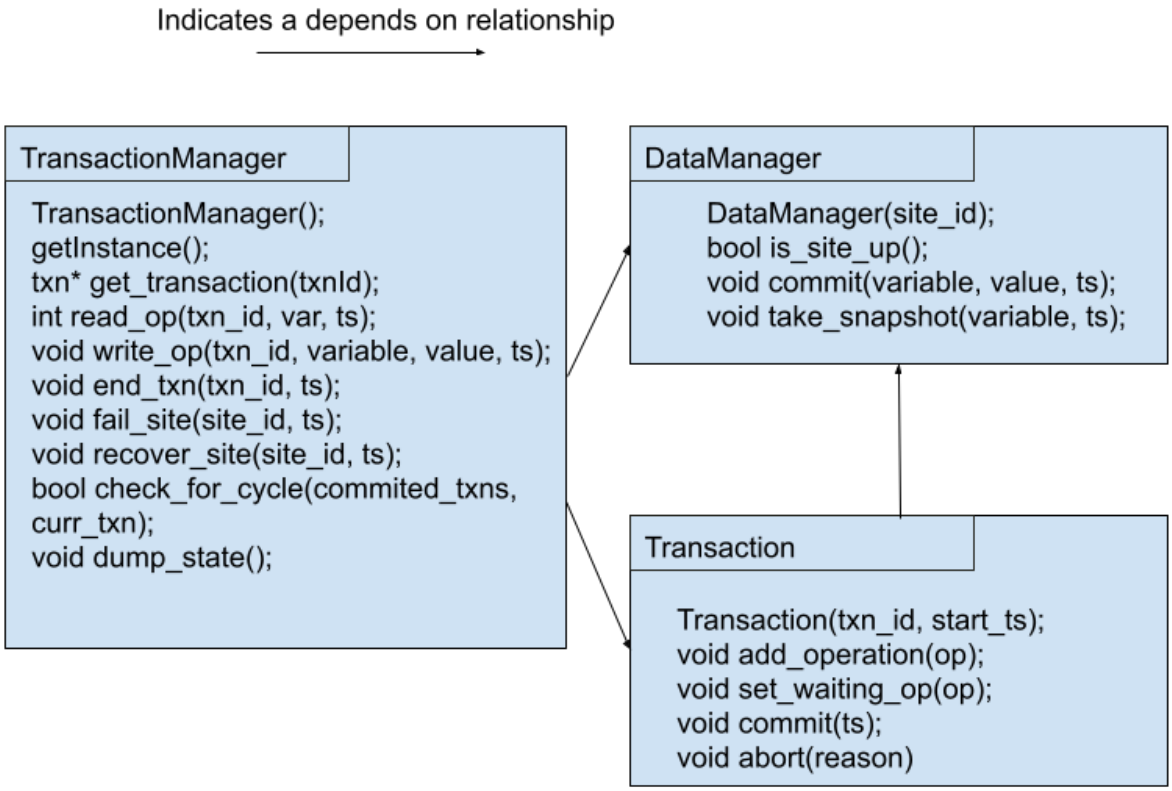

Figure 1: UML Diagram

1. **Begin Operation:**
   - Creates a new transaction object with active status.
2. **Read Operation:**
   (a) **Unreplicated Variable:**
      - Check if the site is up.
      - If it is up, read the value from the time the transaction began.
      - If the site is down, add the read operation to the waiting operation of this transaction.

   (b) **Replicated Variable:**
      - Check if the site is valid (i.e., check if there was any failure of that site between the last commit time and the transaction start time).
      - If it is valid and the site is up, read the value from the time the transaction began.
      - If valid but the site is down, move to the next valid site that is up.
      - If all valid sites are down, add the operation to the queue.

- If no valid sites, abort the transaction and print the reason for abort along with transaction ID.

3. **Write Operation:**

   (a) **Unreplicated Variable:**
   - Check if the site is up.
   - If it is up, write the value.
   - If the site is down, add the write operation to the waiting operation of this transaction.

   (b) **Replicated Variable:**
   - Since the project ensures that not all sites will be down at once, check all active sites available for writes.
   - Store these active sites for writing.

4. **End Operation:**
   - Commit the transaction if all the following checks pass Available copies check, First committer check, rw cycle check

5. **Fail Site Operation:**
   - Marks a site as failed.

6. **Recover Site Operation:**
   - Marks a site as recovered and available again.
   - Resets the access of variables
   - Checks if there are any transactions waiting for this site and start executing them

7. **Dump Operation:**
   - Dumps the current system state.

3 Usage

- Run `make` in the root directory to create the executable `main`.
- To run the program with a test file, execute: `./main <path to test file>`
- To run the program from stdin, execute: `./main` After entering all the lines, press `CTRL + C` to exit.
- To run all the test cases located in the `test` folder, execute: `make test`

3.1 To use with reprozip:

```
reprounzip directory setup rcc-submission.rpz run-dir
reprounzip directory run ./run-dir < test/test1
```

3.2 Sample output:

If we run `./main test/test1` where `test1` contains the following:

- begin(T1)
- begin(T2)
- W(T1,x1,101)
- W(T2,x2,202)
- W(T1,x2,102)
- W(T2,x1,201)
- end(T2)
- end(T1)
- dump()

The output will be:

```
bash-3.2$ ./main test/test1
x2,202 from T2 is written to sites 1 2 3 4 5 6 7 8 9 10
x1,201 from T2 is written to sites 2
----- T2 committed
----- T1 aborted: First committer checks failed
site 1 - x2: 202, x4: 40, x6: 60, x8: 80, x10: 100, x12: 120, x14: 140, x16: 160, x18: 180, x20: 200,
site 2 - x1: 201, x2: 202, x4: 40, x6: 60, x8: 80, x10: 100, x11: 110, x12: 120, x14: 140, x16: 160, x18: 180, x20: 200,
site 3 - x2: 202, x4: 40, x6: 60, x8: 80, x10: 100, x12: 120, x14: 140, x16: 160, x18: 180, x20: 200,
site 4 - x2: 202, x3: 30, x4: 40, x6: 60, x8: 80, x10: 100, x12: 120, x13: 130, x14: 140, x16: 160, x18: 180, x20: 200,
site 5 - x2: 202, x4: 40, x6: 60, x8: 80, x10: 100, x12: 120, x14: 140, x16: 160, x18: 180, x20: 200,
site 6 - x2: 202, x4: 40, x5: 50, x6: 60, x8: 80, x10: 100, x12: 120, x14: 140, x15: 150, x16: 160, x18: 180, x20: 200,
site 7 - x2: 202, x4: 40, x6: 60, x8: 80, x10: 100, x12: 120, x14: 140, x16: 160, x18: 180, x20: 200,
site 8 - x2: 202, x4: 40, x6: 60, x7: 70, x8: 80, x10: 100, x12: 120, x14: 140, x16: 160, x17: 170, x18: 180, x20: 200,
site 9 - x2: 202, x4: 40, x6: 60, x8: 80, x10: 100, x12: 120, x14: 140, x16: 160, x18: 180, x20: 200,
site 10 - x2: 202, x4: 40, x6: 60, x8: 80, x9: 90, x10: 100, x12: 120, x14: 140, x16: 160, x18: 180, x19: 190, x20: 200,
```