# Project RepCRec Design Overview
## (Replicated Concurrency Control and Recovery)


SUMMARY



| | |
|---|---|
| **Author** | Yichang Chen, Weiqiang Li |
| **Status** | Draft \| **Final** |
| **Created** | 11/01/2018 |
| **Last updated** | 12/07/2018 |

# Objective

In this project, a distributed database is implemented, equipped with multiversion concurrency control, deadlock detection, replication, and failure recovery. This project will simulate a simplified distributed database, taking inputs from an input file or standard input as execution operations, and behave like a real database with concurrency and recovery. The outputs of our project will be printed on screen.

# Overview

The project consists of two main modules: transaction manager and data manager. Transaction manager is responsible for reading inputs from the inputs files and execute operations on it. It will also handle any waitlisted operations and coordinate with other modules to maintain the functions of the database system. Data manager is responsible for handling data consistency and actions on site. When a site is down or when the site recovers, the data manager would perform necessary operations to ensure data spawn among the sites are consistent. There are also other modules worth mentioning: lock manager will handle the read and write lock requests for a single variable, and deadlock manager will perform cycle detection and report to transaction manager when deadlock happens.

# Project Structure

## Project Files

The project folder contains several files that are related:

- src/ folder contains all the source code of this project. Since it is written in Java, all source code files are ended with .java.
- RepCRec.rpz file is the packaged file using reprozip that can be run with reprounzip.
- test/ folder contains sample the test files of this project. They can be used as the input of our program.
- design_document.pdf file (this file) serves as the design document.

Our source code is well documented. Every class has an information section showing the main feature of this class, together with author and update date. All methods in

every class have general descriptions, inputs, returns or any side effect. Please refer to the source code for this information.

### How to Run

To run the project, first install reprounzip using pip. Using command line to locate to this folder, and then type:
> reprounzip directory setup RepCRec.rpz ~/repcrec
> reprounzip directory run ~/repcrec
This will start the project and run it on test/input18.txt. Actually, the project can either take a file as input or standard input. To run it on another file, simply type:
> reprounzip directory run ~/repcrec --cmdline java -jar RepCRec.jar <input_file_path>
Or, to run it with standard input, simply type:
> reprounzip directory run ~/repcrec --cmdline java -jar RepCRec.jar

Also, this project can be run under pure Java. The following commands will also run the project:
> cd src
> javac *.java
To run it on an input file, simply type:
> java RepCRec <input_file_path>
Or, to run it with standard input, simply type:
> java RepCRec

## Data Storage and Data Structures

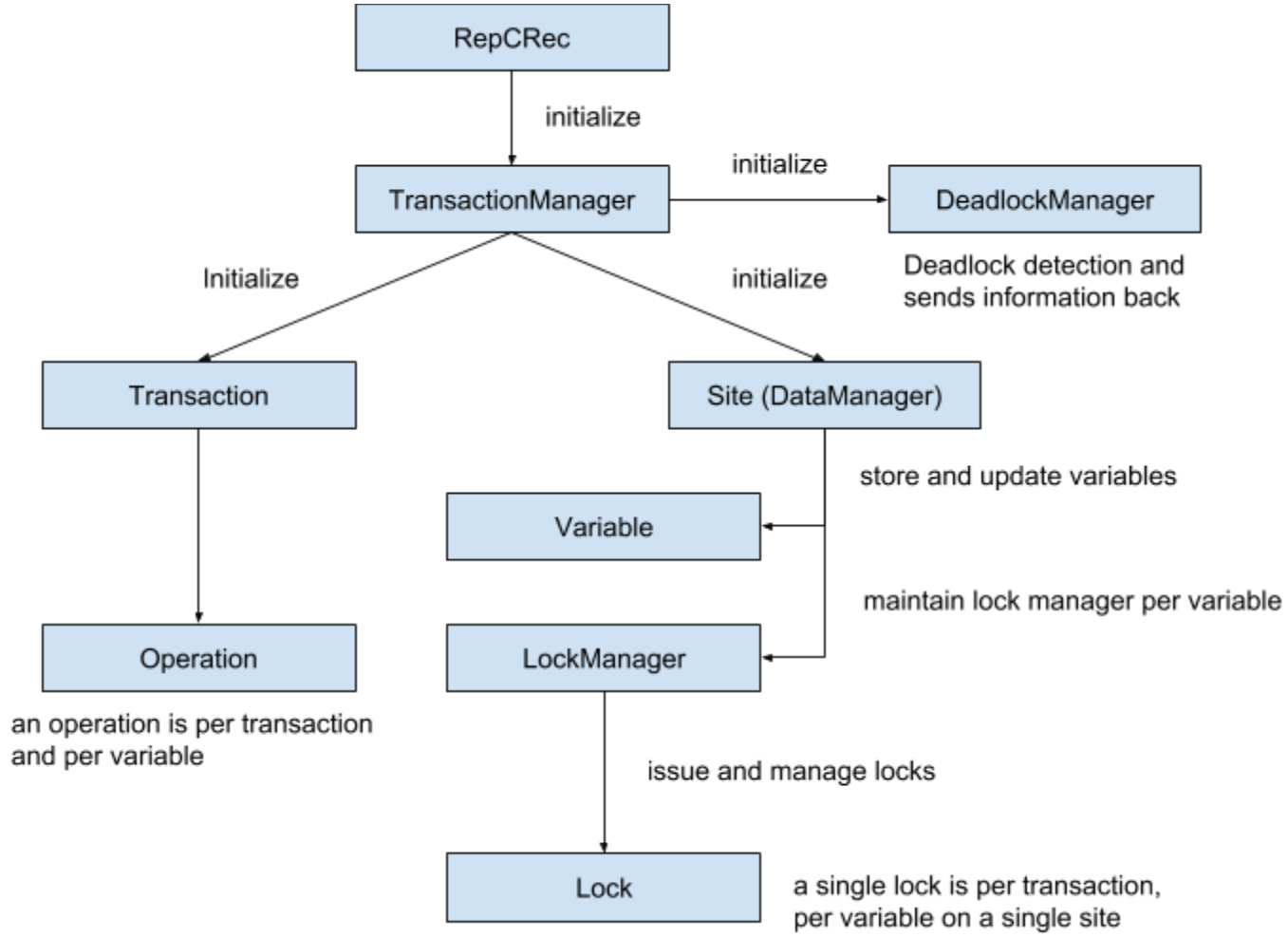
### Site

A site is represented by a class, within which contains data stored in the site, status of the site, waiting transactions that perform read or write on the site, lock manager, and other necessary information. The variables, lock managers and operations are stored in maps.

### Transaction

In our implementation, each transaction would be implemented as a class. In the transaction class, we maintain important informations such as whether the transaction is read-only or read-write, birth time, operations in the transaction and status of the transaction (aborted, ended or ready to commit).

## Modules and Implementations




### Transaction Manager

Transaction manager is the most important class of the whole distributed database. In the scope of this project, it could be a class that manages all transactions. The transaction manager is responsible for the following: read from input file or standard inputs, execute an operation or send to waitlist, maintain the current time, send information to deadlock manager, abort transactions when exception occurs, etc. When a read or a write is executed, we perform the execution and output the result if execution is valid. When a transaction is started, committed or aborted, we update status of the corresponding transaction and check potential conflicts.

### Data Manager (Site)

Data manager (site) would be responsible for managing data storage and locks. Also, site can fail or recover from failure. The data manager maintains status of the current site and each variable together with the corresponding lock manager. Whenever a site fails, the data manager will perform necessary operations on the variables and transactions; whenever a site recovers, the data manager marks this site as recovered and updates the status of the variables. When new operation is sent to this site, the data manager will determine that it can be executed immediately or put in the waitlist. When a transaction will commit on this site, the data manager will also make the actual modifications to help commit the transaction.

### Lock Manager

Lock manager is the lock holder for a single variable on a site. It will maintain a list of locks that currently holds on this variable. Built on top of the lock class, it manages all the tests and requests for locks and responds to every request.

### Deadlock Manager

Deadlock manager detects cycles by building a graph using the current transactions to avoid deadlocks. It maintains a graph containing all transactions that are currently executing, and runs BFS topological sort to detect if there is any cycle. If a cycle is detected, it sends the information to our transaction manager where the deadlock will be resolved.

## Sample Tests and Explanations

The followings are some sample tests as input and the expected output. All the input test files are located in the test/ folder. The following input files are from the sample test on our course website: input1.txt, input2.txt, …, input21.txt. Other tests are input22.txt, …, input27.txt. We pick several inputs from the test/ folder and provide some explanations.

| Input | Explanation |
|---|---|
| input1.txt | Deadlock situation. T2 will abort because it is younger. Finally, x1=101 and x2=102 on all sites. |

| | |
|---|---|
| input3.5.txt | T2 aborts because site 2 fails. Due to fail, T1 cannot write to x4 on site 2. Finally, x4=91 on all sites except site 2. |
| input7.txt | T2 will read the initial value of x3, hence 30. Finally, x3=33. |
| input10.txt | T3 waits for T2 to end and reads 44. Finally, x2=22 on all sites and x4=44 on all sites. |
| input18.txt | T5 aborts because of deadlock. All of the rest can commit. Finally, x2=10 on all sites, x3=20, x4=30 on all sites, x5=40. |
| input21.txt | T2 aborts due to deadlock. Because T1 is waiting for a write lock, T2 cannot directly promote its read lock to write lock, hence deadlock. Finally, x2=202 on all sites. |
| input25.txt | T4 aborts due to deadlock, then T3, then T2. Finally only T1 commits, and x2=222 on all sites. |