# CS425 MP3: Design Document

Bhopesh Bassi, Zhongshen Zeng

May 1, 2017

## Contents

# 1 Overview

In this document, we describe the design of a distributed transaction system that support read and write to distributed objects while ensuring full Atomicity, Consistency and Isolation properties. We start by discussing various aspects of the design and our approach followed by implementation details.

# 2 Consistency, Concurrency and Isolation

## 2.1 Isolation

We use locks for isolating transactions from each other. As we are assuming no failure model, we use a centralized lock service as it is easier to implement and reason about. It also makes the deadlock detection easier as discussed in section 3.

## 2.2 Concurrency

We use **one lock per key**, so that transactions working on different keys don't interfere with each other. Moreover, **we used read-write locks where read locks could be shared while write locks are exclusive**. The read lock is upgradable to write lock but this could result in deadlock when multiple readers try to upgrade.

**In a real system, we wouldn't use one lock per key,** rather one lock per hash and keys with same hash will share the lock. This allows us to trade off a bit of concurrency for memory. Infact, this is what we implemented initially but then Professor said that they won't test with too many keys, so, it's okay to have one lock per key for smooth testing.

## 2.3 Sequential Consistency

We adopt **two phase locking** algorithm discussed in class to ensure concurrency and isolation. We did not use strict two phase locking algorithm since it uses only exclusive locks and does not satisfy the read sharing requirement of this MP.

We are not using optimistic concurrency control method because it is more suitable for systems with low data contention. For our MP requirements, if testing with high data contention, the constant roll back of transactions introduced might be significantly larger than the overhead of 2-phase locking.

# 3 Deadlock detection and resolution.

We do deadlock detection at the lock service. As lock service is aware of which transactions hold which locks and which transactions want which locks, it easier to do it at the lock service. We create a wait for graph between transactions and if a cycle is detected, we kill one transaction randomly. This ensures minimum number of transactions killed because if deadlock is not resolved even after killing one transaction, there will be a cycle in the resulting graph and we will kill one more transaction. In this way, killing one transaction at a time ensures minimum number of killed transactions. We could have used some more sophisticated techniques to detect how many transactions need to be killed in one shot but didn't have much time to explore this further.

There are multiple attributes can be used to evaluate priorities among different deadlocked transactions, such as timestamps, number of operations and number of locks waiting. However, due to simplicity consideration and the practical time we have, we decided to randomly decide a transaction to abort.

## 3.1 Other schemes considered

Another deadlock detection solution considered is the edge chasing method introduced in class. The major reason we did not implement the decentralized method is to lower the false positive detection rate(locks might be released during the transportation of messages). Also, since we have been guaranteed by the MP problem statment that clusters will be stable throughout the applications, we feel like centralized detection will be easier and more accurate than decentralized method.

# 4 Implementation

- Just like previous MPs, we have tried to keep code modular and build up from small pieces. You will see lots of interfaces/classes in the code but each of them are quite small.

- We reused code to store and read key-value pairs at different machines from previous MP. Because, we wrote code for previous MP in a modular way, **we didn't have to write a single line of new code for storing and reading keys.** We just had to implement locking service and client interface.
- We use Java's read-write locks which give us almost everything we want. However, Java's read locks are not upgradable to write locks for the sole reason that it is deadlock prone. We use one more lock(per key) to work around this and atomically upgrade to write lock.
- We use **interruptible locks** and when deadlock detector detects a deadlock, it interrupts one of the transaction threads. Transaction thread upon receiving interrupt releases all it's locks and kills itself.

# 5 Acknowledgement

We would like to thank the entire staff for staying on top of our Piazza (public and private) posts and always providing useful feedback and hints.