CS 425 MP3 Report

I. Submission Information

Group Members: Jianfeng Xia(jxia11), Cheng Hu(chenghu3)
VM Cluster Number: 10
GitLab Repository: https://gitlab.engr.illinois.edu/chenghu3/cs425_mp3

Instructions:

- Configurations:
    1. Servers:
       A running at VM01, port 9000
       B running at VM01, port 9001
       C running at VM01, port 9002
       D running at VM01, port 9003
       E running at VM01, port 9004
    2. Coordinator:
       Coordinator running at VM02, port 9000
- Install graph library: `go get github.com/twmb/algoimpl/go/graph`
- To build: `go build mp3.go`
- To run:
    1. Run server: `./mp3 server name port`
    2. Run client: `./mp3 client name`
    3. Run coordinator: `./mp3 coordinator port`
- Revision to be graded:
  54d5f293dceeb5fd2e1d3039bffe60b10a6ed570

II. Algorithms & System Design

Overview

To ensure full ACI(D) properties, we implemented two-phase locking with read sharing and atomic lock promotion. We also implemented centralized deadlock detection with a coordinator. For this MP, we used Go's `net/rpc` to implement all functionalities.

Atomicity: Tentative Update

We used tentative updates to ensure atomicity. Specifically, the client `SET` command only requests a writer lock from the server. After acquiring the writer lock, the client will create a local shadow copy of the object and read or write to the local copy. The "real" remote objects on the server will only be overwritten at commit time, and the updates will only be visible to other

transactions after commit. If a transaction is aborted, the tentative updates will be discarded and no changes will be applied.

Concurrency & Isolation: Two-phase Locking

We implemented two-phase locking to ensure serializability of concurrent transaction executions. In our implementation, each remote object has a distributed reader-writer lock. Locks on different objects do not interfere with each other (unless a deadlock is detected). The lock is implemented by setting readers/writer flags (protected by a mutex) and maintaining a queue of lock requests. Lock requests are added to the queue if they cannot be granted immediately. Lock promotions are added to the front of the queue. Other requests are added to the back of the queue. Our reader-writer lock is **write-preferring**, so writer lock requests in the queue will block later read requests. On releasing a lock for an object, the server will grant as many locks in the queue as possible for maximum concurrency (1 writer lock/all consecutive reader locks).

Deadlock Detection

We implemented centralized deadlock detection with a coordinator. The coordinator maintains a directed wait graph of transactions. If a lock request on an object is to be added to the queue, the server will first report to the coordinator, adding edges from the requesting transaction to all transactions it is waiting for. Specifically:

- Requests must wait for all reading or writing transactions.
- Write requests must wait for all previous transactions in the lock request queue.
- Read requests must wait for all previous transactions in the lock request queue, **except** any trailing read requests which can be granted concurrently.

The coordinator then detects cycles by computing the strongly connected components in the graph. If a cycle is detected, the **requesting transaction** has to abort.