HW5 – Locking and Transaction Management

Excepted Effort: 20 hours

Objectives

- Become familiar with locking
- Become familiar with transaction management concepts

Overview

In this assignment, you built a simple transaction management system with locking capabilities. Like HW4, this system will support multiple concurrent worker processes. Unlike HW4, however, it will allow for multiple concurrent resource managers. Each resource manager will keep track of a simple resource (a string value) and will support the following operations on that resource: lock, read value, write value, unlock. The system will also include one transaction manager that supports the following operations: start transaction, commit transaction, and abort transaction.

System Description

The HW5 system will contain name service, three process components, and at least two helper classes. See Figure 1. As in HW4, all of the process components have an identifying name and one or more communication end points. If you do all of your communications via one reliable communication sub-system, then one end point should be sufficient. Below are descriptions of three process components and two helper classes. However, you are few to evolve and extend the design as you see fit, as long as the system a) support multiple current resource managers, b) supports multiple concurrent workers that use resources managers by those resource managers, c) can handle multiple conflicting transactions, and d) guarantees that all concurrent transaction executions are serial equivalent using a pessimistic concurrent control mechanism, e.g. locking.

Resource

A resource is an object that the workers use (via the resource managers). For simplicity, our resources will be simple string values identified by an integer ID. See Figure 1. To control concurrent access to the resources, the system should only allow access to resource through their resource managers, like in HW4.

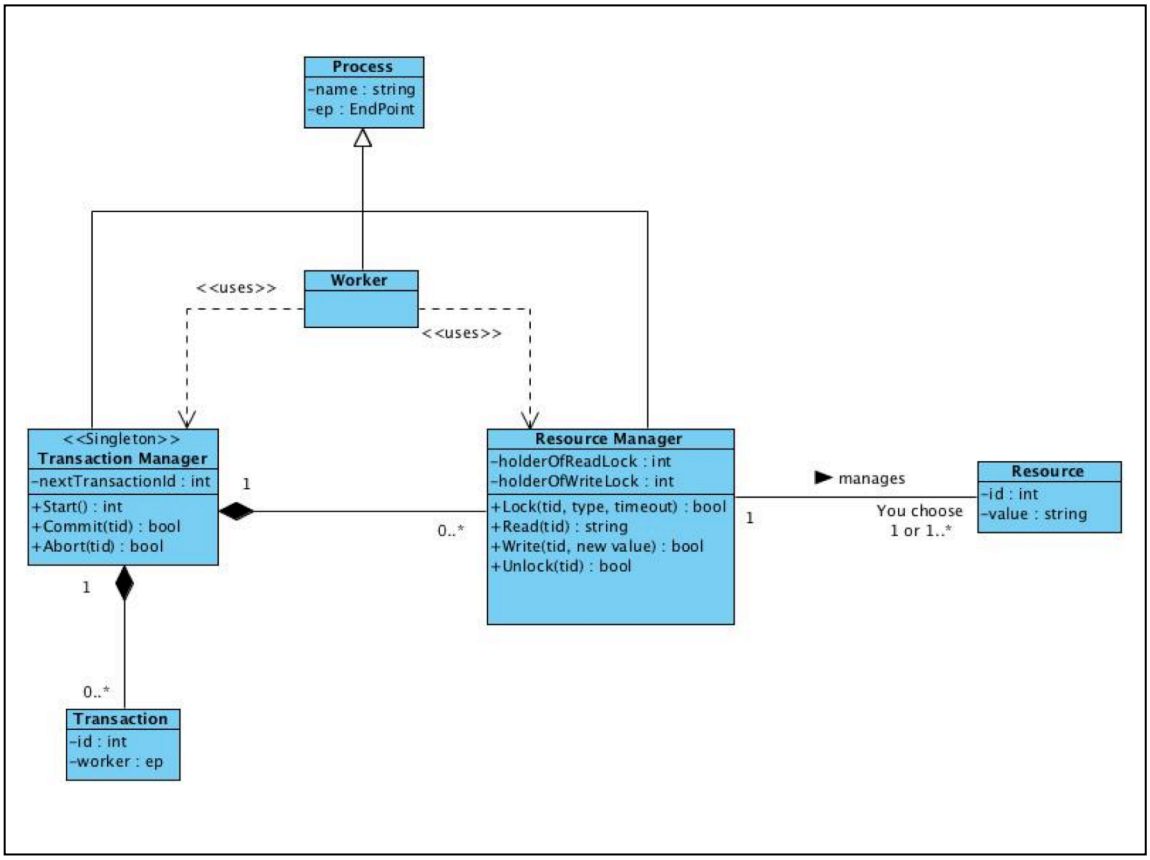

Figure 1: Conceptual model of key components

Resource Manager

A resource manager is a front-end to one (or optionally more than one) resources with the following operations, as a minimum:

Lock(tid, type, timeout) : bool

This method will attempt obtain a specific type of lock for a resource for transaction *tid*. The type can be "READ" or "WRITE". The resource manager should grant a READ lock only if no other transaction has a WRITE lock and a WRITE lock only if no other transaction has a READ or WRITE lock.

Read(tid) : string

This method returns the value of a resource if the specified transaction holds a read or write lock on the resource, otherwise it return null.

Write(tid, newValue) : bool

This method writes the newValue to the resource (at least tentative pending the committing of the transaction) and returns true only if the specified transaction holds a write lock on the resource; otherwise it returns false.

Unlock(tid) : bool

If the specified transaction holds a lock, this method releases the lock and returns true; otherwise it returns false.

Note that if you are going to allow a resource manager to handle multiple resources, then you must adapt above operations to include a parameter that identifies the desired resource.

Also, you will need to figure out the following:

- How to rollback changes (via an undo or restore type of operation) to a resource if a transaction is aborted
- Whether you need to keep track of tentative versions

Transaction Manager

The transaction manager's role manage transactions in progress and allow workers to start, commit, or abort them. As a minimum, it needs to support the following methods:

Start() : int

This method starts a new transaction and returns the id. The transaction needs to keep track of the worker who started it, so depending on how you are managing your end points, you may need to add a parameter that identifies the worker.

Commit(tid) : bool

This method commits a transaction, making permanent changes to the resources that it updated.

Abort(tid) : bool

This method aborts a transaction, causing of the resources that it tentatively updated to return to whatever their state was at the beginning of the transaction.

Transaction

This is a passive helper class for tracking transaction state. As a minimum, it needs to keep track of the transaction id and the worker which started it.

Worker

This is the worker process that tries to execute transactions on the resources. To keep things simple, a single worker should have at most one open transaction at a time.

Workers should be able to read transactions from script file and execute them. These script files may include START, READ, WRITE, COMMIT, and ABORT expressions, along with other kinds of data manipulation operations like string concatenation. It is your job to design your own transaction scripting language. Keep it simple.

For discussion purposes, consider an example language where resources are identified by an "R" followed by a three digital number. Read commands have a variable on the left hand side into which the read results are saved. Write commands have a variable on the right hand side from which the new value is take. Also, to test special conditions and simulate failures, this language allows transactions to include SLEEP statements. Here are few simple sample transactions in this language.

Sample 1

```
tid = START
R001.LOCK(tid, READ, 1000);
X = R001.READ(tid)
R002.LOCK(tid, READ, 1000);
Y = R002.READ(tid)
Z = X + Y;
R003.LOCK(tid, write, 1000);
R003.WRITE(tid, Z);
R003.UNLOCK(tid);
R002.UNLOCK(tid);
R001.UNLOCK(tid);
COMMIT(tid);
```

Sample 2

```
R004.LOCK(tid, READ, 1000);
X = R004.READ(tid)
R003.LOCK(tid, WRITE, 1000);
R003.WRITE(tid, X);
SLEEP(5000);
R003.UNLOCK(tid);
R004.UNLOCK(tid);
COMMIT(tid);
```

In your scripting language, you may want to consider making locking and unlocking implicit.

Instructions

Step 1 – Make Key Design Decisions

First, you need to make some key design decisions, like

- Whether to have a resource manager handle multiple resources
- The features and syntax of your transaction scripting language
- Whether to do implicit locking
- Whether transaction have to conform to two-phase or strict-two-phase locking
- How aborts will be causes undo's or restores on modified resources

Step 2 – Implement and test your resource and resource manager components

With key design decisions made, design, implement and test your resource and resource manager components.

Step 3 – Stub out your transaction and transaction manager

Do enough of the transaction and transaction manager components that they public interfaces are available. This will help you write and test your work in the next step.

Step 4 – Implement and test the worker

Implement the worker so it can read all of the scripts in a directory (one at a time and execute time). Test it without relying a fully functional transaction manager.

Step 5 – Complete basic implementation

Finish the basic implementation of the transaction and transaction manager component, and integrate all of the components. Then, do integration and system testing.

Step 6 – Handle special situations and certain failures

Implement some degree of fault tolerance. At a minimum, you need to supports workers going down while in the middle of a transaction. Also, consider taking measures to prevent deadlock.

Step 7 – Final system testing

Come up with a meaningful system configuration and set transaction scripts that perform a thorough system test.

Grading Criteria

	Points
	=====
Good design choices	30
Good implementation and testing of resource manager	30

Good implementation and testing of transaction manager	30
Good implementation and testing of worker	30
Meaningful system-test configuration and transaction scripts	30
Total	150