PG - 1
COMP-512: Distributed Systems, Fall 2017

Project Phase 2: Locks & Trasaction Management

Shabir Abdul Samadh
McGill University
Student ID: 260723366
shabir.abdulsamadh@mail.mcgill.ca

Jianhua Li
McGill University
Student ID: 260351181
jianhua.li2@mail.mcgill.ca

ABSTRACT

This report provides an overview of the architecture and the design choices of distributing a standalone application across multiple instances under two different distributed paradigms: **TCP Sockets** & **RMI Calls**. The application used here is a simple reservation system that allows the user to add, remove, update & query information related to its first class entities. These entities are `Cars, Flights, Hotels (Rooms)` and `Customers`. The standalone application provides a simple **interface** along with a **client** program and a corresponding **server** implementation and it uses an **RMI** registry to enable the client to invoke methods on the server. We have distributed this standalone system in such a way that each first-class entity of the system `(i.e.Car, Flight, Hotel & Customer)` are managed by separate instances of the server. We have bridged these separate instances via a **Middleware Server**. Given that the `Customer` is common and is related to all three other entities, we have managed customer related processing in our middleware server. In addition we have experimented this distributed setup using two technologies as mentioned above: **TCP** & **RMI**. The system is completely written in `Jave`.

Keywords

Distributed-Systems; Middleware; TCP Sockets; RMI; Nonbocking; Client-Server

1. INTRODUCTION

In this report we will describe the way we implemented 2-Phase Lock(2PL) mechanism inside the travel reservation system and present its performance characteristics. We decided to use RMI instead of TCP due to ease of use and being less prone to errors. And we also chose to put customers just inside middleware instead of all three resource managers like in phase one.

2. OVERALL FLOW OF A TRANSACTION

Unlike before, if a client wants to read or write an item, the client has to start a new transaction. And after finishing the desired operations, the client has to commit the transaction in order to save all the changes.

2.1 Transaction Class

Each time a new transaction is called, we create a new transaction object. Inside the *transaction* class, there is a map *accessedItemSet* that contains all the data items that have already been accessed.

3. TRANSACTION MANAGER

3.1 Reserve Operation

When we call *submitReserveOperation(tId, customerItem, requestType, resourceItem)*, first we retrieve the *Transaction* object. Then we have to acquire the necessary lock for the operation. After getting the lock, first we check whether the customer has already been accessed within the same transaction. If so, we just directly use the customer object that's inside the *Transaction* object. If not, then we have to start a mini transaction where we try to get the customer object from customer manager which is inside the middleware, once we get the *customer*, we commit this mini transaction. Then we need to actually reserve the item for the customer, for which we have to call *submitOperation(tId, requestType, resourceItem)* first. Inside this method, it will also first check whether the item has already been accessed by looking inside *transaction.getAccessedItemSet()*, only has it not been accessed will it try to retrieve the reservable item from the resource managers. Eventually it returns the desired value so that we can proceed with the reservation.

3.2 Modify Customers

If we want to modify customers, the principle is similar to a reserve operation. We just have to call *submitOperation(tId, requestType, customerItem)*. We first check whether the cusomter has already been accessed. If so, we just access the customer object that is already inside the transaction. If not we make a mini transaction in order to get the customer object.

3.3 Commits

When commiting, for each of the resource type, we start up a transaction with the appropriate resource manager. Then for each modified item, we use the appropriate action (write or delete). After we finish all the actions we just commit the transaction. And after we are done with all the changes, we unlock all the locks held by the transaction.