Distributed Transaction Application in Java or C#

Project Description

The purpose of this project is to gain an understanding of the interaction between various components of a TP system, and the implementation issues involved. Your goal is to construct a distributed application in Java or C# that implements a travel reservation system. Students will work in pairs.

The project is organized as a sequence of steps that iteratively add components and system structure, working toward the ultimate goal of a multiple client, multiple server, scalable system. The steps are not all of comparable difficulty, and therefore should not be used as weekly milestones. Effort required is also not proportional to the length of the specification, so long specs might be easier to implement than short ones. Many of the individual blocks and methods can be done in parallel.

We provide a lock manager, parts of the basic application, and a basic test script, to ensure minimal functionality. It is your responsibility to augment the tests (using the standard interfaces) to make certain that your service resists failure.

Your options for software development environment are the following:

- Oracle's JDK. This is relatively labor-intensive, but RMI is supported natively. With Oracle's JDK you can use Windows or Unix machines. Recommended reading for novice users of Java: http://docs.oracle.com/javase/tutorial/rmi/index.html

- The Microsoft .NET Framework SDK or Visual Studio .Net with the C# language, using SOAP for remote procedure calls. The implementation effort is lower than for Java, since support for distribution is more built-in.

We expect you to hand in two milestone reports of your progress, which will be reviewed but not graded.

Provided Components

1. We have provided a lock manager package/class. It supports the following operations:

- lockForRead(Xid, thingBeingLocked) throws DeadlockException

- lockForWrite(Xid, thingBeingLocked) throws DeadlockException

- unlockAll(Xid). Xid is an abbreviation for transaction identifier.

The lock manager handles deadlocks by a timeout mechanism. A failed (deadlocked) lock operation throws an exception. Lock managers are described in the textbook in Chapter 6, Section 2.

2. Reservation Application

There are four types of resources: flights, rooms, cars, and customers. For each type of resource, there are operations to add or remove units (e.g., addCars), reserve units (e.g., reserveCar), and query the state of units (e.g., queryCar, queryCarPrice). The operations to be supported are outlined in the attached Java

interface. We know these assumptions sacrifice verisimilitude, but they are rich enough to expose the problems to be addressed by this project.

Please read the code to understand the semantics of the basic operations. Here's a quick summary: The data is stored in memory. There is only one airline (so a flight identifier is an integer), only one type of car, only one type of hotel room, and only one day for which units can be reserved. Since there is only one type of flight, car and hotel room, each of them has only one price. The net effect of { addCars(T1, 'San Diego', 4, $52); addCars(T2, 'San Diego', 7, $54); } adds 11 cars at $54, not 7 cars at $54 and 4 cars at $52. One can query for which reservations the customer holds and for how much the customer should be charged. Note that there's no account payment feature.

Project Steps

1. *Atomicity* - Build a simple Resource Manager (RM) that supports atomicity. The simple RM implements transactions. That is, it supports the methods start, commit, and abort, and all data access (read/write) operations that are associated with a transaction. In the first few steps of the project, you can use one instance of the storage class for all four RM types. After that, you will need one storage instance for each RM type.

The RM stores the database in hash tables. First, write a new transactional storage class that is generic enough to store resources on behalf of any or all of the four resource types. The storage class implements read, write, abort, and commit. Abort undoes all of a transaction's updates. Commit simply installs the transaction's updates so that other transactions can read them. Commit does not need to store the transaction's updated data on disk (that comes in Step 2 on Durability).

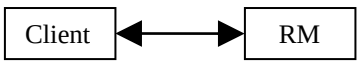
To implement atomicity, we recommend that you use shadowing: make a copy of the in-memory database; update it; and then to commit, update the database pointer for the active memory image so that it points to the updated copy. In step 2, this will be a disk image that will be copied, updated, and relinked (renamed). Shadowing is described in the lecture notes and in the textbook in Chapter 7, Section 6. For this step, it is enough to shadow records; page shadowing is considered extra credit.

Now rewrite the RM so that it uses the new storage class, in fact, so that all RM instances use *the same* storage instance. By making the RMs use the same storage instance, you avoid the need for two-phase commit. If you don't understand that comment, re-read the description of two-phase commit in Chapter 1.

For verisimilitude, you might want to modify the make file so that it creates a .dll for the RM, instead of an .exe. (The ultimate goal, however, is to have the RMs run as separate processes, starting in Step 7.) That way, when you create RM instances, the instances will live in the caller's process. Thus, all three RMs will run in the same process, so you can think of it as a single server. This step is optional, since it just makes the model more realistic without changing the functionality of any component. It's also optional whether the storage instance runs in this process or in a separate process.

In this step, the only failure to handle is an abort. Since the memory image is lost when the process terminates, this step doesn't need a recover() method.

The Technical Interface methods (located in ResourceManager.java and TP.cs respectively) are defined to make it easier to test for faults. The shutDown() method implies that the RM should shut down gracefully. In this case, that means cleaning up its files, so that the next time it starts up, it does not attempt to recover its state. The selfDestruct() method exists to allow failure generation between two disk writes. The idea is that it sets a counter of disk writes that will be executed successfully before the RM terminates. The RM will have to startup and recover from termination.

The system now looks like this: 

2. Durability. Add persistence and recovery to the Resource Manager. All state is stored on disk. The disk image is updated when a transaction commits. The RM must implement a recover() method to restore its state from the state on disk and gracefully handle various exceptions, such as methods called with unknown (forgotten) transaction ids.

The system now looks like this: 

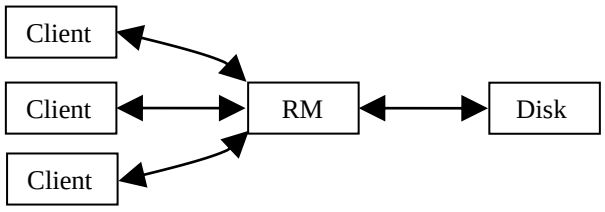
3. Lock Conversion - Modify the lock manager that so that it can convert locks, e.g.,

```
Lock(xid, foo, read);
/* read foo */
Lock(xid, foo, write);
/* write foo ... plus error checking and exception handling...*/
```

Keep in mind that other transactions may have read locks on foo, so deadlock is possible. The main purpose of this step is to gain an understanding of the lock manager code.

The lock manager uses a hashtable for each lockmode, because the author thought it would lead to succinct code with few special cases. Feel free to rewrite it, if you don't like that choice of data structure.

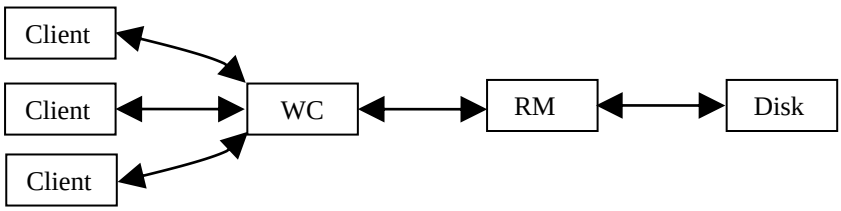
4. Isolation. Add lock and unlock operations to the RM. That is, the RM should lock data appropriately for each transaction, and unlock data when the transaction commits or aborts. Test this implementation using multiple clients and a single resource manager. You might experiment with different locking granularities at this stage. There are several clients interacting with the single RM, so the system now looks like this:



5. *Implement a workflow controller (WC).* This is called a Request Controller in the textbook. It is described in Chapter 1, Section 2, and in Chapter 3. The WC is a front-end so that (eventually) the location and partitioning of the RM's are not exposed to the client. To do this, the WC supports the same interface as the RM, plus the new method:

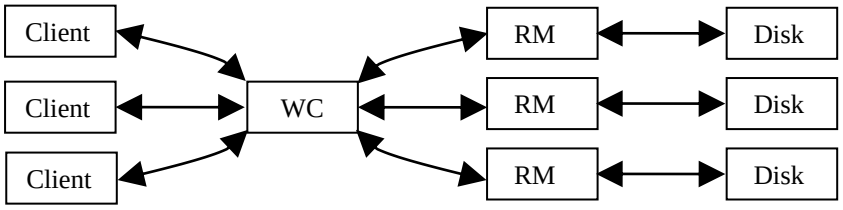
reserveItinerary(customer, flight_list, location, bCar, bRoom) method.

A customer has only one itinerary. The parameters of the reserveItinerary method are: customer, the customer id from newCustomer; flight_list, a vector of integer flight numbers; location, the place where rooms or cars are reserved; and bCar/bRoom, true if the customer wants a car/room reservation. Within this step, assume there is only one RM, and all methods except reserveItinerary are directly passed on to it. So the system looks like this:



For example, suppose the client asks the WC for a reservation on flights 435 and 534 and a rental car in St. Louis. To process this request, the WC starts a transaction, calls the RM to make a reservation for flight 435, calls the RM to make a reservation for flight 534, and then calls the RM to reserve a car in St. Louis.

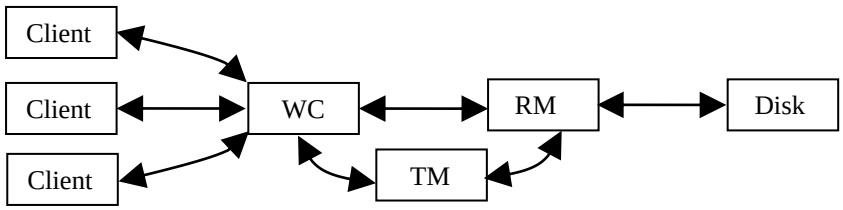
The goal in later steps is to support multiple RMs. For example, flights, cars and hotel rooms could be handled by separate RMs, each with their own private disk storage. There are more pieces to build before the WC can handle this functionality. When you get to that stage the system will look like this:



Since the ultimate goal is to handle multiple RMs, the workflow controller should be given the list of active RMs as command line arguments on startup.

6. *Implement a Transaction Manager.* The TM supports the following operations: start, commit, abort, and enlist. It coordinates each distributed transaction, i.e., each transaction that accesses multiple RMs. The TM is used as follows: whenever a request is made to an RM, the RM calls the TM's enlist method to tell the TM that it is involved in a transaction. The TM keeps track of which RMs are involved in which transactions. The WC forwards a start/commit/abort call by the client directly to the TM. All other calls by the client to the WC are forwarded to the appropriate RM.

At this stage, the TM needs no persistence. Since the TM exists behind the WC interface, no client interfaces are provided for the TM. The workflow controller needs to be given the hostname (probably localhost, if you are running on the same machine) of the TM, in addition to the list of active RMs. The RMs will be given the hostname of the TM on startup. The system now looks like this:



7. *Run multiple RMs.* For example, flight, car and room reservations could be handled by separate RMs. For each active transaction the TM maintains a list of which RMs are involved and implements one-phase commit. The WC decides which data requests and transactions go where. On a commit or abort request (which is forwarded to the TM), the TM calls the appropriate methods on all RMs involved in the transaction.

8. *Add a list of committed transactions*. Modify the TM to store a list of which transactions committed, which is needed for two-phase commit below.

9. *Implement two-phase commit.* At this stage, you should ignore failure handling. That is, implement commit and abort under the assumption that messages are never lost or excessively delayed.

10. *Two-phase commit with failure handling.* Now worry about what happens on failure. In particular, handle cases where messages get lost and ensure the RMs can recover from being in the undecided state (in those cases where it's technically feasible).

11. You have finished the project. But feel free to add some enhancements, such as the following:

i. Logging instead of shadowing. Shadowing is simple but has poor performance. Logging allows better performance, but is very tricky and is a lot more code than shadowing.

ii. Garbage collect the TM's committed transaction list. The Transaction Manager keeps a list of committed transactions, so that a Resource Manager can connect to it after recovery and ask if a particular transaction was committed. Since storage is not infinite, implement a garbage control scheme for this list of committed transactions.

iii. Presumed abort for two-phase commit.

iv. Explicit deadlock detection using a waits-for graph.

v. Partition flights by flight number and do parameter-based routing in the workflow controller.

vi. Implement paged files.

vii. Implement weaker degrees of isolation, e.g,, by changing how long locks are held.

JavaTransaction

Interface ClientInterface
All Superinterfaces:
java.rmi.Remote

---

public interface **ClientInterface**
extends java.rmi.Remote

An interface for interaction with the clients.
This interface is already implemented by class MyWC in package WC.
Class MyWC will eventually be transformed into a real workflow controller. Currently, class MyWC only redirects calls to MyRM: the implementation of the resource manager. If you want clients to interact directly with your resource manager, make sure that your Resource Manager class implements this interface

---

Method Summary

void	**abort**(Transaction context) Abort a transaction
boolean	**addCars**(Transaction context, String location, int numCars, int price) Add cars to a location.
boolean	**addRooms**(Transaction context, String location, int numRooms, int price) Add rooms to a location.
boolean	**addSeats**(Transaction context, String flight, int flightSeats, int flightPrice) Add seats to a flight This method will be used to create a new flight but if the flight already exists, seats will be added and the price overwritten
boolean	**cancelItinerary**(Customer customer) Cancel an itinerary owned by customer
void	**commit**(Transaction context) Commit a transaction
boolean	**deleteCars**(Transaction context, String location, int numCars) Delete cars.
boolean	**deleteFlight**(Transaction context, String flight) Delete the entire flight.
boolean	**deleteRooms**(Transaction context, String location, int numRooms) Delete rooms.
boolean	**deleteSeats**(Transaction context, String flight, int numSeats) delete seats from a flight
String[]	**listCars**(Transaction context) list existing cars
Customer[]	**listCustomers**(Transaction context) list existing customers that have itinerary

String[]	**listFlights**(Transaction context) list existing flights
String[]	**listRooms**(Transaction context) list existing rooms
int	**queryCar**(Transaction context, String location) Get the number of cars available.
int	**queryCarPrice**(Transaction context, String location) Get the cars price.
int	**queryFlight**(Transaction context, String flight) Get the number of seats available.
int	**queryFlightPrice**(Transaction context, String flight) Get the flight price.
String	**queryItinerary**(Transaction context, Customer customer) Get the bill for the customer
int	**queryItineraryPrice**(Transaction context, Customer customer) Get the total amount of money the customer owes
int	**queryRoom**(Transaction context, String location) Get the number of rooms available.
int	**queryRoomPrice**(Transaction context, String location) Get the room price.
boolean	**reserveItinerary**(Customer customer, String[] flights, String location, boolean car, boolean room) Reserve an itinerary
Transaction	**start**() Start a transaction return a unique transaction ID

Method Detail

reserveItinerary

```
boolean reserveItinerary(Customer customer,
                         String[] flights,
                         String location,
                         boolean car,
                         boolean room)
                         throws RemoteException
```

Reserve an itinerary

Parameters:

customer - the customer

flights - an integer array of flight numbers

location - travel location

car - true if car reservation is needed

room - true if a room reservation is needed

Returns:

true on success, false otherwise.

Throws:

RemoteException

See Also:

RM.reserve(Transaction, Customer, RID)

---

cancelItinerary

```
boolean cancelItinerary(Customer customer)
                        throws RemoteException
```

Cancel an itinerary owned by customer

Parameters:

customer - the customer

Returns:

true on success, false otherwise.

Throws:

RemoteException

See Also:

RM.unreserve(Transaction, Customer)

---

queryItinerary

```
String queryItinerary(Transaction context,
                      Customer customer)
                      throws RemoteException,
                             TransactionAbortedException,
                             InvalidTransactionException
```

Get the bill for the customer

Parameters:

context - the transaction ID

customer - the customer ID

Returns:

a string representation of reservations

Throws:

RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

RM.queryReserved(Transaction, Customer)

---

queryItineraryPrice

```
int queryItineraryPrice(Transaction context,
                        Customer customer)
```

```
                    throws RemoteException,
                           TransactionAbortedException,
                           InvalidTransactionException
```

Get the total amount of money the customer owes

Parameters:

context - the transaction ID

customer - the customer ID

Returns:
total price of reservations
Throws:
RemoteException
TransactionAbortedException
InvalidTransactionException
See Also:

RM.queryReserved(Transaction, Customer)

---

addSeats

```
boolean addSeats(Transaction context,
                 String flight,
                 int flightSeats,
                 int flightPrice)
                 throws RemoteException,
                        TransactionAbortedException,
                        InvalidTransactionException
```

Add seats to a flight This method will be used to create a new flight but if the flight already exists, seats will be added and the price overwritten

Parameters:

context - the transaction ID

flight - a flight number

flightSeats - the number of flight Seats

flightPrice - price per seat

Returns:

true on success, false otherwise.

Throws:
RemoteException
TransactionAbortedException
InvalidTransactionException
See Also:

RM.add(Transaction, RID, int, int)

---

deleteSeats

```
boolean deleteSeats(Transaction context,
                    String flight,
                    int numSeats)
                    throws RemoteException,
                           TransactionAbortedException,
                           InvalidTransactionException
```

delete seats from a flight

Parameters:

context - the transaction ID

flight - a flight number

numSeats - the number of flight Seats

Returns:

true on success, false otherwise.

Throws:

RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

RM.delete(Transaction,RID,int)

---

deleteFlight

```
boolean deleteFlight(Transaction context,
                     String flight)
                     throws RemoteException,
                            TransactionAbortedException,
                            InvalidTransactionException
```

Delete the entire flight. deleteFlight implies whole deletion of the flight, all seats, all reservations.

Parameters:

context - the transaction ID

flight - the flight number

Returns:

true on success, false otherwise.

Throws:

RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

RM.delete(Transaction,RID)

---

addRooms

```
boolean addRooms(Transaction context,
                 String location,
                 int numRooms,
                 int price)
                 throws RemoteException,
                        TransactionAbortedException,
                        InvalidTransactionException
```

Add rooms to a location. This should look a lot like addFlight, only keyed on a string location instead of a flight number.

Parameters:

context - the transaction ID

location - the location to add rooms

numRooms - number of rooms to add

price - room price

Returns:

true on success, false otherwise.

Throws:
RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

RM.add(Transaction, RID, int, int)

---

deleteRooms

```
boolean deleteRooms(Transaction context,
                    String location,
                    int numRooms)
                    throws RemoteException,
                           TransactionAbortedException,
                           InvalidTransactionException
```

Delete rooms.

Parameters:

context - the transaction ID

location - the location to add rooms

numRooms - the number of rooms to delete

Returns:

true on success, false otherwise.

Throws:
RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

RM.delete(Transaction,RID,int)

---

addCars

```
boolean addCars(Transaction context,
                String location,
                int numCars,
                int price)
                throws RemoteException,
                       TransactionAbortedException,
                       InvalidTransactionException
```

Add cars to a location. This should look a lot like addFlight, only keyed on a string location instead of a flight number.

Parameters:

context - the transaction ID

location - the location to add cars

numCars - number of cars to add

price - rental price

Returns:

true on success, false otherwise.

Throws:

RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

RM.add(Transaction, RID, int, int)

---

deleteCars

```
boolean deleteCars(Transaction context,
                   String location,
                   int numCars)
                   throws RemoteException,
                          TransactionAbortedException,
                          InvalidTransactionException
```

Delete cars.

Parameters:

context - the transaction ID

location - the location to add cars

numCars - the number of cars to delete

Returns:

true on success, false otherwise.

Throws:

RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

RM.delete(Transaction,RID,int)

---

queryFlight

```
int queryFlight(Transaction context,
                String flight)
                throws RemoteException,
                       TransactionAbortedException,
                       InvalidTransactionException
```

Get the number of seats available. return the number of seats available

Parameters:

context - the transaction ID

flight - the flight number

Returns:

the number of seats available. negative value if the flight does not exists

Throws:

RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

RM.query(Transaction, RID)

---

queryFlightPrice

```
int queryFlightPrice(Transaction context,
                     String flight)
                     throws RemoteException,
                            TransactionAbortedException,
                            InvalidTransactionException
```

Get the flight price. return the price

Parameters:

context - the transaction ID

flight - the flight number

Returns:

the price. Negative value if the flight does not exists

Throws:

RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

RM.queryPrice(Transaction, RID)

---

queryRoom

```
int queryRoom(Transaction context,
              String location)
              throws RemoteException,
                     TransactionAbortedException,
                     InvalidTransactionException
```

Get the number of rooms available. return the number of rooms available

Parameters:

context - the transaction ID

location - the rooms location

Returns:

the number of rooms available

Throws:

RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

RM.query(Transaction, RID)

---

queryRoomPrice

```
int queryRoomPrice(Transaction context,
                   String location)
                   throws RemoteException,
                          TransactionAbortedException,
                          InvalidTransactionException
```

Get the room price.

Parameters:

context - the transaction ID

location - the rooms location

Returns:
the price
Throws:
RemoteException
TransactionAbortedException
InvalidTransactionException
See Also:
RM.queryPrice(Transaction, RID)

---

queryCar

```
int queryCar(Transaction context,
             String location)
             throws RemoteException,
                    TransactionAbortedException,
                    InvalidTransactionException
```

Get the number of cars available.
Parameters:

context - the transaction ID

location - the cars location

Returns:
the number of cars available
Throws:
RemoteException
TransactionAbortedException
InvalidTransactionException
See Also:
RM.query(Transaction, RID)

---

queryCarPrice

```
int queryCarPrice(Transaction context,
                  String location)
                  throws RemoteException,
                         TransactionAbortedException,
                         InvalidTransactionException
```

Get the cars price.
Parameters:

context - the transaction ID

location - the cars location

Returns:
the price
Throws:
RemoteException
TransactionAbortedException
InvalidTransactionException
See Also:
RM.queryPrice(Transaction, RID)

listFlights

```
String[] listFlights(Transaction context)
                     throws RemoteException,
                            TransactionAbortedException,
                            InvalidTransactionException
```

list existing flights

Parameters:

context - transaction id

Returns:
list of existing flights

Throws:
RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

addSeats(Transaction, String, int, int), RM.listResources(Transaction, RID.Type)

---

listCars

```
String[] listCars(Transaction context)
                  throws RemoteException,
                         TransactionAbortedException,
                         InvalidTransactionException
```

list existing cars

Parameters:

context - transaction id

Returns:
list of existing cars

Throws:
RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

addCars(Transaction, String, int, int), RM.listResources(Transaction, RID.Type)

---

listRooms

```
String[] listRooms(Transaction context)
                   throws RemoteException,
                          TransactionAbortedException,
                          InvalidTransactionException
```

list existing rooms

Parameters:

context - transaction id

Returns:
list of existing rooms

Throws:
RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

addRooms(Transaction, String, int, int), RM.listResources(Transaction, RID.Type)

---

listCustomers

```
Customer[] listCustomers(Transaction context)
                         throws RemoteException,
                                TransactionAbortedException,
                                InvalidTransactionException
```

list existing customers that have itinerary

Parameters:

context - transaction id

Returns:

list of existing customers that have itineraries

Throws:

RemoteException
TransactionAbortedException
InvalidTransactionException

See Also:

reserveItinerary(Customer, String[], String, boolean, boolean),

RM.listCustomers(Transaction)

---

start

```
Transaction start()
                  throws RemoteException
```

Start a transaction return a unique transaction ID

Returns:

a new transaction identifier

Throws:

RemoteException

---

commit

```
void commit(Transaction context)
            throws RemoteException,
                   InvalidTransactionException,
                   TransactionAbortedException
```

Commit a transaction

Parameters:

context - the transaction ID

Throws:

RemoteException
InvalidTransactionException
TransactionAbortedException

---

abort

```
void abort(Transaction context)
           throws RemoteException,
                  InvalidTransactionException
```

Abort a transaction
Parameters:
context - the transaction ID
Throws:
RemoteException
InvalidTransactionException