Distributed Transaction Application in Java or C#

Project Description, Version 2

> *This document replaces the preliminary project description that was handed out on January 3.*

The purpose of this project is to gain an understanding of the interaction between various components of a TP application, 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 functions can be done in parallel.

Common interfaces will be provided for components interacting with the client so that a common client can attach to and use any project's server. We will provide a basic test scripting interface for the Java version, to ensure minimal functionality. (We need a volunteer to port this to C#.) It is your responsibility to augment these tests (using the standard interfaces) to make certain that your service resists failure. If the test scripts turn out to be flexible enough, we may collect all student-written test scripts and apply them *all* to your project to see if it fails.

Your options for software development environment include the following:

- Microsoft J++ with RMI, which has the strong caveat that MS does not support RMI, and it would be necessary to integrate the Sun JDK with J++, or use the seemingly incomplete rmi package tucked away on the Microsoft Java web site.

- Sun's JDK. The tools are less powerful, but RMI is supported natively. Using Sun's JDK means that it wouldn't matter if you used NT or unix machines.

- Microsoft J++ with COM/DCOM. We don't recommend this option, because past students have found this more difficult than RMI, and we don't know enough about COM to help. A volunteer would need to port the RMI client to speak COM.

- The Microsoft .NET SDK with the C# language, using SOAP for remote procedure calls. This is a beta release, but we hear that it's stable enough to depend on. Since it's new, we won't be able to give much technical support, compared to Java. However, we suspect the implementation effort is lower than for Java, since support for distribution is more built-in.

  The .NET Framework SDK Beta 1 can be found in the following location:

  http://msdn.microsoft.com/downloads/default.asp?URL=/code/sample.asp?url=/msdn-files/027/000/976/msdncompositedoc.xml

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

---

1. We have provided a lock manager package/class. The operations to be supported include:

- lock(Xid, thingBeingLocked, read|write) 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.

The lock manager that we have provided needs to be modified so that it can convert locks, e.g.,

```
lock(xid, foo, read);
/* read foo */
lock(xid, foo, write);
/* write foo ... you would include error checking and exception
handling...*/
```

Keep in mind that other transactions may have read locks on foo, so deadlock is possible.

The implementation that we provide in C# does not implement the UnlockAll method. So if you're working in C#, you need to add this.

Lock managers are described in the textbook in Chapter 6, Section 6.2, and will be presented in lecture on January 17. Note that the lock manager is not required for part (2), and hence the latter can be implemented first if so desired. However, it makes sense to do it early, to give you experience with the Java or C# environment while you get warmed up to TP concepts described in class.

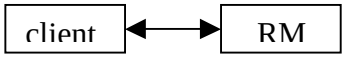
2. Build a simple Resource Manager.

The simple RM implements transactions. That is, it supports the methods start, commit, and abort, and all data access (read/write) operations are associated with a transaction. The operations to be supported are outlined in the attached Java interface. We know these assumptions sacrifice verisimilitude, but they should enable implementation of the persistent database relatively quickly.

Ignore atomicity to start with. Assume that the data is stored in memory, and also that 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. Since there is only one type of things, there is only one price. The net effect of { addCars(T1, 'San Diego', 4, $52); addCars(T2, 'San Diego', 7, $54); } should leave 11 cars at either $52 or $54, not 7 cars at $54 and 4 cars at $52.

It should be possible to query for which reservations the customer holds, and how much the customer should be charged. Don't bother with an account payment feature.

The system now looks like:



Initially, everything will be stored in one RM. Later, data will be partitioned into multiple RMs, where each RM stores some part of the data to be operated on.

Now add atomicity to the above interaction using shadowing: make a copy of the in-memory database, update it, then set the database pointer to the active memory image to the new one on commit. In step 4, this will be a disk image that will be copied, updated, and relinked (renamed).

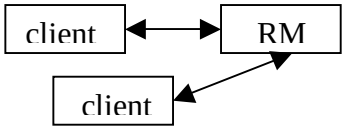
In the current system, the only failure to handle is an abort. Since the memory image is lost when the process terminates, there doesn't need to be any recover() method at this stage.

The Technical Interface methods 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 functionality of the Java and C# code that we provide are somewhat different, but don't materially affect the level of effort for this step.

3. Isolation. Combine parts 1 and 2. That is, the RM should lock data appropriately for each transaction, and unlock data when the transaction commits or aborts. We will test this implementation using multiple clients and a single resource manager. You might experiment with different locking granularities at this stage. The system will now look like:

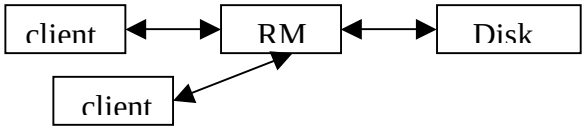


i.e., there are several clients interacting with the single RM.

4. *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 operations called with unknown (forgotten) transaction ids.

This will be accomplished using shadowing, described in the lecture notes on Database Recovery. See also, footnote on page 251 of text. There will be other references on shadowing.

The system will now look like

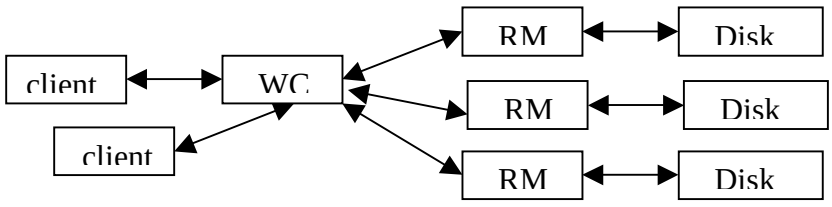


5. Implement a workflow controller. Workflow control is described in Section 2.4. The WC will be a front-end so that the eventual location (partitioning) of data on the RM's is not exposed to the client. To do this, the WC will support the same interface as the RM, and also the new function:

reserveItinerary(customer, flight_list, location, bCar, bRoom) method. A customer will have only one itinerary. This intinerary reservation is the sort of high level operation associated with a workflow controller, and can be implemented here. The parameters are: customer, the customer id from newCustomer; flight_list, a vector of integer flight numbers; location, the place where rooms or cars might be reserved; and bCar/bRoom, true if the customer wants a car/room reservation.

To start with, assume there is only one RM, and other than the above function, all functions are directly passed on to it.

In general, the goal is to have the following system:



For example, the client will ask the WC for a reservation on flights 435 and 534 and a rental car in St. Louis, and the WC will start a transaction, contact RM1 for flight 435 and make a reservation, contact RM2

for flight 534 and make a reservation, then contact RM3 for cars in St. Louis and make a reservation. As mentioned, at the moment, this is done using just one RM. There are more pieces to build before the WC can handle this functionality.

The workflow controller will be given the list of active RMs as command line arguments on startup.

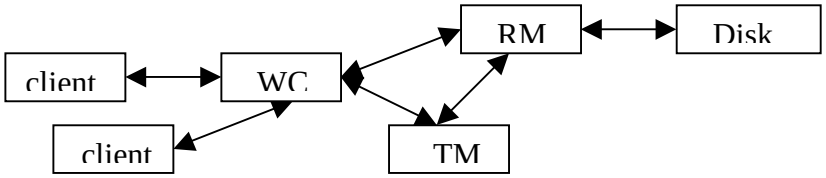
6. Implement a Transaction Manager. The TM supports the following operations: start, commit, abort, enlist. The transaction manager is present to coordinate the distributed transactions taking place across the several RMs involved. The interface of the TM is to be used as follows: whenever a request is made to an RM, it calls the TM's enlist method to tell the TM that it is involved in a transaction. The TM then 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 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 will be provided for the TM.

The workflow controller will need to be given the hostname 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 picture is now: (with several RMs)



7. Run multiple RMs. For example, flight, car and room reservations could be handled by separate RMs. The TM will maintain a list for each active transaction of, which RMs are involved, and implement one phase commit. The WC will decide which data requests / transactions go where. On a commit/abort request (which is forwarded to the TM), the TM calls the appropriate functions on all RMs involved in the transaction.

8. Modify the TM to store a list of which transactions committed, necessary for two phase commit below.

9. Implement two phase commit. At this stage, code the basics of two phase commit. That is, implement commit and abort under the assumption that messages never get lost or excessively delayed.

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

11. After you finish the first set of steps, we've thought about a handful of enhancements that could be made to make the project more enjoyable, or more applicable. This is certainly not an exhaustive list.

1. Logging instead of shadowing. Shadowing has the advantage of simplicity and the disadvantage of poor performance, sort of like bubble sort. Logging allows better performance, but is tricky. This sub-project would be appropriate for someone literate in Java and interested in performance issues, and might involve not implementing shadowing at all.

2. TM committed transaction list garbage control. 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.

Interface JavaTransaction.ResourceManager

public interface **ResourceManager**
extends Remote
Distributed Transaction System in Java. Class: ResourceManager Description: necessary ResourceManager interface.

abort(int)
Abort transaction.
addCars(int, String, int, int)
Addition and deletion of cars.
addFlight(int, int, int, int)
Add seats to a flight.
addRooms(int, String, int, int)
Add rooms to a location.
commit(int)
Commit transaction.
deleteCars(int, String, int)
deleteCustomer(int)
deleteFlight(int, int)
Delete the entire flight.
deleteRooms(int, String, int)
Delete Rooms from a location.
newCustomer()
queryCars(int, String)
queryCarsPrice(int, String)
queryCustomerInfo(int)
queryFlight(int, int)
queryFlightPrice(int, int)
queryRooms(int, String)
queryRoomsPrice(int, String)
reserveCar(int, int, String)
reserveFlight(int, int, int)
reserveRoom(int, int, String)
selfDestruct(int)
Call exit after a specified number of disk writes.
shutdown()
Shutdown gracefully.
start()
Start transaction.

start
```
public abstract int start() throws RemoteException
```
Start transaction. Returns: a unique transaction ID.

commit
```
public abstract boolean commit(int transactionId) throws RemoteException
```

Commit transaction. Returns: success.

abort

```
public abstract void abort(int transactionId) throws RemoteException
```

Abort transaction. Returns: nothing, but this may change.

addFlight

```
public abstract boolean addFlight(int Xid,
                                  int flightNum,
                                  int flightPrice,
                                  int flightSeats) throws RemoteException
```

Add seats to a flight. In general this will be used to create a new flight, but it should be possible to add seats to an existing flight. Adding to an existing flight should overwrite the current price of the available seats. Returns: success.

deleteFlight

```
public abstract boolean deleteFlight(int Xid,
                                     int flightNum) throws RemoteException
```

Delete the entire flight. deleteflight implies whole deletion of the flight. all seats, all reservations. It's undecided what will happen if a customer has a reservation on this flight, but one possibility is to delete the customer as well. The other possibility is to return failure. Returns: success.

addRooms

```
public abstract boolean addRooms(int Xid,
                                 String location,
                                 int numRooms,
                                 int price) throws RemoteException
```

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

deleteRooms

```
public abstract boolean deleteRooms(int Xid,
                                    String location,
                                    int numRooms) throws RemoteException
```

Delete Rooms from a location. This subtracts from the available room count without allocating the rooms to a customer. It should fail if it would make the count of available rooms negative. Returns: success

addCars

```
public abstract boolean addCars(int Xid,
                                String location,
                                int numCars,
                                int price) throws RemoteException
```

Addition and deletion of cars. Cars have the same semantics as hotels.

deleteCars

```
public abstract boolean deleteCars(int Xid,
                                   String location,
                                   int numCars) throws RemoteException
```

shutdown

```
public abstract boolean shutdown() throws RemoteException
```

Shutdown gracefully. When this RM restarts, it should not attempt to recover its state if the client called shutdown to terminate it. Returns: success

selfDestruct

```
public abstract boolean selfDestruct(int diskWritesToWait) throws
RemoteException
```

Call exit after a specified number of disk writes. Support for this method requires a wrapper around the system's write to disk command that decrements the counter set by this method. This counter should default to 0, which implies that the wrapper will do nothing. If the count is non-zero, the wrapper should decrement the counter, see if it becomes zero, and if so, call exit(), otherwise continue the write. This method is not part of a transaction. Returns: success

queryFlight

```
public abstract int queryFlight(int Xid,
                                int flightNumber) throws RemoteException
```

queryFlightPrice

```
public abstract int queryFlightPrice(int Xid,
                                     int flightNumber) throws RemoteException
```

queryRooms

```
public abstract int queryRooms(int Xid,
                               String location) throws RemoteException
```

queryRoomsPrice

```
public abstract int queryRoomsPrice(int Xid,
                                    String location) throws RemoteException
```

queryCars

```
public abstract int queryCars(int Xid,
                              String location) throws RemoteException
```

queryCarsPrice

```
public abstract int queryCarsPrice(int Xid,
                                   String location) throws RemoteException
```

queryCustomerInfo

```
public abstract String queryCustomerInfo(int customer) throws RemoteException
```

newCustomer

```
public abstract int newCustomer() throws RemoteException
```

deleteCustomer

```
public abstract boolean deleteCustomer(int customer) throws RemoteException
```

reserveFlight

```
public abstract boolean reserveFlight(int Xid,
                                      int customer,
                                      int flightNumber) throws RemoteException
```

reserveCar

```
public abstract boolean reserveCar(int Xid,
                                   int customer,
                                   String location) throws RemoteException
```

reserveRoom

```
public abstract boolean reserveRoom(int Xid,
                                    int customer,
                                    String location) throws RemoteException
```

All Packages Class Hierarchy This Package Previous Next Index