Distributed Transactional Persistent Reservation System

COMP 512 Fall 2015

Report for Project Deliverable III

Guillaume Labranche 260585371
Serguei Nevarko 260583807

General Design and Architecture of the System

Our system has 8 packages. The main ones are “client”, “server” and “middleware”. The rest hold classes that are shared between them. The following is a quick overview of what they are used for:

rmi (used by client, server and middleware):

The rmi package is used to communicate between the servers and the clients. It holds the type of objects that are sent and received over the sockets and does the sending and reading.

system (used by client, server, middleware, rmi, transactions):

This package holds the domain-specific classes for this project. For example Flight, Car, Room and Customer but also ReservableItem. It also has classes like “LocalResourceManager” that interact and manage those type of objects.

transactions (used by client, server, middleware and rmi):

This package contains the Transaction class that is used to represent the transactions and the TransactionsManager that does all the processing on the existing transactions. Transaction related exceptions are also included in there.

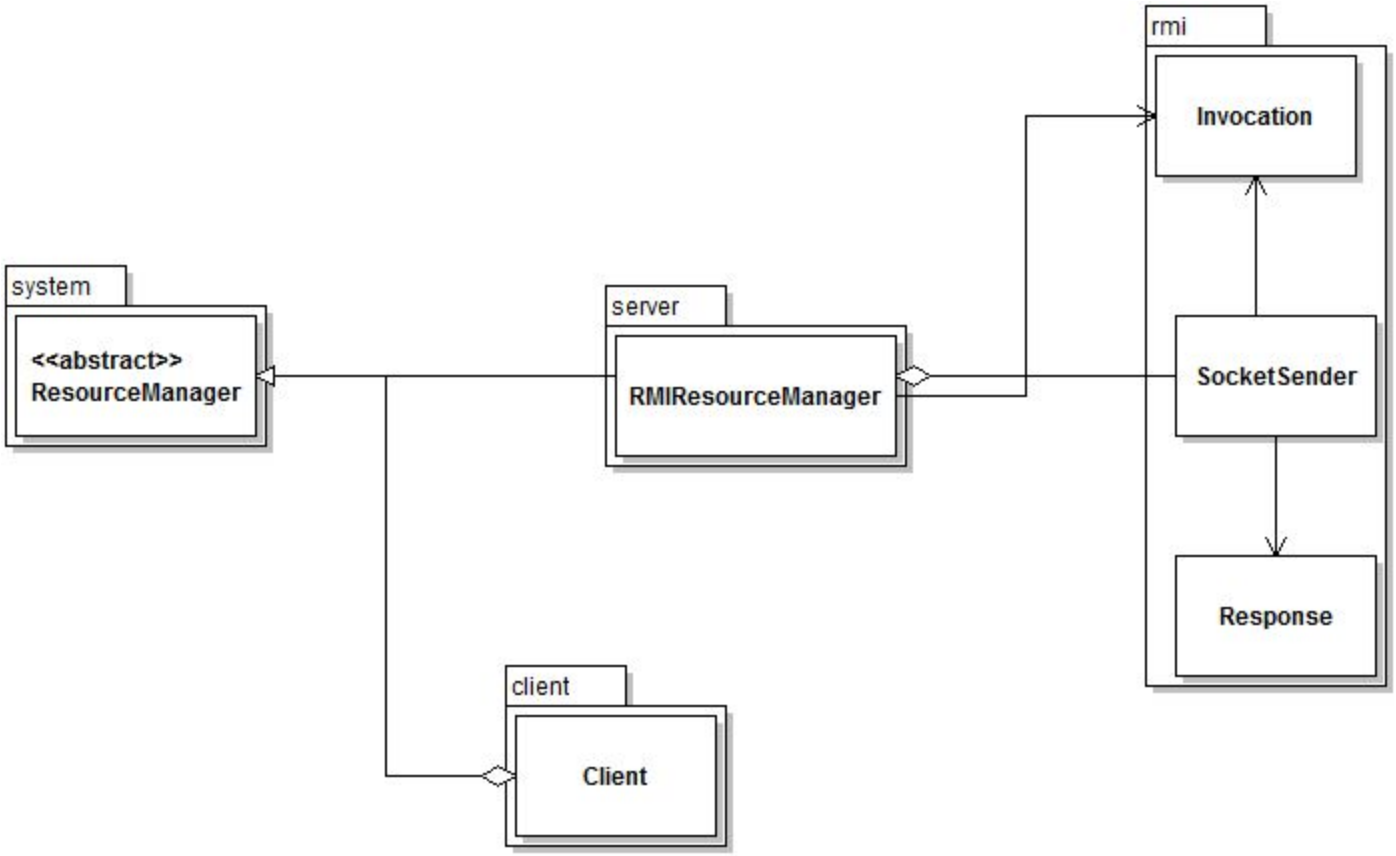
locks (used by client and transactions):

This package contains the LockManager and all the classes that it needs. All the resource locking in the project is done with this package.

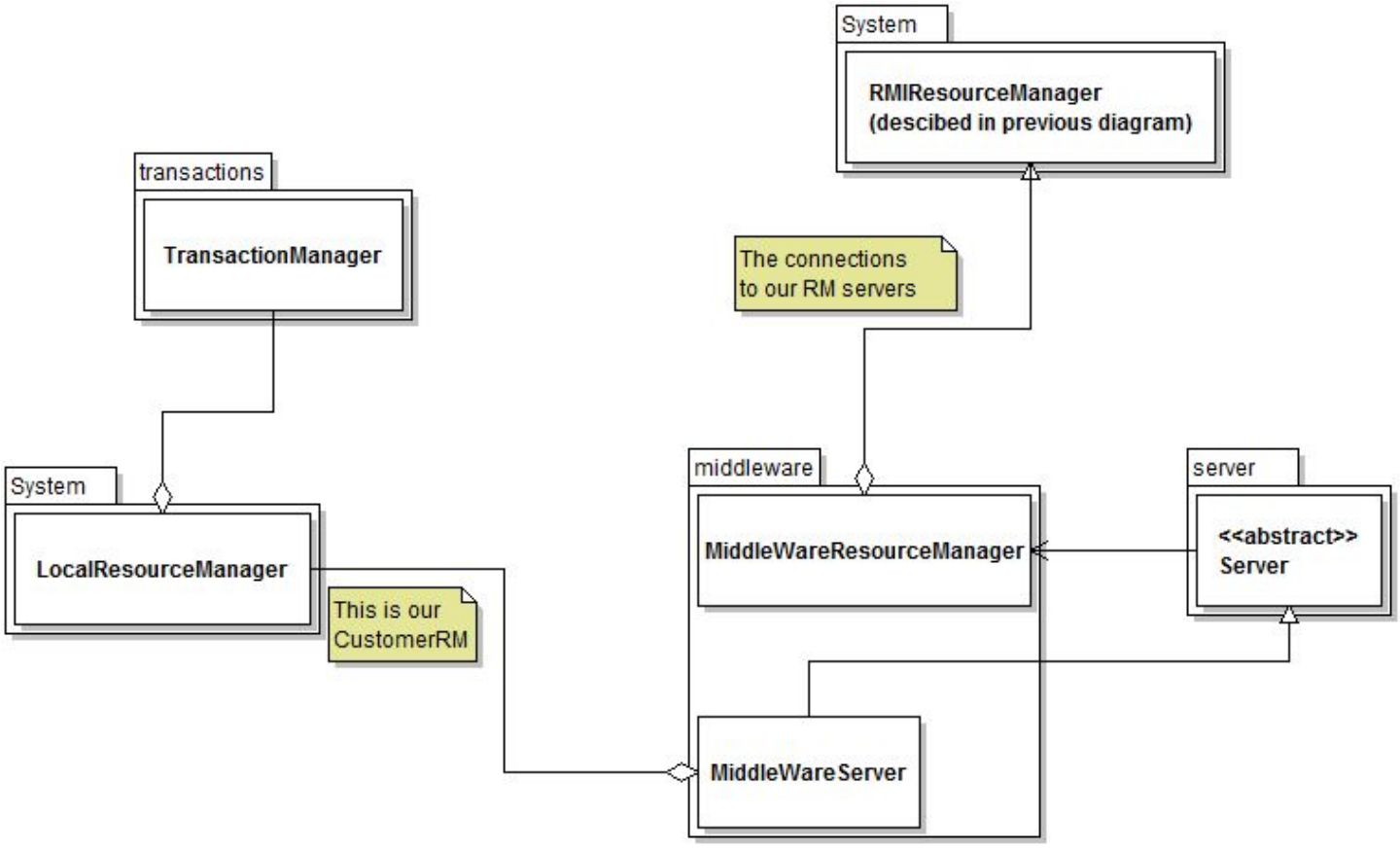
helper (used by system):

This package contains classes that implement some mechanics of the project. There is the RepairConnection class that continuously checks if the lost server is back, the SerializerHelper class that saves and loads serializable object into and from files and the VoteManager class that holds all the Yes/No votes and gives the outcome.

Three Main Packages: client, middleware and server

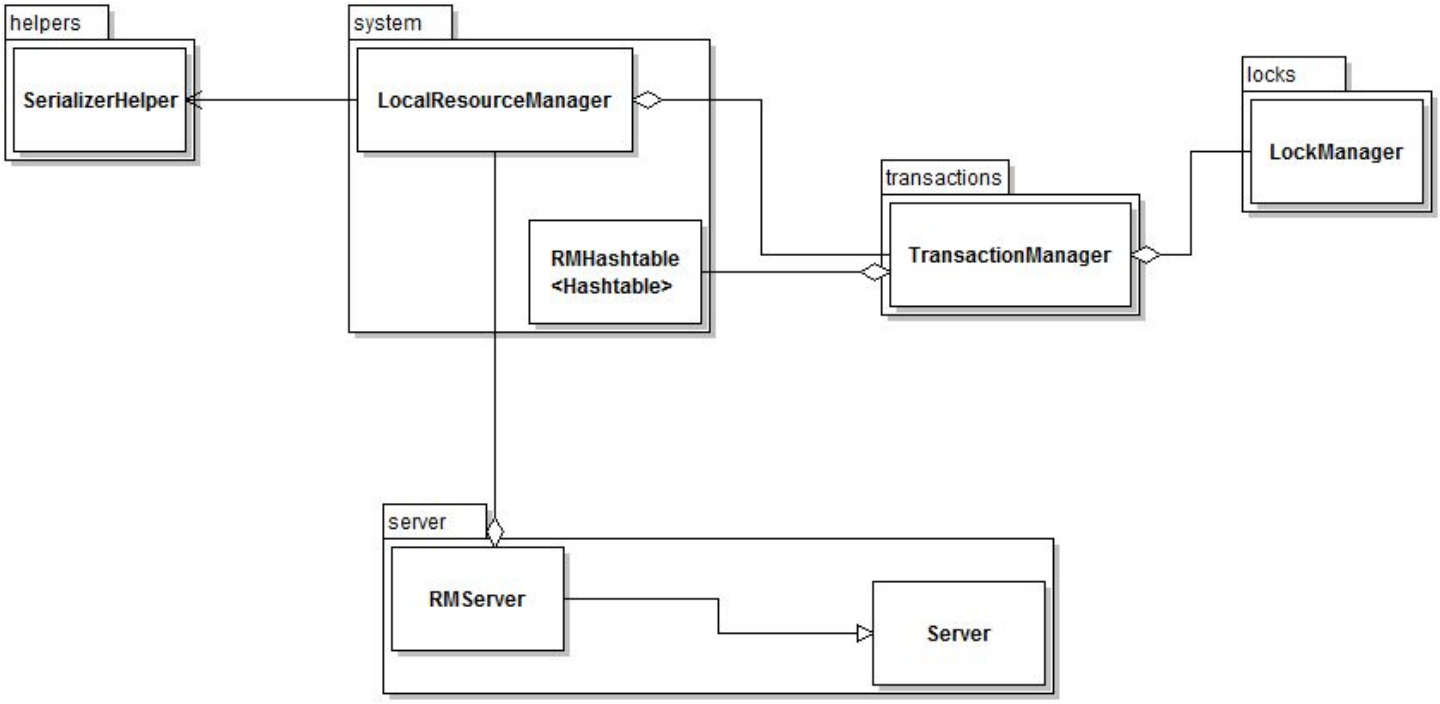


The client is pretty straight forward. The client holds an object of type RMIResourceManager. That object contains all the functions that the client is required to support (newflgiht, querycar, etc). When the user enters the command in the terminal, the Client calls the apropriate functions of that class and passes the function name with the parameters. Then the RMIResourceManager uses the SocketSender to send a Invocation to the middleware and waits for the response (that will then be sent back to the Client class). An Invocation is an object that holds the function name and the parameter values of that function. That object is serialized and passed through sockets. When a server responds, a Response object is sent back.



The middleware holds the connections with the client and the RM servers. The class MiddleWareServer extends the class Server so it is the one listening to the clients and communicating with them. This server uses the MiddleWareResource manager, the class that implements all the commands that are required from the project (addflight, queryroom, etc) and contains the connections to all of our RM servers to communicate with them. The MiddleWareServer also has the LocalResourceManager, is is like a RM server but internal. Inside the code we call it the customerRM and we use it to store and read all the data related to the customers.

Our customerRM is also has a transactionManager to manage all the changes made and store them in the appropriate transaction.



There are three RM Servers in the system. One for managing flights, one for rooms and the last one for cars. The RMServer class extends Server so it listens to incoming connections and messages and responds.

Our RMServer contains the LocalResourceManager that implements all the functions that have to do with the management of our data (flights, rooms and cars). It uses the SerializerHelper class to save its state to files so that in case it crashes it can recover the data when rebooted.

On each server we have a TransactionManager to have the support of transactions. That class can read and wrtie data, commit, abort and start a new transaction. It also uses the LockManager class and locks the resources to make sure that different transactions do not conflict with eachother.

All the data about our items are stored in the RMHashtable.

Data Design

The data is stored within a LocalResourceManager (originally ResourceManagerImpl) the same way as provided, in a RMHashtable. Because we must be able to serve concurrent client requests, many threads (Receiver) call methods on the LocalResourceManager possibly at the same time. Therefore we make use of Java's synchronized() keyword in order to synchronize access to mutable data. As explained further down below, we also store this database to file in order to recover from crashes and keep a persistent form of storage.

Implementation of Individual Features

Networking with socket-based Remote Method Invocations

We implemented our custom implementation of remote method invocations. On the sender side, we have a subclass of ResourceManager called RMIResourceManager who receives function calls and is tasked of forwarding them over the network. Upon a function call, it creates an Invocation and uses the SocketSender to send it through the socket using the Serializable interface. The SocketSender then reads from the socket's input stream for a Response object.
On the receiver side, we have a class Receiver who is attached to a ResourceManager. It reads the Invocation from the socket, unpacks it and calls the appropriate method of the attached RM using Java's reflection API (java.lang.reflect.Method.invoke()). An Invocation is done when the name and number of parameters matches a function found on the attached RM. It stores the return value of the method invocation into a Response object. It also catches any exception thrown by the called function and bundles it with the Reponse which is sent back to the socket.
On the sender side, we receive the Response and throw the bundled exception if there is one. If not, we return the bundled return value, which RMIResourceManager returns in the proper type. This allows seamless remote method invocations from the client to the RMs. In fact, the MiddlewareResourceManager is a redirector who uses this RMI system from both ends. All that is required is careful setup with the sockets and the right classes but once that is done, the newFlight(...) function can be written in one line essentially: flightRM.newFlight(...).

Transactions

We made the system transactional by having a TransactionManager in each LocalResourceManager (one per server). This removed the need to exchange lock information between the different RMs, something which can be error-prone and inefficient if the network is slow. Because each RM's content does not overlap with another RM, it was entirely possible to have a LockManager on each RM who did not know about the locking state of other resources that did not belong to itself.
The concurrency control method we used is two-phase locking. Each transaction keeps a local copy of items is has read and written without affecting the real database. During a transaction's lifetime, READ and WRITE locks are acquired (or converted from READ to WRITE) but only released all at the same time during the commit or abort phase.

Persistent storage and recovery

On the RMServer side the persistence storage is based on serialization of objects.
Every time there is commit prepare phase, a new transaction with the state PREPARED is added to the transaction list and all the transactions with that state are saved in a file by serializing the array.

Every time an RM commits a transaction, it updates the items of that server and the entire hashmap is serialized into a file called items.

Each RM Server will generate those two files in the appropriate folder: flight, room and car. And the next time they are launched they will search for those two files after setting up and load the data if they exist.

On the Middleware we also have an internal RM that we called customerRM. It works the exact same way as the Server RM by generating the two files in the folder called “customer”.

Two-Phase Commit

Because of the way our system is connected using sockets, it made the 2PC implementation very smooth. When the middleware does a request for vote, it can already see whether any socket connection is healthy or lost. If all are connected, it can then request a vote. Once we have confirmation that everybody has prepared for commit, we then alert all RMs that they can indeed commit. The use of permanent socket connections, whose connection status can be verified instantly, removed the need for tricky timeout mechanisms. Therefore if a 2PC is bound to abort, it will do so very quickly.

Breakdown of Common Client Actions

start (a new transaction)

The client calls his RMIResourceManager’s start method, which is then mirrored on the MiddlewareServer’s MiddlewareResourceManager. In order to keep the same TC (Transaction Count) throughout, we first call start on the customer RM (recall that this one is local to the middleware) and obtain the new transaction ID. We then call start(tid) on the flight, car and room RM, which specifies what ID is used. In every LocalResourceManager, the TransactionManager handles creating a new transaction.

newcar

The MiddlewareResourceManager receives the function call from the Client. It then forwards the call to the car RM only and returns the result. The car RM requests to its LockManager a WRITE lock on item with key “car-1”. If no other transaction currently holds a lock of any type on it, the lock is granted right away. Otherwise, the RM’s thread for the client executing the command will block (wait) until the item is unlocked (and consequently the client’s terminal will also block as it is waiting for the reply from the RM through the middleware).

newcustomer

The MiddlewareResourceManager first uses the customer RM to create a new customer and obtained an ID. It then forwards the call with a specified ID so that the same customer with the same ID gets created throughout the system. Contrarily to car-related commands, the middleware does not forward the request only to the customer RM because in order to reserve flights, cars or rooms, an RM must be aware of the clients in existence. Therefore customer commands are forwarded to all RMs.
One other difference to note is that the customer RM is actually stored within the MiddlewareServer instead of a separate RM. This is because customer data needs to be stored on every RM anyways, so making it a standalone server RM is not useful. In fact is it quicker to have it sit in the middleware because there is no latency for access.

commit

The MiddlewareResourceManager calls the commit method on the customer RM. At the LocalResourceManager level, it calls commit on the TransactionManager, who then applies all the changes of the transaction to the real database. Then it unlocks all locks and calls the commit on all RMs involved with the transaction.

commit2pc

This is a two-phase commit where we first ask sequentially all involved RMs whether they are ready to commit. This also makes them prepare the commit (save to persistent storage) or reject the commit. If it was a success for all, we then call commitFinish() on all involved RMs.

Problems Encountered

Concurrency

Dealing with concurrency is something that has to be done with a lot of planning and careful implementation. Whenever threads share the data, there is risk of corruption. We first had to design how servers would behave with regards to listening for new incoming connections versus being connected to a socket. Further down the road, after much refactoring and implementing new functionalities, we hit a roadblock. Nothing worked as intended and we discovered that our view of the multi-threading and resource-sharing was flawed. We had to then draw by hand the diagrams you see in this report in order to get a full grasp of what was really going on. We then had to consolidate and streamline the access of shared resources by multiple threads. A lot more had to be synchronized than we thought at first: the LocalResourceManagers, the socket I/O, etc.

LockManager

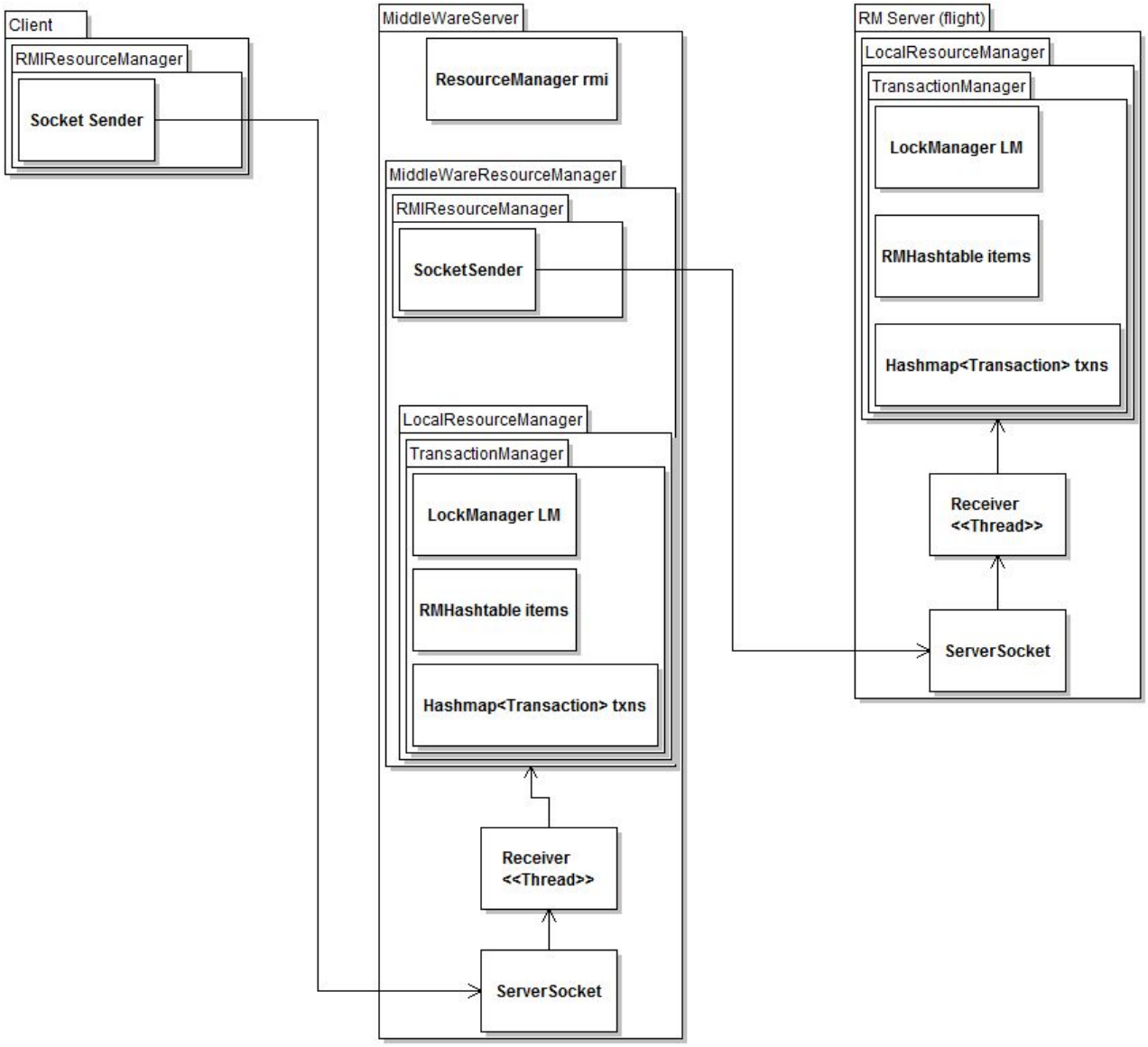
Although we used the code that was provided, made the required modifications, and tested its behaviour, the LockManager ended up causing us a lot of issues. And because we assumed it was working correctly, we always believed it was an error in using it rather than in its very implementation. As it turns out, there were many bugs that had to be cleared from it until we got a strong, reliable LockManager that behaved as intended.

Transaction Protocol

At first, it was not very clear what were our options for implementing the transaction system. After discussing for a while, we discovered a few approaches and had to decide which one was the best for our needs (we were using sockets unlike most of the class). It is a bit of luck that made us choose the right protocol the first time.

Figures and Diagrams

Remote Method Invocations and Data Location



Multi-Threading and Network Setup