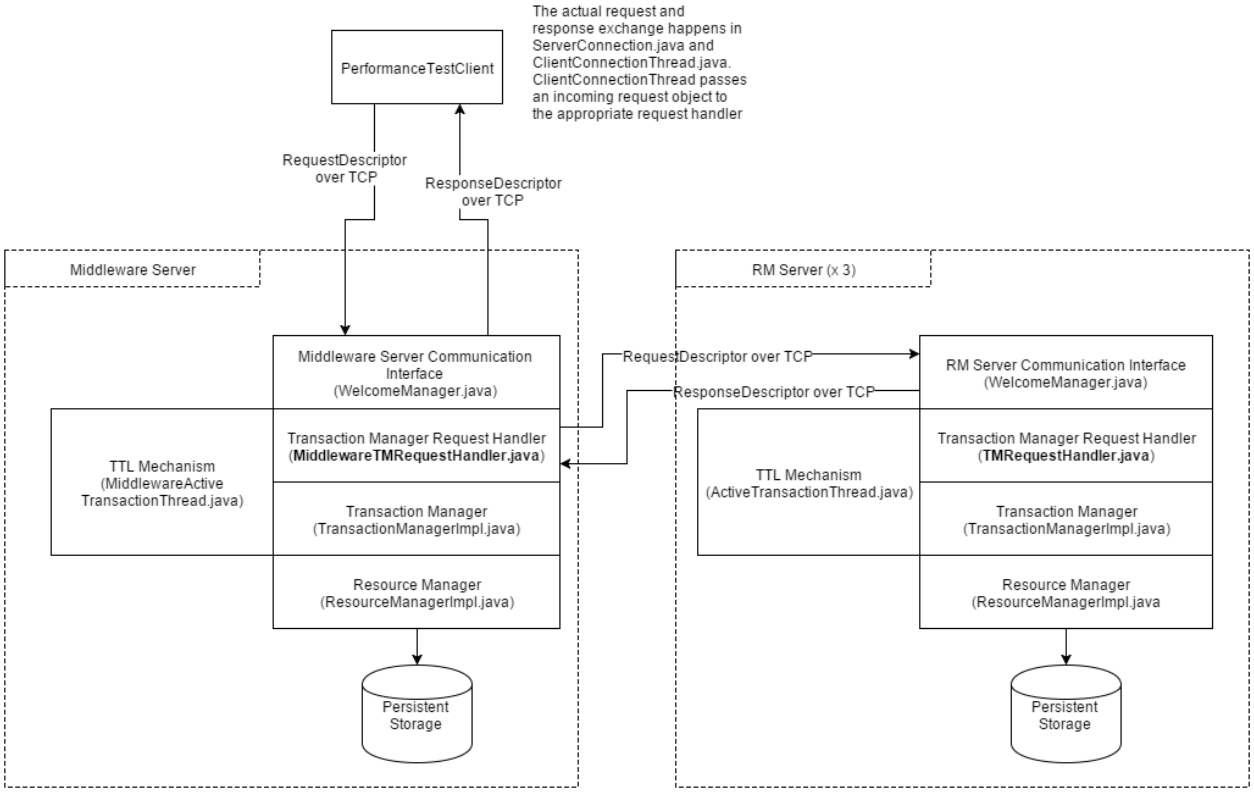
COMP 512 Project Deliverable 2 Report

Introduction

This description uses the attached architecture diagram as a reference. The diagram is also here:



System Architecture

Communication

Our architecture from the first deliverable was sufficiently robust that adapting it to the transaction model was fairly straightforward. We had to build two new kinds of request handlers (TMRequestHandler and MiddlewareTMRequestHandler) which fit nicely into the existing communication component. The description from the first deliverable is summarized below:

Communication between client, middleware, and RM servers is achieved using descriptor objects passed over TCP. We have one descriptor for requests (RequestDescriptor) and one descriptor for responses (ResponseDescriptor).

Concurrency

The TransactionManagerImpl class handles concurrent requests via locking. It also guarantees thread safety on the start transaction, abort and commit operations.

There is also the TTL mechanism, implemented by ActiveTransactionThread and MiddlewareActiveTransactionThread which aborts transactions after a time defined in the abstract

super-class, AbstractTTLThread. These classes are all thread safe by necessity because they contain a thread to achieve the TTL behaviour but also allows the transaction manager to remove transactions by request.

Special features

In addition to implementing the atomicity requirement of transactions, our system also persists information to disk. When the RM Servers restart, they will load back into memory the last committed RMHashtable. This means that if our system crashes in an inconsistent state, it will recover to the last stable state.

Problems encountered

- Our architecture includes a TransactionManager on the middleware which also required the TTL mechanism. We wanted to ensure that the RM servers would be able to remove transactions independent of the middleware TTL mechanism because we wanted to support the case during our tests where we would replace the middleware server while there were transactions still pending on the RM servers. We solved this problem by using a TTL mechanism at each RM server and the middleware server that performed cleanup independent of each other. So, assuming network latency was less than the 1 second interval of TTL checks, we could guarantee a timed out transaction would be removed within 1 second across the entire system. However, our initial solution involved replicating a lot of code at both the middleware and the RM server. We were able to reduce the amount of code replication using an abstract class.
- During our multi-client testing, we ran into exceptions from socket creation. The problem was that with multiple simulated clients (in different threads) on the same machine, occasionally two different threads will attempt to create a socket to the same address (the middleware). This caused an exception. We resolved this problem by making the method that concerns the creation of a socket a synchronized block, allowing only one thread at a time to be using the method that sends and receives over a socket.

Testing procedure

During the programming portion, we tested mainly by hand. Initially we ran programs that made calls against the transaction manager to ensure that its behavior was correct and later we used the command line client to test the distributed transactions and the TTL mechanism.

The tests we ran for the performance evaluation are explained in the performance evaluation document.