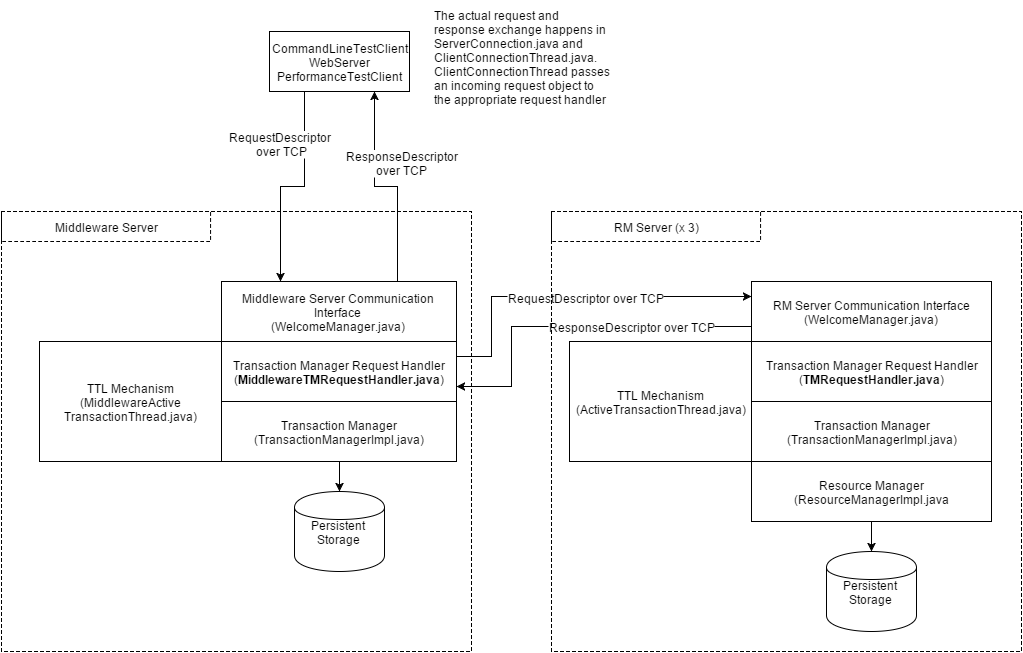
COMP 512 Project Deliverable 3 Report

# Introduction

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



# System Architecture

## Communication

Communication between client, middleware, and RM servers is achieved using descriptor objects passed via ObjectOutputStream and ObjectInputStream over TCP. The descriptor objects allows us to encode strongly typed data in a meaningful way. We have one descriptor for requests (RequestDescriptor) and one descriptor for responses (ResponseDescriptor).

The RM servers and middleware uses a WelcomeManager object which listens for new client connections on a ServerSocket and creates a Socket wrapped in a ClientConnectionThread when a new client is accepted. ClientConnectionThread forwards requests it receives to the appropriate request handler. For the RM server it is TMRequestHandler and for the middleware it is MiddlewareTMRequestHandler.

The client and middleware creates and opens a Socket to the server based on command line input or as parameters when the program is opened. The hostname and port of the server is stored in a ServerConnection object which contains an operation for sending out a request and returning the response. This operation also creates a new socket for each request which creates a ClientConnectionThread on the server-side. Since the middleware must connect to multiple RM servers, it keeps track of its server connections using ConnectionManager.

## Concurrency

We achieve concurrency by creating a new thread (ClientConnectionThread) for each new request to the middleware and RM servers via the WelcomeManager. Thus one client can make a request while another client is still waiting on a response.

The TransactionManagerImpl class handles concurrent requests via locking. It also guarantees thread safety on the start transaction, abort and commit operations. All methods in the ResourseManager have been made thread-safe, so heavy loads will not damage data integrity.

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.

# Transaction Support

This system supports transactions though two phase locking, using the provided LockManager class. When a transaction is started, the RM data is written to disk for later recovery in case of abort or a crash.

# Two Phase Commit

Transactions are committed using two phase commit. It works with the middleware acting as coordinator and the RMs acting as participants. After the second deliverable, the TransactionManager was removed from the middleware, so it does not vote in the commit vote. Recovery is implemented on both the middleware and RMs, through a class called CommitLogger, which writes log information to disk that is used to recover properly. RMs write their data to file at transaction start, as well as after sending a YES vote in 2PC, so that it could commit if it crashed and recovered.

# Special features

This system can handle crashes at any point during execution, during a transaction, and in the middle of the two phase commit protocol. We also have a web server and web client that can make calls the reservation service.

# Problems encountered

* In the two phase commit recovery procedure, when a participant crashes after sending a yes vote, it cannot recover independently, and has to ask the coordinator for the vote result. Our problem was that in our design the RMs did not know where the middleware was located, it was the middleware who connects to the RMs. Rather than making the RM have to know the location of the middleware, it waits to ask for the votes it needs to recover until the middleware connects to it for some other reason. When the middleware tries to take any action on an RM waiting for votes, it gets a response saying so and it then sends the vote that the RM requested. This continues until the RM can recover, then it executes the original request from the middleware.

# 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 multiple command line clients to test the distributed transactions and the TTL mechanism. Testing the recovery of servers during the two phase commit process was done by spoofing log files to simulate crashes that would be recovered from. This was much faster than finding a way to crash the servers during normal execution, though that functionality was added for the live demonstration.

Deliverable 2 Performance Evaluation

# Testbed description

## Client

The client consists of series of basic transaction operations (i.e. a sequence that creates a flight, queries it, and reserves it). The client also has a main method that runs experiments by invoking the various transaction elements in different ways. For example it computes the average transaction time for n transactions with a load interval of m milliseconds. The client can also take a delay parameter which is used in the multiple clients experiment (figure 1).

## Measurement techniques

The time for transactions is measured by taking the system time before the transaction is invoked and the system time after and using the difference.

To ensure that size of data set is not an issue, the data on the RMs was reset for every data point (e.g. for each transactions per second value, the whole system was reset).

## Transactions used

We have two primary transaction sequences, one for a single RM test and one for a multiple RM test.

The single RM test is the following transaction:

NEWFLIGHT, QUERYFLIGHT, RESERVEFLIGHT, COMMIT

The multiple RM test is the following transaction:

NEWFLIGHT, NEWCAR, NEWROOM, COMMIT

# Experiments conducted

We conducted two experiments.

Experiment 1 (figure 1), Increasing load (i.e. transactions per second to middleware) across multiple clients (10 clients) and measuring response time for both single and multiple RMs

Experiment 2 (figure 2), Increasing number of consecutive iterations of a transaction submitted to middleware from a single client for both single and multiple RMs

# Performance figures

The performance figures begin on the following page.

Figure 1 – Multiple Clients

## Observations (Experiment 1)

Both Multiple and Single RMs start out relatively flat, then increases quickly before plateauing and changing at a slower, although more volatile pace. Multiple RM always has a slower average response time than Single RM for the same number of transactions submitted to the middleware per second.

## Analysis (Experiment 1)

* The higher response time for multiple RM is likely explained by the need to start “sub-transactions” on each RM and consequently, the higher number of IO operations (each RM start transaction / commit operation involves an IO operation on disk).
* The rapid growth is likely occurring around the point at which the delay between transactions (for controlling load) is reduced to 0 and operations are now reaching each of the RMs concurrently, also potentially triggering a competition for resources as only one operation may be sent from the client at a time due to the limits of Java Sockets.
* The rapid growth occurs earlier for Multiple RM because the requests take longer to serve on average and the concurrency on the RMs arrives earlier.
* The volatility after the rapid increase is likely the result of non-deterministic behaviour when transactions / operations become concurrent and must compete for resources. For example, a transaction may simply be unlucky and starts before other transactions but ends up finishing after the other transactions, dragging up the average response time.

Figure 2 – Single Client

## Observations (Experiment 2)

Both Multiple and Single RMs are somewhat consistent, then increases at a faster pace, although there is volatility in both cases, Multiple RM appears to be more volatile. Multiple RM has a higher response time than Single RM for the same number of iterations.

## Analysis (Experiment 2)

* The higher response time for multiple RM is likely explained by the need to start “sub-transactions” on each RM and consequently, the higher number of IO operations (each RM start transaction / commit operation involves an IO operation on disk).
* The appearance of consistency near the beginning may be noise. If so then the general trend upwards could be explained by increasing load on disk resources. As the number of iterations grow, so does the total size of the serialized file being written to disk (this is simply because the transaction was set up in such a way that each additional iteration of the transaction would add more data to the dataset). It is also possible other hardware limitations are at play here.
* The gap between single RM and multiple RM remains fairly constant which would suggest the cost of accessing multiple RMs scales with the transaction and not with the number of consecutive iterations.

# Conclusions

## Potential bottlenecks

Given the design of our system, it is likely the largest bottleneck is the IO operations involved in starting, aborting, and committing transactions.

Certainly, accessing multiple RMs incurs a penalty. However, it is unlikely this penalty is the result of network constraints as the whole system was evaluated on a single machine and each operation is already sent as a separate message. It is also unlikely that this is the result of CPU constraints because the multiple RMs are all running in the foreground anyway. So the only remaining difference between a transaction that involves one RM and a transaction that involves many RMs is the number of places the middleware must start (or enlist an RM for) a transaction and similarly how many places the RM must commit or abort. The commit mechanism is what likely ends up being the most costly because it requires that expensive disk operation.

Although it is difficult to speculate on the behaviour of the hardware (there may be optimizations done by the OS that I do not know about), figure 1 suggests that concurrency is not a problem. Namely that the Multiple RM time is about double the Single RM time both at the stable early part (low load) and also double after the rapid increase has occurred for both.

## Where is the time spent

It is likely that the time spent in the middleware is larger than that of the RM servers. This is because there is overhead on top of the middleware’s own RM for customers and coordinator TM.

We see this in figure 2 where the multiple RM that involves 3 RMs and 1 middleware versus the single RM that involves 1 RM and 1 middleware. Despite doubling the number of transaction managers and I/O operations (the middleware also writes to disk on commit), the response time increases by less than double. Indeed, the gap between Single and Multiple remains fairly constant suggesting at least some portion of the time is not spent in the RMs.