Project Deliverable III

COMP-512 Distributed Systems

Fall 2013

Navjot Singh Nan Zhu
McGill University
Department of Computer Science

Abstract

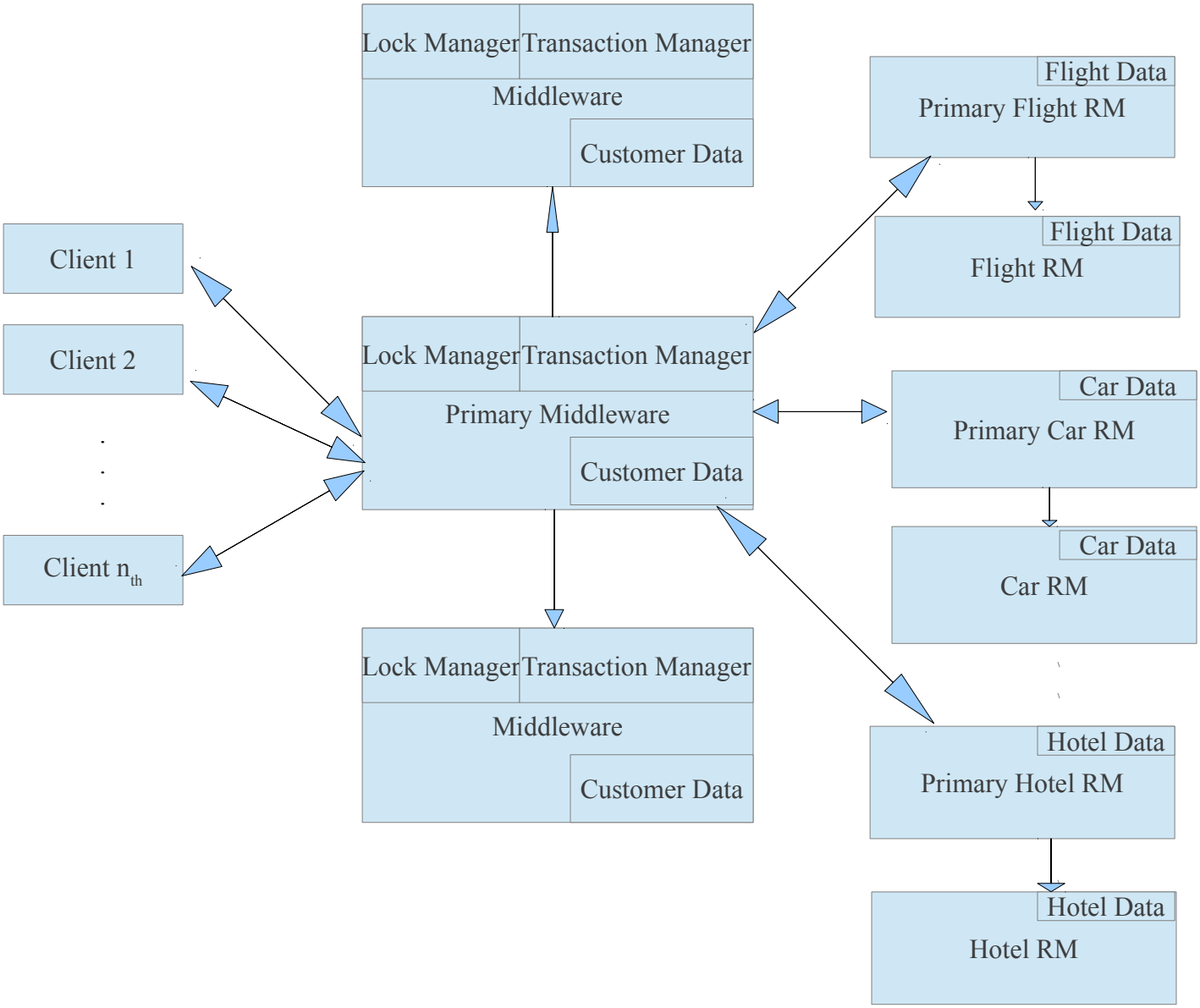
This report presents the implementation of concurrency control and replication mechanism on the distributed reservation system which includes a middleware, flight resource manager, car resource manager and hotel resource manager.

Architecture

We use Java RMI for the distributed reservation system. Concurrency control and transaction management module were implemented. The implementation of replication for fault tolerance uses JGroups library for syncronizing state changes between server clusters. Our system can be tolerant to as many as N – 1 failures (N is number of servers in a cluster). For the replication, we choose primary copy for its certainty and adopt eager approach to guarantee the strong consistency.

We will discuss the test cases covering all possiblities of failures as well as some tricky problems like concurrency control during the synchronization of data. At the end of this report, we evaluate our system with an automatic workload generator. This performance evaluation is conducted with the implementation without fault tolerance.

Figure shows the architecture of the system.



Components

The main components in this deliverable contains, RepClient, MiddlewareServer and ResourceManagers. They are implemented with the awareness of the primary/secondary in the server end setup.

RepClient

This client is started with the loading the server list. It sends requests to the primary middleware (the primary selection algorithm would be introduced later); on failures, it switches to a new primary and resends the requests automatically. Besides that, we implemented a background thread in client end to periodically check the dead middlewares and reconnects to them when they are recovered.

Middleware
Middleware uses JGroups to send state changes and synchronize messages. In our implementation the transaction and lock managers are installed in middleware, and at the same time, it manages the customer data, so when synchronization it has to keep consistency among these three kinds of data.

Middleware serves the requests from the client but also takes the role of client of ResourceManagers. It resends requests on faults to the new chosen Primary ResourceManagers and starts a background thread dynamically checking any dead resource managers and reconnects to them when they are recovered.

ResourceManager

Flight, Car and Hotel ResourceManagers are implemented by extending a new base class, named TransGCResourceManager. This base class implements all JGroup-related functionalities by realizing MessageDispatcher interfaces. The subclasses, like Flight-, Car- and Hotel- ResourceManagers utilizes the methods in base classes to send state change and synchronize messages.

Algorithms

Primary Selection

For choosing a primay among the processes with the same functionality, we utilize the View concept in JGroup and take the process with the address in the first position in the view member list as the primary. Each server process (Middleware and ResourceManager) implements boolean isPrimary() which is called by the client (RepClient and Middleware) while electing a new primary.

Synchronization

For each request at primary server (Middleware and ResourceManager), it sends a state change message to the secondary servers. When a new member joins the group, it sends a sync request to primary server and receives the complete state of primary as a response.

Fault-tolerance

During server failure the acting clients (client/middleware) looks for a new primary server (after timeout or connection errors) and resend the requests. The tricky scenario is that if the primary server has sent the state change message to the secondary server which is the new primary now, the update requests (like reserve, delete, add, etc) are actually received for two times by the new primary. To solve this issue, we assign operationIDs to each request and keep a list of operations already performed. We check for transactionID and operationID to verify if an operation has been already performed.

We consider the case about the possible state changes happening during the synchronization. If a write request is processed at the same time when we are sending a synchronization response for the new joined group member, the received data in new member side would be undetermined. We use the Java built-in “synchronized” keyword to exclusively access the hash-table. If write comes during synchronization, it will be blocked till all the data is synchronized .

We also considered the fault scenario that the new joined member sends a synchronization message but the primary fails before the message arrives. If primary middleware fails during synchronization or secondary does not receive the complete state in 10 secs, it will resend the request to a new primary. If primary resource managers(Flight/Car/Hotel) fails during synchronization or secondary does not receive the complete state in 5 secs, it will resend the request to a new random resource manager.

Failure Test Cases

We considered and tested our system for below failure cases:

1) crash at Resource managers before/after executing requests.

2) crash before executing request at middleware

3) crash after executing and before sending the response

4) crash after executing at middleware but before sending to RMs.

5) crash after executing at primary middleware but before sending to secondary middlewares.

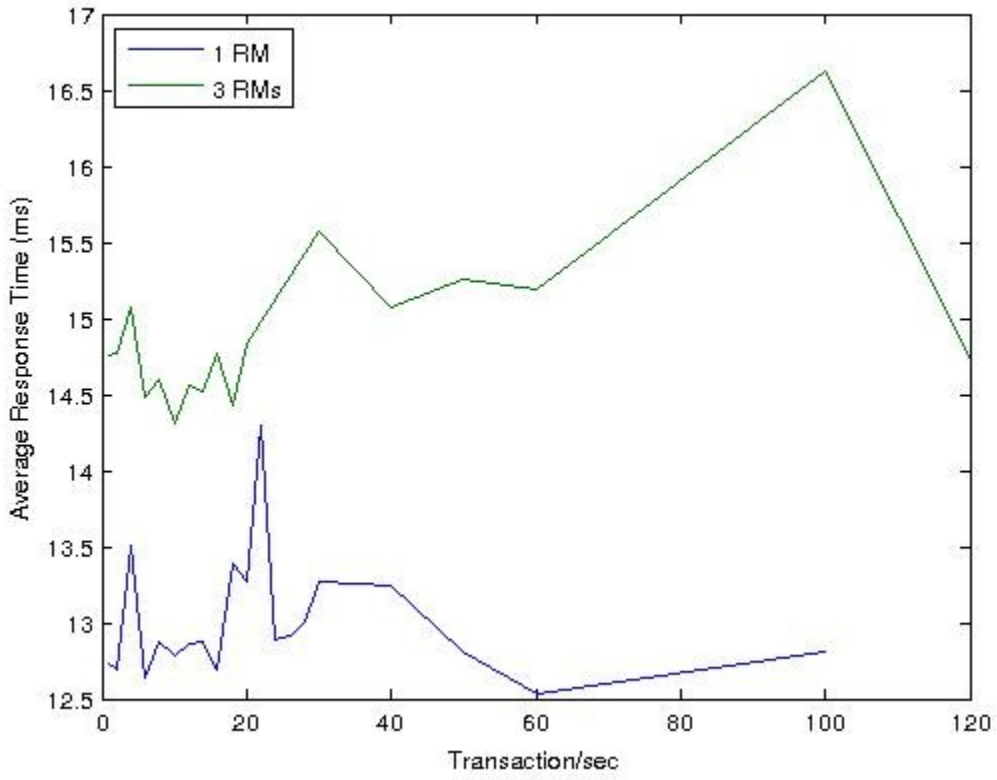
Results
It was observed that the data remained consistent during all of the above failure cases.

Performance

We aim to find a bottleneck for the responses in the distributed system without fault tolerance. For measuring response times and change the rate and type of transactions, we implemented a new client (GreedyClient) and a data generator. The data generator was run first, which created a large data set of objects(customer,flights,cars,rooms). For the GreedyClient the transaction rate and the number of transactions are given as arguments and the average response times are noted for that particular transaction rate.
As the client is blocking there is an upper limit of the transaction rate for the client. It cannot send more transactions than this rate. Even if we give a large transaction rate in the argument list, this rate could not be achieved.

Results:
Each transaction contains three reads and three writes.

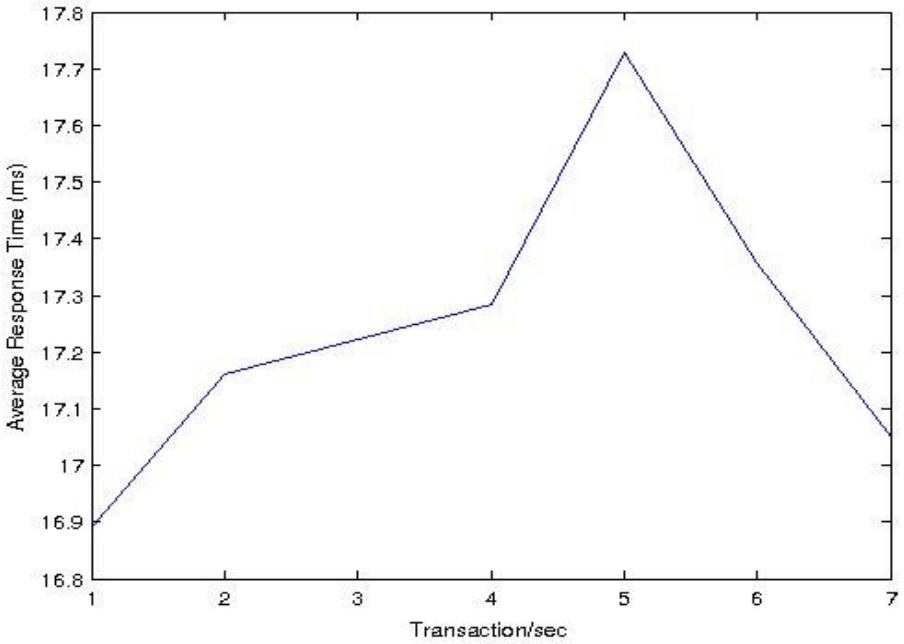
1)Test for single client.



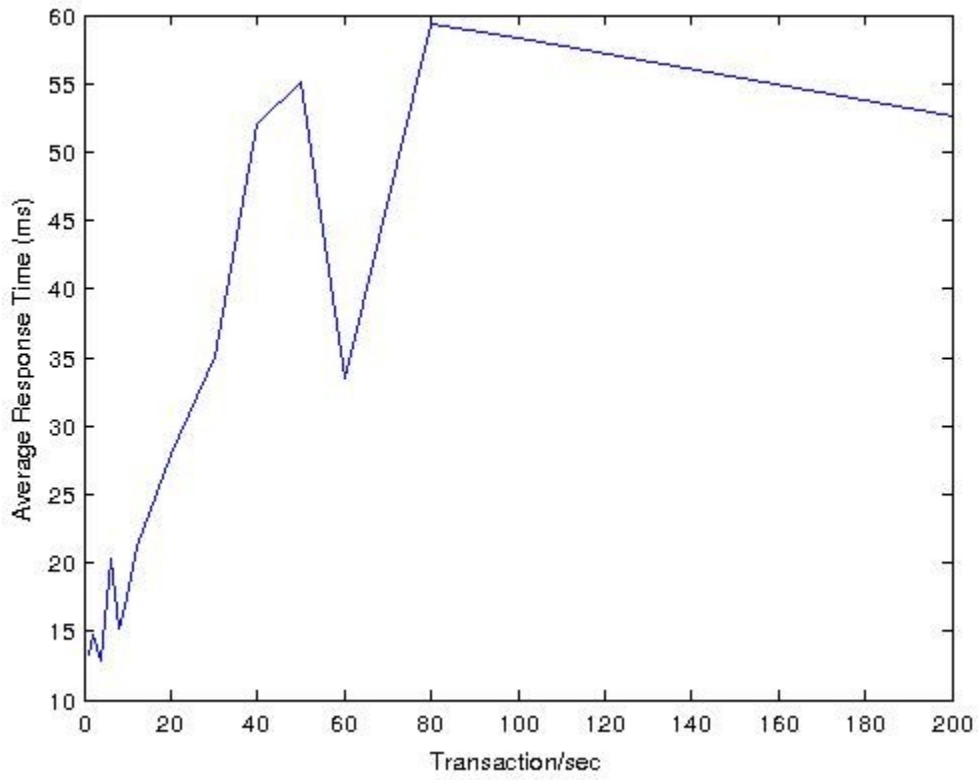
Due to the upper limit on client sending rate, we could not find any bottleneck on the middleware or resource manager. There is no concurrency in this case so there is no concurrency bottleneck. The case for 3RMs has larger response time because the request is sent to 3 different RMs and one of the RMs might have more latency. Moreover for single RM the object might have been cached and the response time for subsequest requests would be better.

2)Test for multiple clients and multiple resource managers.
We tested for **10 clients and 10 objects**(customer,flights,cars,hotels) at 1 transaction/sec.
Average response time was 51.1407 secs. This is so high because there is a concurrency bottleneck as there are very few objects in data set.

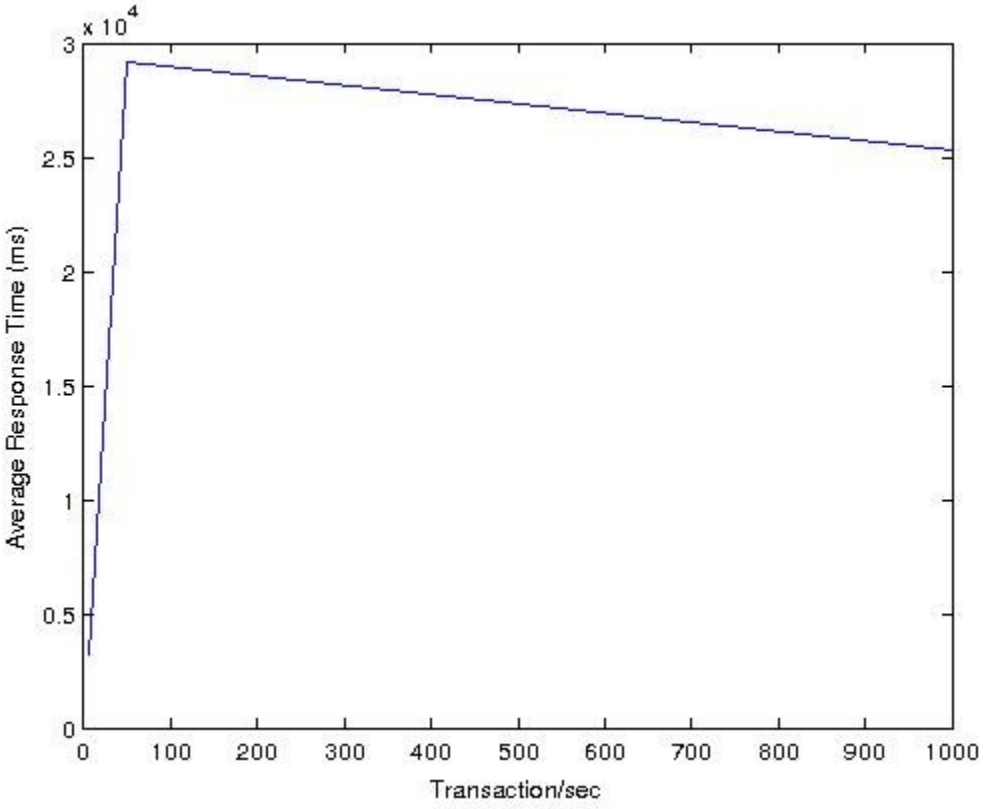
Again we tested for **10 clients and 50 objects**. Below is the figure for this case. At 8 transactions/second, a concurrency bottleneck is reached and the average response time shoots up to 30 secs.



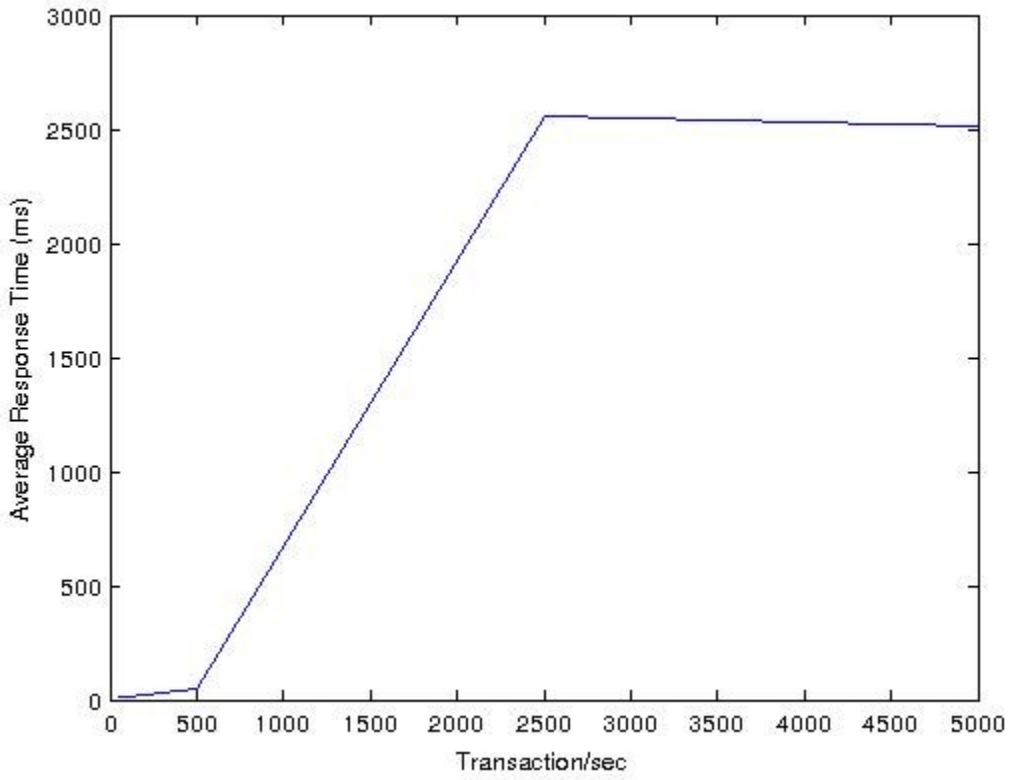
Now for **10 clients and 1000 objects** in data set. Response times increase mainly due to concurrency control.



Now for **50 clients 1000 objects**. Concurrency bottleneck hit at around 10 trxns/sec.



For **50 clients and 10,000 objects**. Response times increase mainly due to concurrency control.


3000
2500
2000
1500
1000
500
0
Average Response Time (ms)
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Transaction/sec