COMP 512 Phase 2 Report

Yiwei Xia and Marie Payne

November 24, 2017

1 System Architecture/Special Features

Our original system consisted of 1 client, 1 middleware, and 4 RMs, one each for flights, cars, hotels, and customers.

In our system, implementing locks and transactions required a change to the ResourceManager interface. However, the original inteface was good when used to communicate between the middleware and the RMs. Instead of modifying it and forcing to accomodate functionality it didn't need, a new interface was created called TransactionalResourceManager, which implemented the start, commit, abort functions. Naturally, there is also a new TransactionalMiddleware, which accepts the new start, commit, abort functions.

In our system, locking is managed by a LockManager. This manager is different from the provided one in that it doesn't differentiate between timeouts/deadlocks and acquired/redundant locks. When the middleware requests a lock on an object, it simply returns true within the timeout (10 seconds) if the lock has been acquired, or false once that time limit has been reached. This puts the responsibility of detecting deadlocks on the middleware. Client is not aware of locks. Locks are acquired by the middleware automatically as needed for each function call, and released after commit/abort, as per 2PL.

Transactions are managed by a TransactionManager. The TransactionManager has a stack of "requests" for each transaction, and supports 4 functions, start, commit, abort, and addRequest. The stack of requests keep track of the changes needed to revert the RMs back to their original states in case of an abort. Everytime the middleware performs a modification, i.e. a write, it calls addRequest to and pushes opposite request to the stack for that transaction in the transaction manager.

For example, if the client requests createResource(), the transactionalMiddleware will first lock the necessary resource. Then, it will create the resource on the appropriate RM, and then will push a deleteResource() request to the transaction stack in the transaction manager. If the transactionalMiddleware were to fail at this point, and was irrecoverable, the change would have effectively been committed. However, since we're assuming no failures, this is not a problem. Now, if a commit message is sent, the changes are already in the RMs, the transacation manager clears the stack for that transaction, and no more work is needed. However, in the case of a deadlock/abort situation, the transaction will pop every request in the stack, restoring the RMs into their original condition. Because 2 phase locking is used, popping the entire stack is guaranteed to return the RMs to their original conditions.

In order to push functions onto stacks, the command pattern was used to create two sets of classes in accordance to the the ResourceManager and TransactionalResourceManager interfaces.

With the way aborts work, it was necessary to create "doubles" of each function. If the TransactionalMiddleware create() function were to push the TransactionalMiddleware delete() function onto the stack, popping that delete function would push a create() function onto the stack, making an endless loop. Instead, the TransactionalMiddleware create() function pushes the Middleware delete() function onto the stack, whose operations are final (the same way they were implemented in project one).

2 Performance Analysis Methods

The performance analysis we conducted on our system included two parts; the first part involving one client and the average response times of different API methods in the system, the second involving muliple clients and varying loads to analyze the response times. Since we're focusing on response times, we want to avoid deadlocking the system as much as possible, which can be done with multiple clients by only using one method, query, which has no possibility of deadlock even called in concurrence (multiple read locks can be acquired at once, in our implementation).

The first part of the performance analysis simulated one customer on a loop. Every iteration of the program would choose a method and call it N times to produce an average, and print the response time on the customer side. Doing this narrows down the time spent at each stage in the method call on average, and can thus be used to figure out the bottleneck of the system and analyze where measures to increase efficiency can be implemented.

The second part of the analysis simulates multiple customers in a number of threads, and calls methods selected so as to avoid creating a deadlock in a loop. The load is then varied from 1 to 10 (the threads sleep at a time interval adjusted to reflect this) and the average response time recorded. This is then plotted using matplotlib in Python to illustrate the results (each line is for the number of customers, best-fit). This part of the performance analysis is meant to determine the saturation point of the system, with some level of concurrency. However, this test uses an optimal system with no occurrence of deadlocks, when in reality this system will deadlock somewhat frequently when the number of client threads running increases.

3 Performance Analysis Results

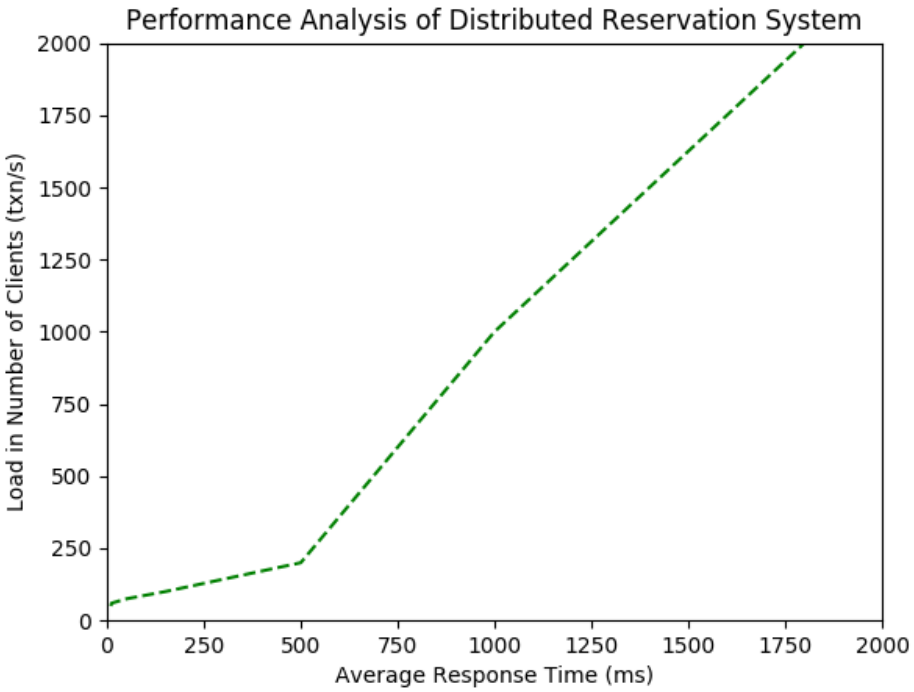
The results for phase 1 of the performance analysis testing can be resolved in the following table:

Method Call:	Client Side (ms)
start	5.43
commit	5.48
abort	5.49
createResource	10.64
updateResource	10.22
reserveResource	7.74
deleteResource	8.85
queryResource	5.52
uniqueCustomerId	5.56
createCustomer	10.78
queryCustomer	11.75
deleteCustomer	10.60
customerAddReservation	10.94
customerRemoveReservation	16.44
itinerary	42.20

Figure 1: Averages of method calls in the system in
milliseconds, using one client and ideal conditions.

The method calls were performed in ideal contexts. The itinerary method was called using two resources chosen at random (this is sufficiently varied for our purposes, however it should be noted that using a larger itinerary will scale the average response time linearly with the resource managers used). Each average was taken by running the method call 100 times and averaging the total time, which ended up being overly cautious and unnecessary but smoothed out any minor discrepancies. Running the loops multiple times revealed a standard deviation of +- 0.30 ms, roughly (more testing could be done at this phase to determine the exact standard devation, though this precision is not necessary in the preliminary development phases of this system).

The results for the second phase of the analysis are demonstrated in the figure shown:



The only method used in the trials was queryResource, with a lot of instantiated resources that get queried randomly so to prevent the threads from simultaneously accessing the list data structures (this throws a data corruption error). Threads sleep before checking if responses are received, so ¡10 ms response times are not achievable.

4 Performance Analysis Discussion

For the first part of the performance testing, we used one client on a loop and took the average response time on the client side of each method available in the API. Several of

the methods hovered around 10ms or below on average, with the exceptions of customerRemoveReservation and itinerary. These two methods used the most TCP calls compared to the other methods, which were more localized to one resource manager and performed checks against locally stored hashmaps and mutable lists. The performance of itinerary depends on a couple of variables, such as the number of resource managers involved, the number of resource IDs involved, and the size of the mutable map of resources being sent to the middleware server. The testing protocols used randomized two resources equally chosen from the three resource pools, so the reservable item mutable map being passed was a constant size, though the number of resource managers involved could vary. The standard deviation for this method was also larger, but if we accounted for the number of resource managers involved it would scale down to the same margin as the other method calls. More thorough performance testing could be used, but it is likely that the bottleneck of the system is the TCP implementation, specifically the queue the replies are stored in. Since TCP is necessary to operate over a network, this is an acceptable overhead for our purposes.

For the second part of the performance analysis testing, it was observed that with a higher number of active clients, the standard deviation varied a lot more (because of the nature of threads, it isn't guaranteed that the method call terminates before switching contexts in runtime, and a higher load made this more volatile). The number of clients is proportional to the amount of load on the system, with an increasing number of clients bringing an increasing load. The response times were steady from 1 to 200 clients and began to exponentially increase around 200. We can conclude that the system remains efficient for loads up to 200 clients, but shows significant strain as the load increases beyond that point.

5 Conclusion

The implementation of locking mechanisms through a middleware server promotes concurrency control of the system. The bottleneck of the system appears to be the TCP calls sending requests over the network. A more efficient implementation would be to perhaps use a hashmap to store replies in the TCP queues, instead of unneccessarily iterating over all the elements.

6 Bibliography

All the theory in this report came directly from the lecture slides on the course website or the recommended textbook, Distributed Systems: Principles and Paradigms by Andrew S. Tanenbaum and Maarten Van Steen, Second Edition, Pearson and Prentice Hall, Amsterdam, 2007.