COMP 512 Phase 2 Report

Yiwei Xia and Marie Payne

November 14, 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 Results

The performance analysis was by creating N customers in different threads, and having each of them submit transactions. Each of the N customers would loop and perform random actions, which included creating/deleting customers,resources and creating reservations(50%), sleeping for a small amount of time (50%), or committing their transactions(10%). They start off with 6 resources and 6 customers, which they are free to randomly add to/modify/delete. For 1 customer, these transactions would always complete. However, with 2, this quickly deteriorates, with about 1/2 of them deadlocking. With 5 customers, usually none of the customers would complete.

The test was modified so that each customer would submit transactions with only one request in it. The customers would sleep 10 ms after each commit. However, since some requests required more than one lock, deadlocks were still possible, and therefore response times were still near zero when the amount of customers was increased to an amount that made nearly all requests deadlock.

When tests were instead changed to have customers only query customers, with no possiblity of deadlock, response times were as follows. Threads sleep before checking if responses are received, so <10 ms response times are not achievable.

50 customers: 10 ms
60 customers: 12 ms
75 customers: 50 ms
100 customers: 150 ms
200 customers: 500 ms
1000 customers: 1000 ms
2000 customers: 1800 ms

3 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 different threads searching through the replies queue to find if their response has been received, as this is being done via TCP. Perhaps changing the implementation of the replies queue to some sort of hashmap would allow for faster reads.

4 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.